Audio-Driven Laser Tetris
Final Project Report

Group 16:
Cameron Lewis
Xin (James) Sun

6.111: Introductory Digital Systems Laboratory
May 18, 2006

Abstract

The purpose of this project is to demonstrate an advanced version of the classic arcade game Tetris. Our version boasts a much more dynamic and random game-play experience than conventional implementations. Users manipulate the falling objects as in the traditional version, but the entire game is also driven by music and projected onto a large screen using a laser raster system.

The audio-based elements of the game operate in the following manner: the drop rate of the game pieces is controlled by music frequencies and magnitudes. The audio input to the system is connected to the AC97 audio decoder on the Labkit, which digitizes the analog signals and pass them into the FPGA for processing.

The video components consist of two display elements. First is the VGA monitor displaying the Tetris game window and an audio visualization. Second is the laser display, which interfaces with the core graphics output of the Tetris game by a separate laser controller module. That module's task is to rapidly modulate the output of the laser at appropriate times as to create a low resolution, scanning raster image of the Tetris playing field.

Table of Contents

List of Figures

List of Tables

I. Background Overview

This project is a dynamic rendition of the classic arcade game Tetris. A VGA monitor displays a frequency-based visualization in addition to the game itself, and the game also sends the video feed to a laser display system for synchronous projection onto a remote surface. The Tetris game responds to real-time audio input by changing the pace of falling blocks in response to the current background music. Essentially, the entire system is divided into four main components, as depicted below in figure 1:

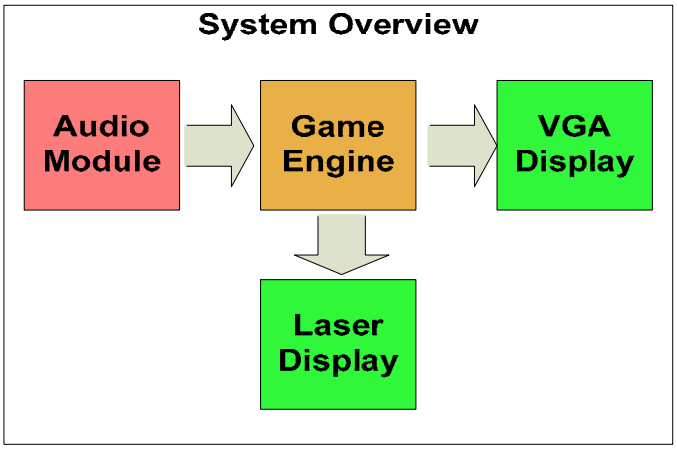

Figure 1: System Overview Block Diagram

Figure 2 illustrates the seven Tetris game pieces, which are conveniently denoted by the letters they resemble, namely I, T, O, L, J, S and Z. Note that all pieces consist of four blocks.

Please see any image of Tetris pieces,
such as http://upload.wikimedia.org/wikipedia/
commons/9/9a/Tetrominoes_letter_oriented.png

Figure 2: The seven game pieces. (Courtesy of *Wikipedia*)

The player can manipulate these pieces with left, right, rotate, and drop maneuvers. Scores are accumulated based on certain scenarios, such as settling a piece at the bottom. The falling piece is chosen at random, and the drop rate is synced to the background music.

II. Module Descriptions and Implementations

Audio Unit (recorder.v, audio_processor.v, audio_fft.v, display.v)

The purpose of the audio module is to perform a real-time FFT (Fast Fourier Transform) on the current audio data, extract intensities of specific frequency-ranges, and use that information to control the speed of the falling Tetris blocks in the main game. This was designed to create the effect of a "responsive" game environment where the current background music has a direct effect on the game play, where Tetris pieces fall faster during intense bass notes. This feature has been implemented in many innovative audio visualization packages such as those used in modern software-based media players. The block diagram of the Audio Unit is shown on the next page in figure 3:

