iGamePlay

A Revolutionary Gaming Experience

6.111 Final Project

Tom Wilson, Martijn Stevenson, Kale McNaney

Professor Anantha Chandrakasan, TA David Milliner

May 13, 2004

Table of Contents

Table of Figures

Introduction

The purpose of this project was to explore the capabilities of the new Xilinx lab kits by creating an innovative video game incorporating audio processing tools. iGamePlay is a combination of retro video game mechanics and gameplay with a new twist: gameplay elements are affected by sound cues derived from processing an input audio signal. iGamePlay features enemies which respond to the beat of an input song. The final system includes support for single-player and two-player cooperative and head-to-head modes, smooth page-buffered 640x480 VGA video, and a highly satisfactory beat detection algorithm.

Overview

The iGamePlay system has three major modules: audio processing, user input/game logic, and video. Figure 1 shows a schematic of the overall system.

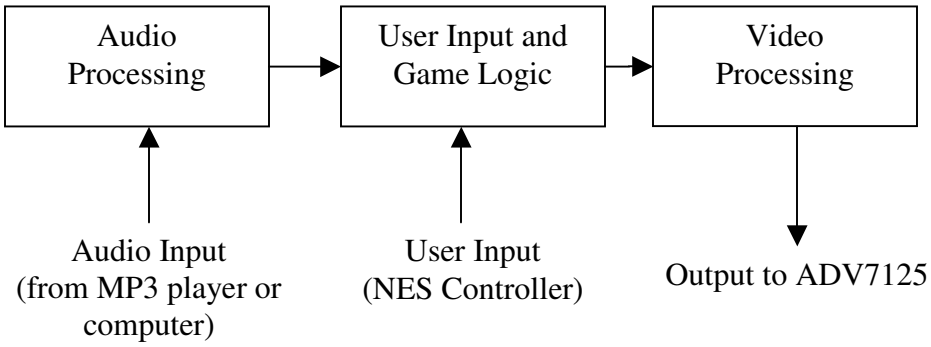


Figure 1. Overall System Schematic

The audio processing unit takes in audio input from an external source. Using the AC97 codec, the audio data is converted to a digital signal. A custom beat detection algorithm then extracts the beats from the music. This information is passed to the user input and game logic unit. This unit combines the audio cues with the user input to coordinate the internal state of the game. This includes tracking the speed and position of all objects in the game world. The game logic unit sends information about the game state to the video subsystem, which processes it and draws the game to the screen. The individual subsystems are discussed in more detail in the following sections.

Audio

The iGamePlay audio component detects the beats that allow the enemy to move to the music. Using the lab kit's on-board AC97 Codec chip and a simple energy comparison algorithm, the audio module was able to detect a range of beats from simple base drums more complicated techno and rap songs with strong down beats. The module would then relay the beat signal to the game logic to influence enemies' speed.

Getting Digital Audio Data

AC97 Codec

Before analyzing the music to detect beats, the analog source needed to be sampled with an A/D converter. Conveniently, the lab kit houses an AC97 codec chip that can do most of the sampling work. AC97 is a general PC audio codec specification published by Intel Corporation. You can view it here: http://www.intel.com/labs/media/audio/.
The 6.111 lab kits contain the National Semiconductor LM4550 implementation of this specification. The LM4550 is in full compliance with respect to the features that it implements, but by no means does it implement every feature described in the Intel specification. For the purpose of iGamePlay, the LM4550 was more than adequate.

The LM4550 audio codec is a powerful tool for digitizing analog inputs, processing analog inputs with its 3D sound circuitry, or playing back digitized samples from a ROM, however it cannot function without an AC97 digital controller. The controller is used for resetting the codec (necessary on power-on), reading digitized data from the codec's A/D, passing in digitized data to the codec's D/A, and modifying internal configuration registers (more on this later).

Frame Protocol

All communication between the AC97 codec and its controller occurs along two 1-bit serial data streams named sdata_out (from controller to codec) and sdata_in (codec to controller). These names are controller relative. Each stream is broken up into segments called frames. Each frame is made up of 256 sequential bits. The start of a frame is signaled by a control signal named SYNC that is generated by the controller. In a properly working controller, SYNC should transition from low to high once every 256 bits. In compliance with the specification, SYNC should have a duty cycle of 6.25%, in other words, it should remain high for 16 of the 256 bits. Any transition of SYNC from low to high *before* 256 bits have been recognized by the codec will be ignored.

How often should a bit be sent across the 1-bit channel from the controller to the codec, and how often do bits come back from the codec? After coming out of reset, the codec will generate a bit clock at 12.288 MHz (half the 6.111 lab kit system clock speed). On the rising edge of the bit clock, the codec samples SYNC and the controller sends the first bit of the new frame. On the next falling edge, the codec samples the bit.
Simultaneously, on the rising edge of the clock, the codec sends the controller 1 bit of information for the current frame. The controller is expected to sample the bit on the next falling edge of bit clock See Figures 2 and 3 below for timing diagrams.

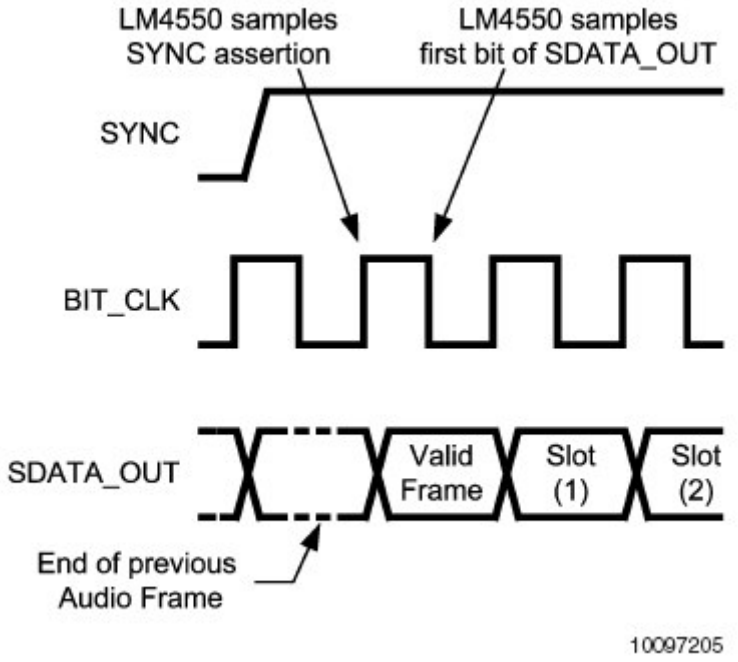


Figure 2. AC97 Output Frame (Controller to Codec)