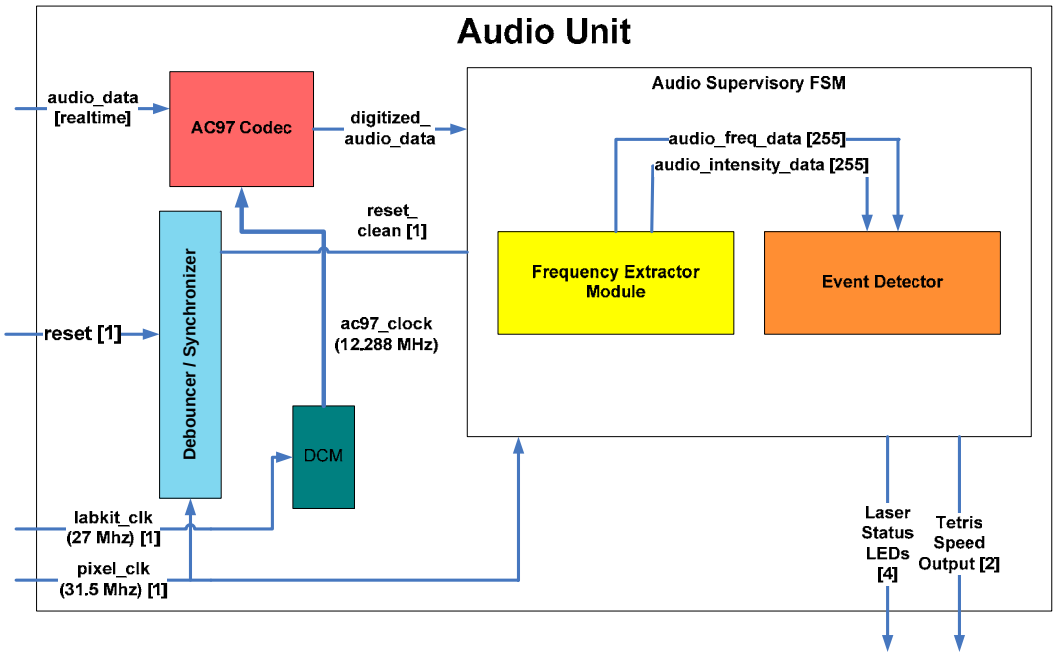

Figure 3: Audio Unit Block Diagram

In our implementation, the audio unit interfaces with the CoreGen FFT module and computes a 1024 point FFT. The magnitudes of the 16 lowest frequency ranges are displayed as bars of varying height on the VGA display, and the intensity of the frequency bucket covering the range 47-94Hz (low frequencies) is used to trigger changes in block drop speed. The way this works is relatively simple. If the magnitude of the frequency range described above exceeds a specific threshold, then the system flashes an LED on the Labkit and sends a faster game "pace" setting to the main Tetris engine.

The 16 bar spectrum analyzer shown below depicts the current FFT results in real-time. A red line on the left marks the threshold level which much be exceeded to trigger an "audio event" which affects game speed.

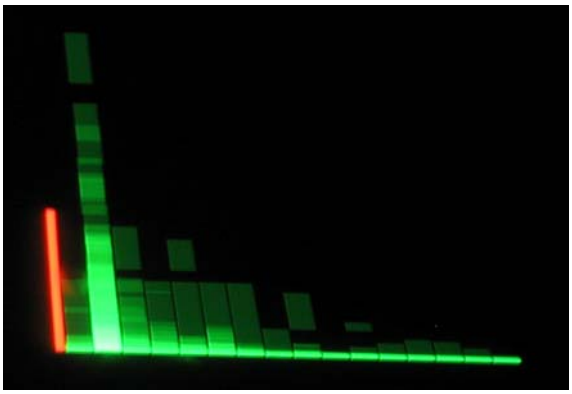

Figure 4: The audio spectrum analyzer. Frequency range is 0-752 Hz.

Verilog module integration organization is quite straightforward. The module recorder.v contains all the AC97-specific interface modules which are instantiates to create the low level audio IO connections. The module audio_processor.v deals with instantiation of the 1024 pt FFT as well as frequency extraction and spectrum analyzer data formatting/scaling. The module audio_fft.v is the FFT module created by Coregen. Finally, the module display.v is a shared resource between the VGA system and the audio system, and is used to display both the Tetris game and the spectrum analyzer on the VGA display screen.