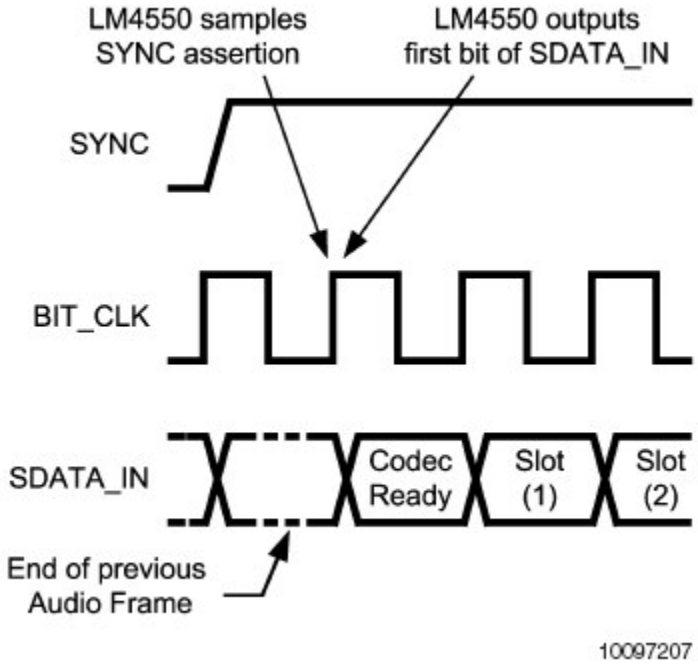


Figure 3. AC97 Input Frame (Codec to Controller)

So the controller and codec communicate simultaneously across 2 1-bit channels by sending frames. What kind information is contained in each of these frames? The 256 frame bits are separated into 13 slots. The first slot is a 16-bit tag slot and the remaining

12 slots contain 20 bits each, as shown in Figure 4 below. The tag slot is essentially meta-data and describes whether or not there is valid data in any of the remaining 12 slots. The information in the remaining 12 slots differs between sdata_in and sdata_out and is described below.

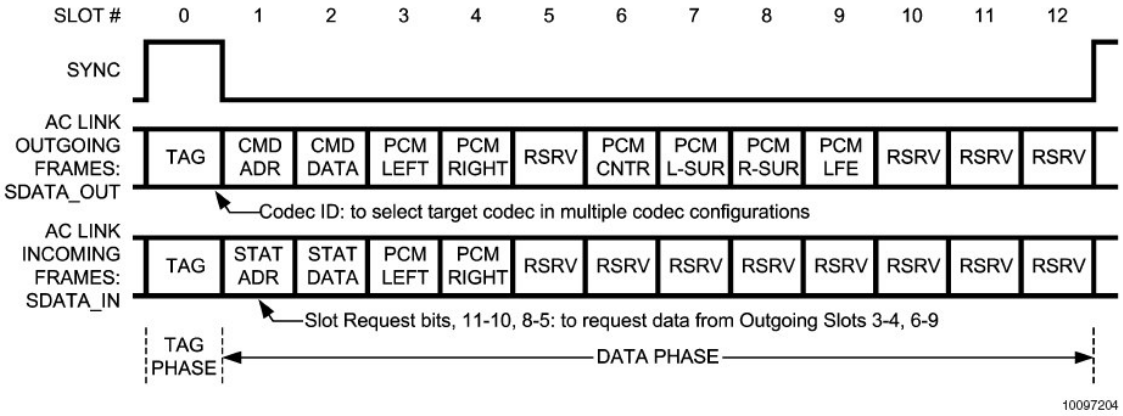


Figure 4. AC97 Frame consisting of 13 slots

Frames from Controller to Codec

Frames sent from the controller to the codec contain two valuable pieces of information. First, there is information used to write or read internal codec configuration registers. Second, there is digital audio data begin sent for processing by the codec's D/A. Eighteen bits of digital data can be passed to slots 3 and 4 (left and right channel respectively) or to slots 6 and 7 for D/A conversion. More important, however, is the information passed to configure the internal registers. Without properly configuring the codec, it will not be able to perform any of its powerful audio processing functions.

Configuration Registers

The AC97 specification contains a description of about thirty internal configuration registers that determine how the codec will operate. Several of these registers are configurable by the controller and, in many cases, must be written to for the codec to function. Among the most important are the registers which apply gain or attenuation on incoming and outgoing signals, and the selector mux which decides which input to listen to for incoming analog information. By default, all gain/attenuation registers are set to mute and the selector mux will listen on the microphone line input. Therefore, to see the desired functionality of the codec, all gain/attenuation registers in the critical path between inputs and outputs must be un-muted. The gain/attenuation registers can be configured between 0 dB and 22.5 dB with a 1.5 dB resolution. In addition to the gain/attenuation registers, the codec contains an 18-bit A/D converter with variable sampling rate ranging from 4 kHz to 48 kHz with 1 Hz resolution and again, there is an internal register that can be set to change this sampling rate. The iGamePlay system used the stereo line inputs with a sampling rate of 48 kHz along with the headphone outputs.

Frames from Codec to Controller

Frames sent from the codec to the controller contain two valuable pieces of information. First, the codec can send information to the controller in response to a register read. This

information would include an echo of the register address requested for the read in slot 2, and the value stored in that address in slot 3. Second, the codec will pass back left and right channel digitized PCM data in slots 3 and 4 respectively.

Analyzing Energy in the Music

The beat detection algorithm used for iGamePlay is a modified version of an energy analysis method described here:
http://www.gamedev.net/reference/programming/features/beatdetection/ .
The basic intuition behind the algorithm is to find sections of the music where the instant energy in the signal is greater than some scaling of the average energy of the signal over the previous approximate second of music. The assumption made is that the instant energy in a signal will be much greater on the beat than between beats. This assumption is reasonable for songs with heavy down beats and little mid and high frequency "noise".

An outline of the original algorithm is as follows:

1. Collect 1024 16 bit digital samples in a buffer a[n].
2. Compute instantaneous energy e[n] by: $e[n] = \sum_{0}^{1024} (a[n])^2$.
3. Compute average 1 second energy <E> by summing over a history buffer that contains the 32 previous values of e[n] by: $< E >= \frac{1}{32} \sum_{0}^{32} (e[n])^2$
4. If e[n] > c*<E>, detect a beat.
5. Flush out oldest value e[n] from history buffer and insert newest e[n].
6. Repeat.

The digitized sample values coming from the A/D are 18 bits 2's complement numbers which correspond to values ranging from (-2^15) through (2^15 – 1). For iGamePlay, these values were all shifted to positive numbers between 0 and (2^16-1) by flipping the most significant bit. By normalizing to all positive values we were able to remove the squaring operations in the above formulas. This is much more efficient because it does not require the use of an 18x18 bit multiplier and a 46x46 bit multiplier. Also, instead of diving by 32 to calculate <E>, the least significant 5 bits were removed from <E>. The constant 'c' in step 4 was discovered empirically to be approximately 3.

This method, although sensitive to high energies resulting from higher frequencies, worked very well at tracking a various inputs. Basic tests involved songs with only bass drum beats at changing tempos, songs with bass drum and high-hats, and songs with a bass drum, high-hats and a bass guitar. The system tracked the beat perfectly for these inputs. More rigorous tests included techno and rap songs with strong downbeats. Although these songs had energy in other frequency bands, the algorithm performed very well and tracked the beat most of the time, losing its way from time to time, but usually getting back on track. Finally, we tested rock music with symbol crashes and high frequency guitar solos. The system did not track very well because this method for beat

detection is “colorblind”. In other words, it only detects a threshold difference between the energies in the music, but does not know what frequency band the energy lies in. A more robust method would involve a frequency analysis of the incoming waveform using FFTs, filter banks or correlation functions.

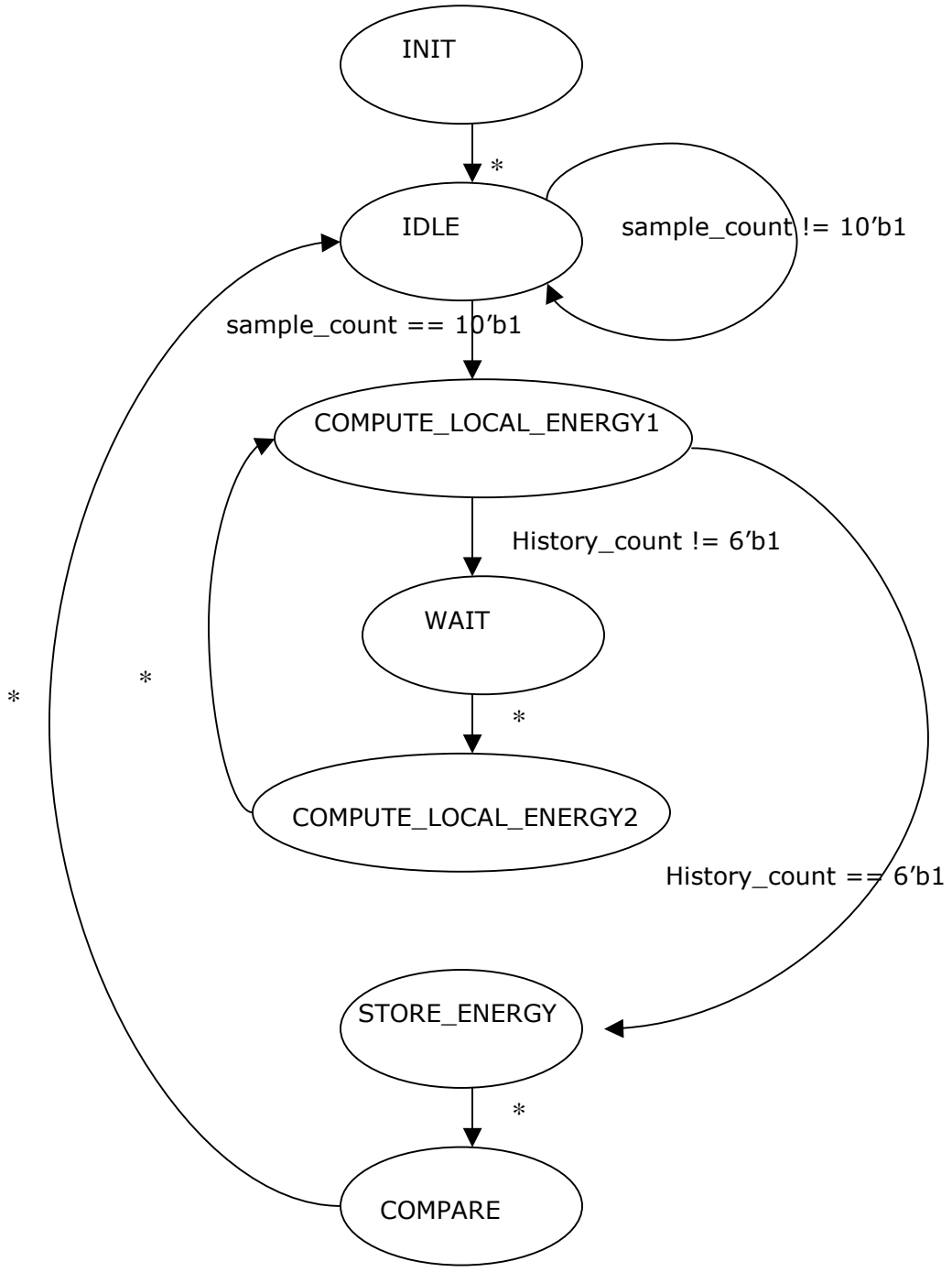


Figure 5. Sound Energy FSM Diagram

Input / Game Logic

There are two obvious tasks in this section of the iGamePlay project. The input component must interface with users, ensuring that each player has a direct say in the flow and variation of any single game. The game logic component must preserve and update the continually varying game state, driving information about crucial game objects to the video output unit.

User Input

For the iGamePlay project, we decided to use two Nintendo Emulation System (NES) controllers as input devices. The gamer generation has extended familiarity with this device, and its use is straightforward. Beyond that, the system interface for NES controllers is rather simple. Unlike analog controllers and joysticks, the NES joy pad only has buttons. At any time, each button is either idle (off) or depressed (on). The only task that the user input module must perform is to sample all eight NES buttons at some frequency.

Figure 6. Physical Interface to NES Controller

The physical interface to the NES controller is a simple one. Only five wires connect each NES pad to the iGamePlay kit. There are four controller inputs (power, ground, latch and pulse) and one controller output (data). Since there is only one data line, buttons states must be transferred serially. An input finite state machine within the project handles all controller communications, according to the input protocol shown in Figure 7. A latch signal from the input FSM initiates a transaction sixty times every second. Latch is held high for twelve microseconds, after which the first data value (“A”) is guaranteed to be valid. After reading the value, the input sends seven six-microsecond pulses out on the pulse wire. After each one, it reads a new button value from the data line, which the controller will drive there in the order “B,” “Select,” “Start,” “Up,” “Down,” “Left,” and “Right.” See Appendix D for Verilog code that implements this communication protocol.

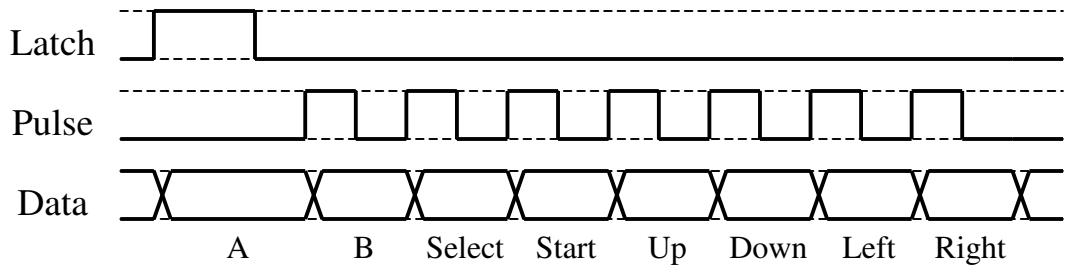


Figure 7. Nintendo Emulation System Input Protocol

A major concern with any asynchronous user input to a digital system is registering and steadying user data. In this case, the user presses keypad buttons that may bounce after the first touch until they finally settle. To prevent asynchronous data from ruining the careful timing of the digital system, the inputs are first registered on the system clock. In some scenarios, it is helpful to also build a button delay, to ensure that a user meant to perform the chosen action. For example, when the user chooses a menu option, the system waits for 100 ms of valid user data to ensure that any button press is intentional. However, such "debouncing" of user input is not always beneficial: when a user wants to shoot a missile, it is important that the action is completed immediately. Because demand for button debouncing varies even across different uses of the same keys, debouncing is performed locally as needed, and not dealt with in or around the input FSM.