Game Engine Unit (minorfsm.v, majorfsm.v, cdfsm.v, etc.)

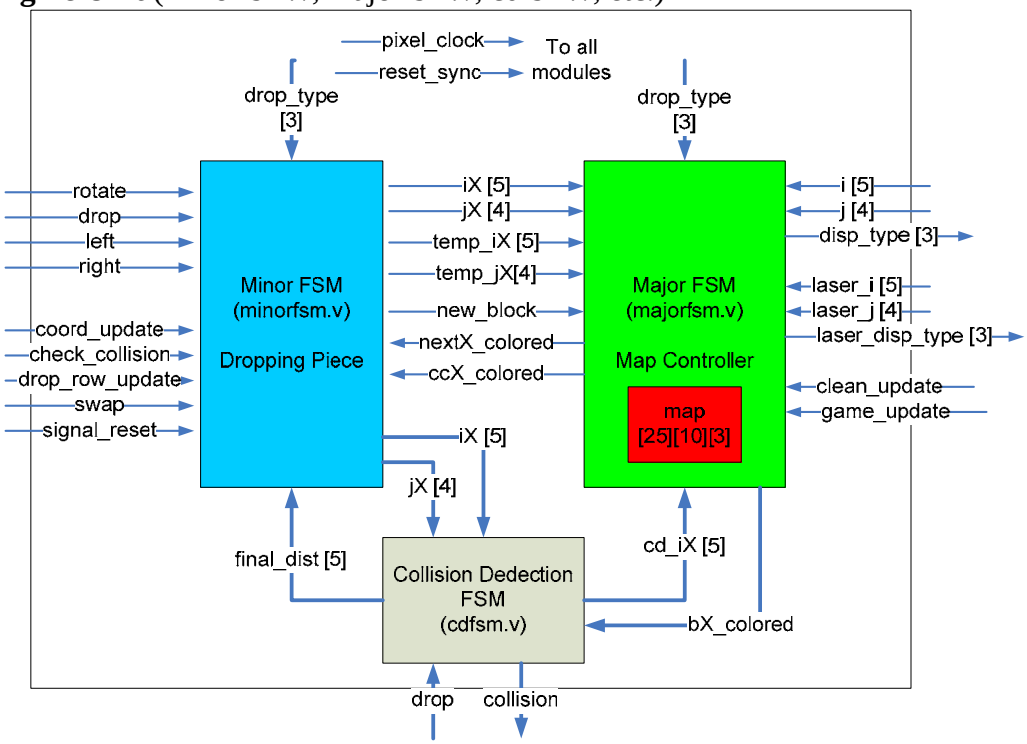

Figure 5: Game Engine Unit Block Diagram

Figure 5 above illustrates the inner workings of the Game Engine Unit. Individual modules are discussed in more detail below. Some modules that have been instantiated many times within the modules are not shown for clarity, but they are also described below. The Tetris game display is essentially a 25 by 10 array of blocks, which can be naturally represented by a matrix with the third dimension representing the block's color. Therefore, the state of the game is kept and updated in the three-dimensional array `reg [2:0]map[0:24][9:0]`, an inferred memory at runtime. To get or set the color of a block, two indices are used to pinpoint its corresponding cell within `map`. Including void (black), all of the eight colors can be represented by exactly three bits.

Minor FSM (minorfsm.v)

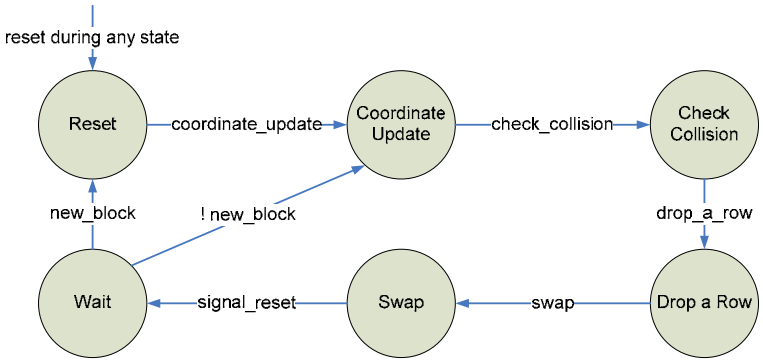

Figure 6: Minor FSM State Transition Diagram

The Minor FSM is in charge of controlling the dropping piece, whose behavior is dictated by the state transition diagram in figure 6 above. Two sets of coordinates are implemented, since a temporary