Game Logic

The game logic portion of the iGamePlay project is the section of code that is the least native to a hardware implementation. The game logic component is a large finite state machine built on several other game object FSMs. The interactions between the modules are often complex and tedious in hardware (i.e. collision detection), and as a result take up a lot of FPGA realty. Figure 8 shows an overview of the game logic component.

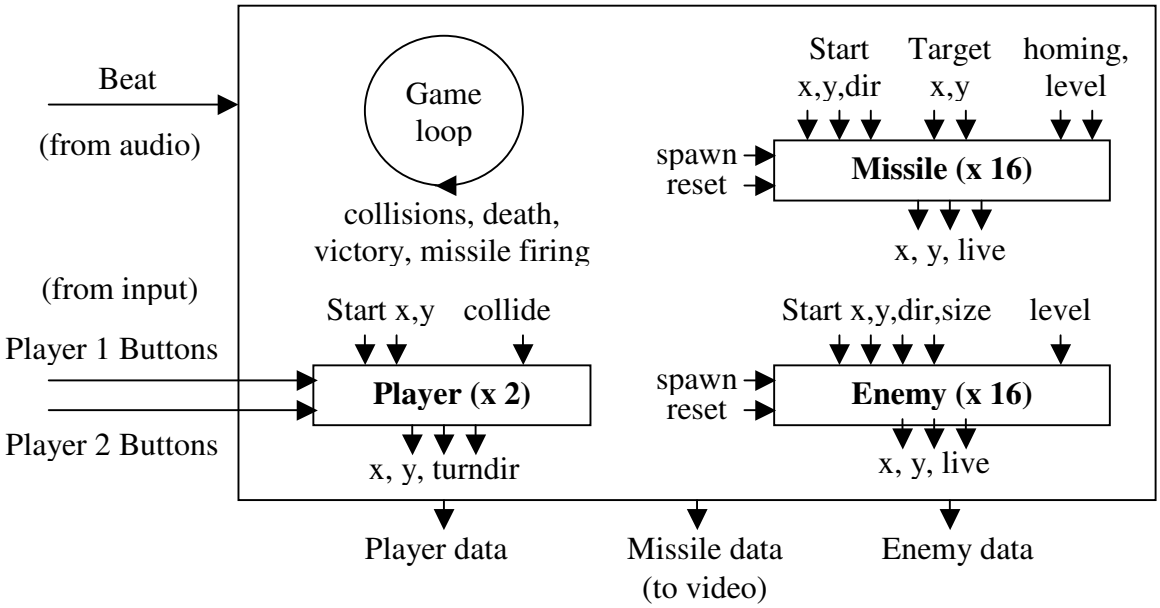


Figure 8. Game Logic Overview

The minor FSMs for players, missiles and enemies are all primarily movement oriented. They control the positions of game objects in response to user input in the case of the players, a beat signal in the case of the enemies, and a target's position in the case of the missiles. However, such movements are predicated on the "liveness" of objects. Hardware restrictions do not allow the allocation of many missile or enemy structures, and as a result the viable number of missiles and enemies were experimentally capped at sixteen. At no one time will all enemies and missiles be active. Instead, the game logic cycles through idle (expired or collided) missiles, keeping track of the current missile and enemy indices, and reactivating them as new game objects with the spawn signal and a set of initial coordinates. When it is time for the game objects to disappear again (whether via expiration or collision), missiles and enemies are given an active reset signal.

The operations that require knowledge about more than one game object all take place in the main game loop. The most visually prominent and also most difficult multi-object actions are collisions, which can lead to any of the following outcomes: player bounce, player death, enemy splitting, enemy death, and missile death. The reason why collisions are so tricky is that each collision requires several large comparators to check for proximity, and the total number of collision checks is in the hundreds. To solve this problem, the iGamePlay system uses a single collider module that checks proximity of its inputs. The system examines collision candidates in sequence, an inefficiency that is made possible by the high system clock speed. At a clock frequency of 27 MHz, and a movement rate of one tick per 100 milliseconds, the system has many thousands of clock cycles to check collisions between movements.

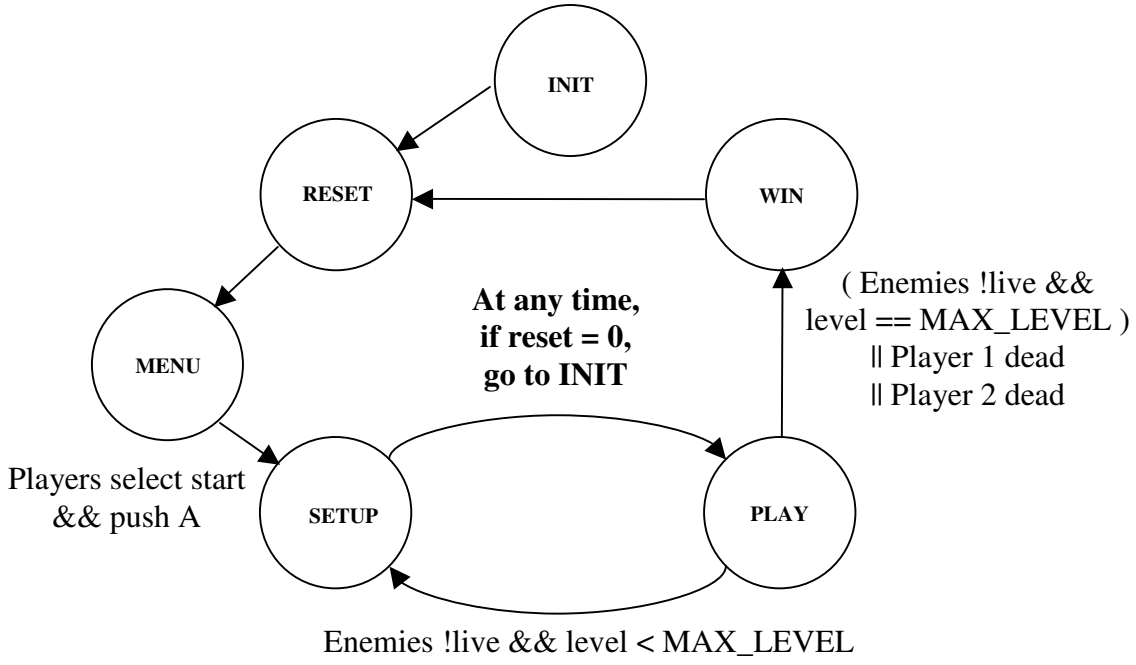


Figure 9. Game Loop Finite State Machine

While the focus of the game logic is mainly on game play and game content, the main loop also performs several functions that gamers might take for granted in games. See Figure 9 for a diagram showing the transitions of the game loop FSM. The system deals with game play aspects in the PLAY loop: collisions (as described above), player death and victory, and missile firing. Beyond this, the system keeps track of levels, changing the level in the transition from PLAY back to SETUP. With each higher level, missiles and enemies speed up, presenting a larger challenge to the player, whose own speed remains unchanged. SETUP is a good place to provide level information to the video system, though project time constraints forced us to bypass this feature. The system also implements a Mario-style menu interface, allowing the user to switch a selector between editing game mode and starting a match. Finally, the WIN state provides an opportunity to present game results to the video system, which could output whether player one, player two, or the enemy won the match. Due to project time constraints, this implementation did not contain such a summary screen, instead skipping through to the RESET state.

Video

The video subsystem is responsible for displaying the game content to the screen. The subsystem takes input from the game logic subsystem and uses it to generate the images displayed to the user. This system displays 24-bit color 640x480 VGA video at 60 frames per second. Figure X below shows a general schematic of the video subsystem.

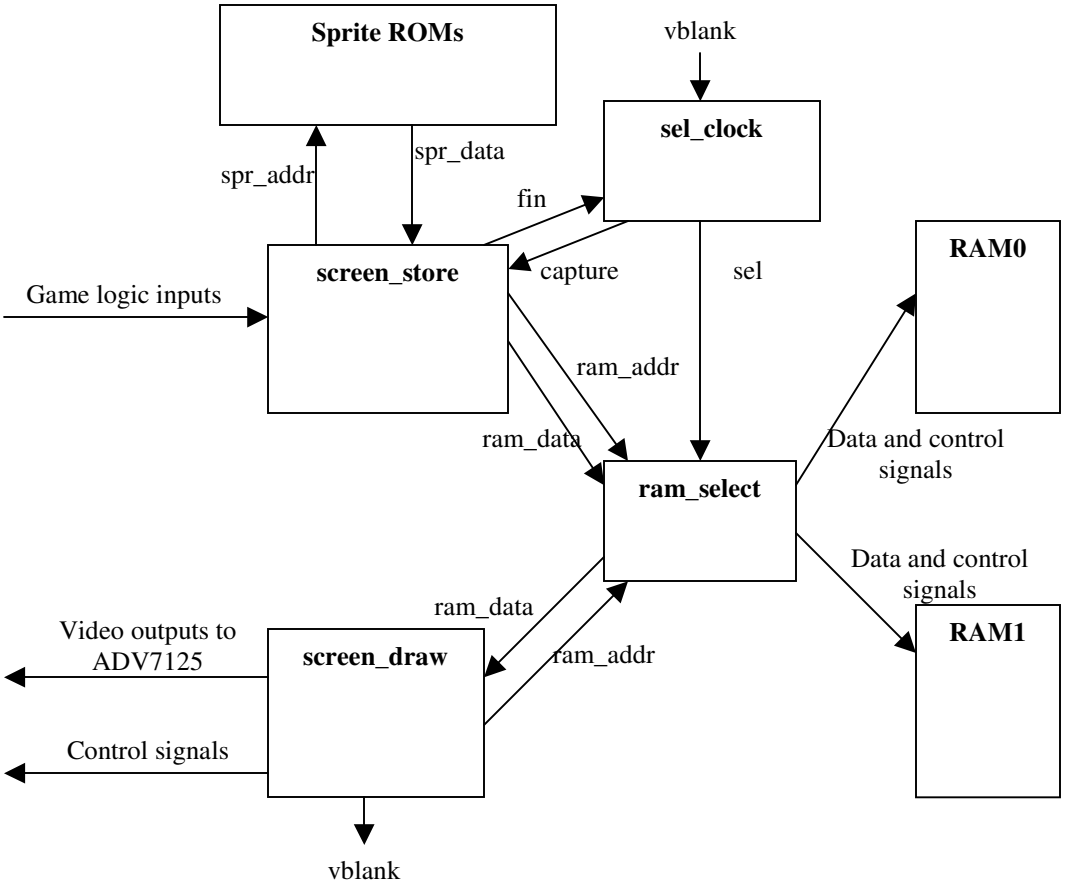


Figure 10. Video Subsystem Schematic

The system implements a page-buffering scheme using the two ZBT SRAMs included in the lab kit. Page-buffering simply means that while the contents of one RAM are being displayed to the screen, the other RAM is being filled with the contents of the next screen. This results in smoother video performance, greatly reducing the visibility of any glitches that may occur. The video subsystem is comprised of five modules: the SRAMs, the ram_select/sel_clock unit, a bank of sprite ROMs, the screen_store unit, and the screen_draw unit. These modules are discussed in greater detail in the sections that follow.

ZBT SRAMs

The ZBT SRAMs included in the lab kit are 512Kx36 SRAMs. The RAMs serve as the page buffers for the video system. Each RAM contains a snapshot of the screen at a given instant. In particular, the RAMs are addressed using a linear combination of the screen coordinate values (x and y) for a given pixel, and contains a 24-bit value representing the color of that particular pixel. For more detail about interfacing with the ZBT RAMs, please see Appendix A.

ram_select and sel_clock

These modules implement the buffer switching for the page-buffering mechanism for the iGamePlay system. Ram_select interfaces between the drawing systems (screen_draw and screen_store) and the RAMs. When the input sel is high, the ram_select module assigns RAM0 to the screen_draw system and RAM1 to the screen_store unit. When sel switches values, the RAMs flip. Thus, while screen_store is writing data to one RAM, screen_draw can display the data from the other RAM.

Sel is controlled by the sel_clock module. Sel_clock also takes in the vblank signal from screen_draw and the fin signal from screen_store. When the screen_store unit finishes writing data into a RAM, it asserts the fin signal. If this signal is asserted while vblank is low, sel_clock inverts the value of the sel output and asserts the capture signal. This causes the ram_select unit to flip the RAMs. When the capture signal is asserted, the screen_store unit begins storing another screen image to the RAM. It is important that sel_clock only trigger a page flip while vblank is low. This ensures that the transition occurs while the screen is being blanked, preventing viewers from seeing the refresh.

Sprite ROMs

In order to display complicated characters more easily, the video subsystem uses an array of ROMs containing sprite images. Just like the ZBT RAMs, these ROMs are addressed by pixel coordinates and contain the color of the corresponding pixel. If the current position on the screen falls within a displayed sprite, the screen_store system indexes the appropriate ROM and selects the value stored at the current location to store in the page buffer. The iGamePlay system uses sprites for players, enemies, missiles, and for the title screens.

Screen_draw

The screen_draw unit generates the control signals necessary to drive the VGA monitor. These signals include the vertical and horizontal sync and blank signals. Additionally, the screen_draw unit reads the color value of the current pixel from memory and presents it to the ADV7125 chip. This chip performs the digital to analog conversion necessary to display the data on the VGA monitor.

Since the screen_draw unit functions as a continuous signal generator, the unit is implemented as several separate Verilog always blocks. Since these blocks execute in parallel, this guarantees that the control signals remain appropriately in sync.

Displaying VGA video starts by generating a pixel clock. Every time the pixel clock pulses high, the current data presented to the ADV7125 will be latched in and converted. A complete line of pixels is preceded by a horizontal sync pulse. Similarly, a complete frame of lines is preceded by a vertical sync pulse. All sync pulses are surrounded by a blanking interval. These intervals are know as the front and back porches. The exact duration and timing of these pulses great depends on the resolution and refresh rate of the

monitor in use. For a table of the timing parameters used in the iGamePlay system and the Verilog code for the screen_draw module, please see Appendix B.

Screen_store

The screen_store unit is the most complicated unit in the video subsystem. This unit takes input from the game control system. Using this input, the screen_store unit determines what color needs to be displayed at each pixel on the screen and stores these values to the ZBT page buffer.

The basic structure of the screen_store unit is as follows. The unit waits in an IDLE state until the sel_clock unit asserts the capture signal. Starting from the upper left corner of the screen, screen_store examines the input to determine if the current pixel is contained within a sprite to be displayed on the screen. If so, the unit uses the current pixel's coordinates to address the appropriate sprite ROM and retrieve the pixel's color value. This value is then written into the ZBT page buffer. If no sprite is present at the current location, the unit writes the background color into the ZBT page buffer and examines the next pixel on the same line. Once the line is completed, the unit moves on to the next line until the entire screen has been stored to the ZBT RAM. The unit then asserts the fin signal, and waits until the sel_clock unit re-asserts the capture signal.

Possible improvements

In any project, time constraints invariably preclude the development of additional features that would improve the system. The iGamePlay project was no exception. There were many things that would have been nice to implement. For example, having more time to experiment with improved sprites would have greatly enhanced the visual experience. Similarly, we had wanted to incorporate brief information screens between levels and after victories but ran out of time. Other features that would have improved the system but were left out included background images, powerups, and additional audio cues.

Conclusion

This project has been an enlightening experience for us. We took away several lessons from our long hours spent poring over our code in the 6.111 lab. One important rule is to always back up project files whenever *anything* works. The smallest change in behavior can be an important breakthrough in the project, and before you realize it, you may have eradicated your progress with a new version. Another classic computer science rule that we found to be important is to think about a design before implementing anything. Temporary "fixes" usually cause more problems than they solve, because they bring along all kinds of new, unplanned inadequacies. When a single project compilation can take up to twenty minutes, it is important to realize what you're changing, and how it will affect the system. Last but not least, we found it helpful to design our project timeline so that each week's work is equally weighted for each team member. It is a problem if two people finish their designs and have to wait on a third before the project can continue. A

good timeline will also encourage team members to start implementing early, which is always good for the project.

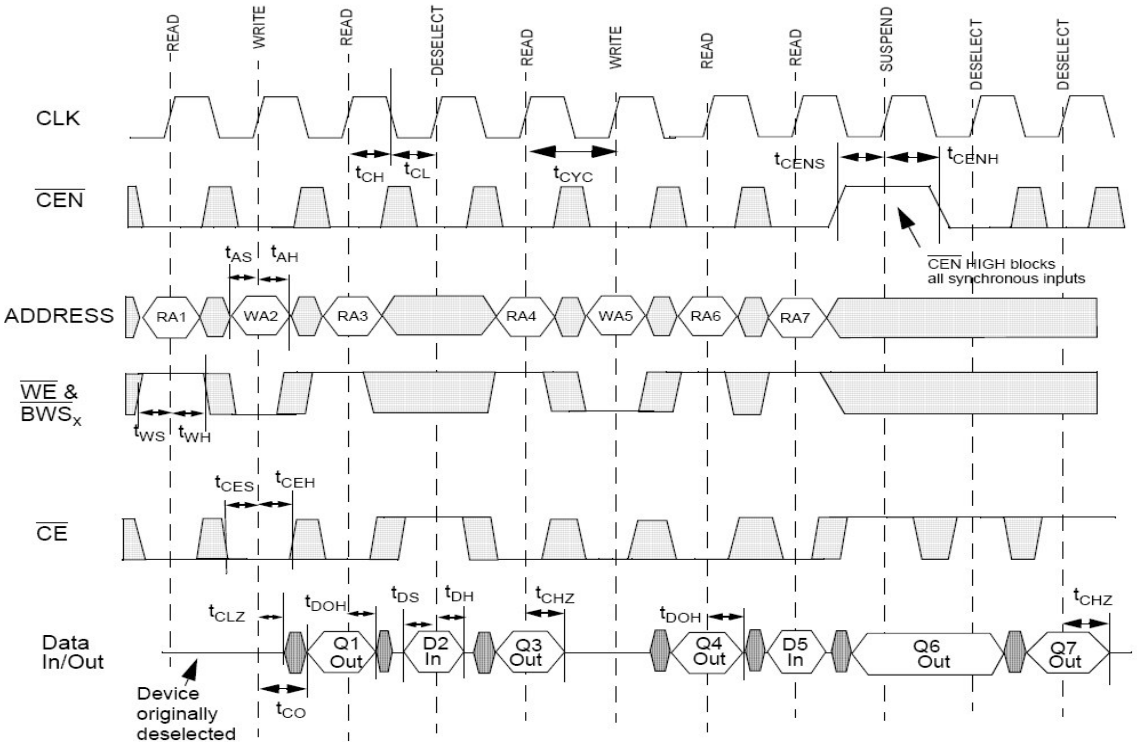
Although there were many features that we did not get a chance to implement, the final system was still a highly playable and enjoyable game. It features all of the intensity and addictiveness of classic arcade games, while managing to inject an element of freshness by incorporating digital signal processing. In our eyes, the iGamePlay project has been an unqualified success.

Appendix A: ZBT RAMs

The following table shows the control signal values needed to perform a read or a write operation on one of the ZBT SRAMs included in the kit. Note that these signals are not the signals defined on the pins of the chip but rather the interface signals defined in the labkit.v top level design file.

Signal name	Read Operation	Write Operation
adv_ld	0	0
cen_b	0	0
ce_b	0	0
oe_b	0	1
we_b	1	0
bwe_b	XXXX	0000

Because the ZBT SRAMs are pipelined, they can greatly improve access speeds over traditional SRAMs. However, this pipelining feature also makes the chips more complicated to use effectively. In general, the rule is that addresses and control signals are presented at time t and data is presented at time t + 2. In other words, a read operation will return data to the bus two clock cycles after the chip is presented with an address, and a write operation will store whatever data is on the bus two clock cycles after the write request. The figure below comes from Cypress' web site, and illustrates these timing features.



Appendix B: screen_draw

```
module screen_draw ( clock, reset, ramdata, ramaddr, red_out, green_out, blue_out, sync_b, blank_b,
hsync, vsync, gamedata, vblank);

    input clock;
    input reset;
    input [35:0] ramdata;  //[23:16]= red, [15:8] = green, [7:0] = blue
    output [18:0] ramaddr;
    input [19:0] gamedata;

    output [7:0] red_out;
    output [7:0] green_out;
    output [7:0] blue_out;
    output sync_b;
    output blank_b;
    output hsync;
    output vsync;
    output vblank;

    reg [10:0] row_count, line_count; //counters for pixels
    reg hsync, vsync, sync_b, h_c_sync, v_c_sync; //syncing signals
    reg hblank, vblank, blank_b; //blanking
    reg hsync0, hsync1, vsync0, vsync1; //Delay sync signals

    reg [7:0] red_out, green_out, blue_out;
    reg [9:0] spr_addr, count;

    //pixel clock: 25.17 MHz
    parameter H_ACTIVE =          640;        // pixels
    parameter H_FRONT_PORCH =   16;       // pixels
    parameter H_SYNC =                          96;          // pixels
    parameter H_BACK_PORCH =    48;       // pixels
    parameter H_TOTAL =           800;        // pixels

    parameter V_ACTIVE =          480;        // lines
    parameter V_FRONT_PORCH =  11;       // lines
    parameter V_SYNC =         2;  // lines
    parameter V_BACK_PORCH =                31;          // lines
    parameter V_TOTAL =           524;        // lines

    reg [9:0] pixel_row;
    reg [8:0] pixel_line;

    //Address = x + 800*y
    assign ramaddr = row_count[9:0] + (800 * line_count[9:0]);

    wire [9:0] x, y;
    assign x = gamedata[19:10];
    assign y = gamedata[9:0];

//DRAWING LOGIC
always @ (posedge clock or negedge reset) begin

            if (!reset)
```

```
            begin
              red_out <= 8'h00;
              green_out <= 8'h00;
              blue_out <= 8'h00;
            end
            else
            begin
              //read data from ram
              red_out <= ramdata[23:16];
              green_out <= ramdata[15:8];
              blue_out <= ramdata[7:0];
            end
end

 // CREATE THE HORIZONTAL LINE PIXEL COUNTER
always @ (posedge clock or negedge reset) begin
            if (!reset)
            begin                                   // on reset set pixel counter to 0
                        row_count <= 11'h000;
            end

            else if (row_count == (H_TOTAL - 1))
            begin                                   // last pixel in the line
                        row_count <= 11'h000;               // reset pixel counter
            end

            else
            begin
                        row_count <= row_count +1;
            end
end

// CREATE THE HORIZONTAL SYNCH PULSE
always @ (posedge clock or negedge reset) begin
            if (!reset)
            begin                                   // on reset
                        hsync0 <= 1'b1;                     // remove hsync
            end

            else if (row_count == (H_ACTIVE + H_FRONT_PORCH - 1))
            begin                                                   // start of hsync
                        hsync0 <= 1'b0;
            end

            else if (row_count == (H_TOTAL - H_BACK_PORCH - 1))
            begin                                                   // end of hsync
                        hsync0 <= 1'b1;
            end
end


// CREATE THE VERTICAL FRAME LINE COUNTER
always @ (posedge clock or negedge reset) begin
            if (!reset)
            begin                                           // on reset set line counter to 0
                        line_count <= 10'h000;
```

```
        end

        else if ((line_count == (V_TOTAL - 1))&& (row_count == (H_TOTAL - 1)))
        begin                                   // last pixel in last line of frame
                line_count <= 10'h000;                          // reset line counter
        end

        else if ((row_count == (H_TOTAL - 1)))
        begin                                           // last pixel but not last line
                line_count <= line_count + 1;       // increment line counter
        end
end

// CREATE THE VERTICAL SYNCH PULSE
always @ (posedge clock or negedge reset) begin
        if (!reset)
        begin                                                                   // on reset
                vsync0 = 1'b1;                                                  // remove v_sync
        end

        else if ((line_count == (V_ACTIVE + V_FRONT_PORCH - 1) &&
                  (row_count == H_TOTAL - 1)))
        begin                                                                   // start of vsync
                vsync0 = 1'b0;
        end

        else if ((line_count == (V_TOTAL - V_BACK_PORCH - 1))           &&
                  (row_count == (H_TOTAL - 1)))
        begin                                                                   // end of vsync
                vsync0 = 1'b1;
        end
        end

// ADD TWO PIPELINE DELAYS TO THE SYNCHs COMPENSATE FOR THE DAC PIPELINE
DELAY
always @ (posedge clock or negedge reset) begin
        if (!reset)
        begin
                hsync <= 1'b1;
                vsync <= 1'b1;
                hsync1  <= 1'b1;
                vsync1  <= 1'b1;
        end
        else
        begin
                hsync1 <= hsync0;
                vsync1 <= vsync0;
                hsync <= hsync1;
                vsync <= vsync1;
        end
end



// CREATE THE HORIZONTAL BLANKING SIGNAL
// the "-2" is used instead of "-1" because of the extra register delay
```

```
// for the composite blanking signal
always @ (posedge clock or negedge reset) begin
        if (!reset)
        begin                                                   // on reset
                hblank <= 1'b1;                 // remove the h_blank
        end

        else if (row_count == (H_ACTIVE - 2))
        begin                                                   // start of HBI
                hblank <= 1'b0;
        end

        else if (row_count == (H_TOTAL - 2))
        begin                                                   // end of HBI
                hblank <= 1'b1;
        end
end


// CREATE THE VERTICAL BLANKING SIGNAL
// the "-2" is used instead of "-1"  in the horizontal factor because of the extra
// register delay for the composite blanking signal
always @ (posedge clock or negedge reset) begin
        if (!reset)
        begin                                                           // on reset
                vblank <= 1'b1;                         // remove v_blank
        end

        else if ((line_count == (V_ACTIVE - 1) &&
                  (row_count == H_TOTAL - 2)))
        begin                                                           // start of VBI
                vblank <= 1'b0;
        end

        else if ((line_count == (V_TOTAL - 1)) &&
                  (row_count == (H_TOTAL - 2)))
        begin                                                           // end of VBI
                vblank <= 1'b1;
        end
end


// CREATE THE COMPOSITE BANKING SIGNAL
always @ (posedge clock or negedge reset) begin
        if (!reset)
        begin                                           // on reset
                blank_b <= 1'b1;        // remove blank
        end

        else if (!hblank || !vblank)            // blank during HBI or VBI
        begin
                blank_b <= 1'b0;
        end
        else begin
                blank_b <= 1'b1;        // active video do not blank
        end
```

```
end

// CREATE THE HORIZONTAL COMPONENT OF COMP SYNCH
// the "-2" is used instead of "-1" because of the extra register delay
// for the composite synch
always @ (posedge clock or negedge reset) begin
        if (!reset)
        begin                                                   // on reset
                h_c_sync <= 1'b1;               // remove h_c_sync
        end

        else if (row_count == (H_ACTIVE + H_FRONT_PORCH - 2))
        begin                                                   // start of h_c_sync
                h_c_sync <= 1'b0;
        end


        else if (row_count == (H_TOTAL - H_BACK_PORCH - 2))
        begin                                                   // end of h_c_sync
                h_c_sync <= 1'b1;
        end
end

// CREATE THE VERTICAL COMPONENT OF COMP SYNCH
always @ (posedge clock or negedge reset) begin
        if (!reset)
        begin                                                           // on reset
                v_c_sync <= 1'b1;                               // remove v_c_sync
        end

        else if ((line_count == (V_ACTIVE + V_FRONT_PORCH - 1))
                        && (row_count == (H_TOTAL - 2)))
        begin                                                   // start of v_c_sync
                v_c_sync <= 1'b0;
        end

        else if ((line_count == (V_TOTAL - V_BACK_PORCH - 1))          &&
                  (row_count == (H_TOTAL - 2)))
        begin                                                   // end of v_c_sync
                v_c_sync <= 1'b1;
        end
end

// CREATE THE COMPOSITE SYNCH SIGNAL
always @ (posedge clock or negedge reset) begin
        if (!reset)
                begin                                           // on reset
                        sync_b <= 1'b1;                 // remove comp_sync
                end
        else begin
                sync_b <= (v_c_sync ^ h_c_sync);
                end
end
endmodule
```

Appendix C: Tips and Tricks

AC97 Codec Tips

- Cold resets reset all registers and should be used to restore the codec to its initial state. Warm resets are used for power saving modes. Unless you are running in low power modes, you can ignore the use of warm resets.
- When the codec comes out of reset (reset transition from low to high), the signals sdata_out and sync MUST be held low. Failure to do so will cause unexpected codec behavior that may be a vendor test mode. These test modes are not documented well in the LM4550 specification.
- Look at sdata_out and sdata_in on the logic analyzer often. Finding early problems in reading and writing will help speed up the debugging process.
- At 12.288 MHz, 1 bit is approx. 81 ns on the logic analyzer.

Xilinx Tips and Hints

- Copy your project on the local drive of the machine you are on instead of the shared drive. Xilinx will compile much faster. Don't forget to place a copy back in your shared folder once you finish because there is no guarantee you will get the same computer!!
- A FATAL GuiUtility error usually means the project file is corrupt. Start a new Xilinx project and copy your .v files over to the new project.
- Don't forget the .UCF constraints file!
- Xilinx core 1024 pt Complex FFT module would compile in project, but would cause "Invalid SDR Directive" error during ACE file generation. No fix found.
- Xilinx does not check typos on variable names. Will create misnamed wires, registers etc.
- Always make sure you are building the correct file (labkit.v). Compiling and generating the ACE file for the wrong .v file will cause a kit error.
- The Xilinx ISE environment seems to have many transient errors. Most can be solved simply by restarting Xilinx ISE.

Appendix D: NES Input FSM

```
/*
	NINTENDO CONTROLLER INPUT FSM
	The NES controller connects to the system via 5 wires.
	2 wires are power and ground
	The others are latch, pulse and data.
	Latch and pulse are signals from the FSM to the controller.
	Data is a signal from the controller to the FSM.

	The data read process starts on the gameclock signal.
	The data protocol (exactly as used by Nintento itself) is as follows:
	1. FSM sends latch signal high for 12 us to controller (LATCH)
	2. "A" button data (high or low) is ready on data line (READ_A)
	3. FSM waits 6 us (WAIT)
	4. FSM sends pulse signal high for 6 us to controller (PULSE)
	5. "B" button data (high or low) is ready on data line (READ_B)
	6. Repeat steps 3-5 for (in order):
		select button		(READ_SELECT)
		start button		(READ_START)
		up button			(READ_UP)
		down button		(READ_DOWN)
		left button		(READ_LEFT)
		right button		(READ_RIGHT)
	For each WAIT-PULSE sequence, the return read state is stored in
returnstate.
	Output data is registered with the 27 MHz clock before passing it on.
*/


module gameinput(clock, gameclock, reset, latch, pulse, data, plyr_input);

	input clock, gameclock, reset, data;
	output latch, pulse;
	reg latch, latch1, pulse, pulse1, data1;
	output [7:0] plyr_input;

	reg left, right, up, down, A, B, select, start;
	reg left1, right1, up1, down1, A1, B1, select1, start1;
	assign plyr_input = {left, right, up, down, A, B, select, start};

	reg [3:0] state, nextstate, returnstate, nextreturnstate;
	reg [11:0] count, nextcount;

	parameter INIT = 0;
	parameter IDLE = 1;
	parameter LATCH = 2;
	parameter WAIT = 3;
	parameter PULSE = 4;
	parameter READ_A = 5;
	parameter READ_B = 6;
	parameter READ_SEL = 7;
	parameter READ_STRT = 8;
	parameter READ_UP = 9;
	parameter READ_DOWN = 10;
	parameter READ_LEFT = 11;
	parameter READ_RIGHT = 12;

	parameter TWELVE_US = 12'h144;	//count for 12 us on a 27 MHz clock
	parameter SIX_US = 12'h0A2;	//count for 6 us on a 27 MHz clock

	always @ (posedge clock)
```

```
begin
	if (!reset) begin
		state <= INIT;
		returnstate <= INIT;
		count <= 0;
	end
	else	begin
		state <= nextstate;
		returnstate <= nextreturnstate;
		count <= nextcount;
	end

	data1 <= data;
	latch <= latch1;
	pulse <= pulse1;
	left <= left1;
	right <= right1;
	up <= up1;
	down <= down1;
	A <= A1;
	B <= B1;
	select <= select1;
	start <= start1;
end

always @ (state or returnstate or count or gameclock or data1)
begin
	//defaults
	nextstate = state;
	nextreturnstate = returnstate;
	nextcount = count;
	latch1 = latch;
	pulse1 = pulse;
	left1 = left;
	right1 = right;
	up1 = up;
	down1 = down;
	A1 = A;
	B1 = B;
	select1 = select;
	start1 = start;

	case (state)
	INIT:
	begin
		nextstate = IDLE;
		nextcount = 0;
	end
	IDLE:
	begin
		nextcount = 0;
		//get input at input rate specified by game clock
		if (gameclock)			nextstate = LATCH;
	end
	LATCH:
	begin
		//latch 12 us, then go to read A
		latch1 = 1;
		if (count == TWELVE_US) begin
			nextcount = 0;
			latch1 = 0;
			nextstate = READ_A;
		end
```

```
        else   nextcount = count + 1;
end
WAIT:
begin
        //wait 6 us, then go to pulse
        if (count == SIX_US) begin
                nextcount = 0;
                nextstate = PULSE;
        end
        else   nextcount = count + 1;
end
PULSE:
begin
        //pulse 6 us, then go to returnstate and read data
        pulse1 = 1;
        if (count == SIX_US) begin
                nextcount = 0;
                pulse1 = 0;
                nextstate = returnstate;
        end
        else   nextcount = count + 1;
end
READ_A:
begin
        A1 = ~data1;
        nextreturnstate = READ_B;
        nextstate = WAIT;
end
READ_B:
begin
        B1 = ~data1;
        nextreturnstate = READ_SEL;
        nextstate = WAIT;
end
READ_SEL:
begin
        select1 = ~data1;
        nextreturnstate = READ_STRT;
        nextstate = WAIT;
end
READ_STRT:
begin
        start1 = ~data1;
        nextreturnstate = READ_UP;
        nextstate = WAIT;
end
READ_UP:
begin
        up1 = ~data1;
        nextreturnstate = READ_DOWN;
        nextstate = WAIT;
end
READ_DOWN:
begin
        down1 = ~data1;
        nextreturnstate = READ_LEFT;
        nextstate = WAIT;
end
READ_LEFT:
begin
        left1 = ~data1;
        nextreturnstate = READ_RIGHT;
        nextstate = WAIT;
```

```
                end
                READ_RIGHT:
                begin
                        right1 = ~data1;
                        nextstate = IDLE;
                end

                endcase
        end

endmodule
```