set is used to test the piece's next movement tentatively just in case the player made an illegal move. If so, the coordinates can then revert back to its previous, uncorrupted states with the other set. During the Coordinate Update state, external inputs such as move left or drop are taken into consideration to determine the next coordinates of the piece. In the next state, potential collisions (overlapping blocks) with the walls or other pieces are being checked. If no collisions occur – implying that the external signals in Coordinate Update state were legal – the coordinates are then updated. In Drop a Row, scores are updated if a new piece will be dropped, perhaps due to the previous piece hitting the bottom; else the piece by default moves down one row. During Swap, the temporary set is updated with the permanent set, thereby updating it for the next tentative move.

Major FSM (majorfsm.v)

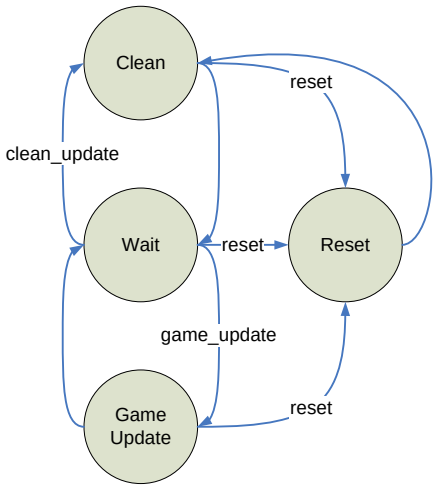

Figure 7: Major FSM State Transition Diagram

The Major FSM consists of four states. During the Reset state, all cells in `map` are reset to zero, thereby clearing the game state. During the Clean state, the current blocks of the dropping piece are made void (drawn black) in order to eliminate a tracing effect for the next move. Then execution enters the Wait state, and it waits for a high `game_update` to enter the Game Update state. In that state, `map` is updated to reflect the next move. In other words, the Clean-Wait-Game Update cycle erases the dropping piece's last game state and updates `map` with its new position as well as any collisions between it and the settled pieces at the bottom.

Randomizer (rand.v and rand.xco)

The following figure illustrates the behavior of the CoreGen Linear Feedback Shift Register, which is used to generate pseudo-random numbers.

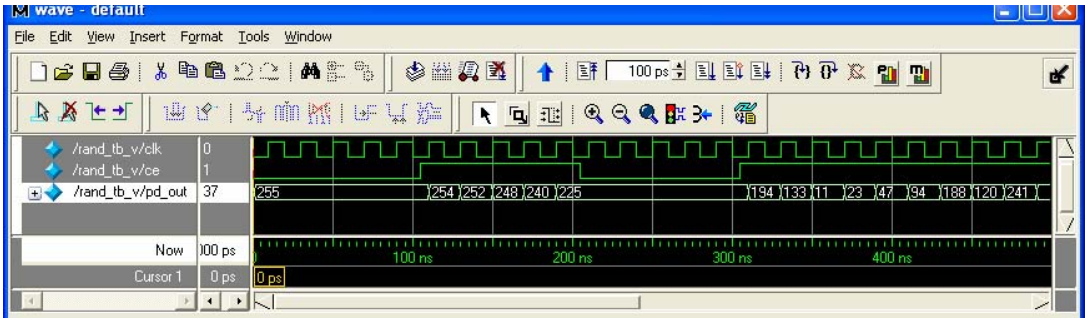

Figure 8: Randomizer Waveform

When `ce` is enabled, `pd_out` outputs an eight-bit number every clock cycle. When a new piece will be dropped, `ce` is enabled and the last three bits of `pd_out` is taken to determine the type of the new piece. If the output is `3'b000`, which is void, the new type is assigned to the red vertical piece, as it is favorite piece of most Tetris fans.

Rotational Modules (rotate_dia.v and rotate_ortho.v)

These two modules take in the coordinates of a block and the center of rotation, and outputs the new coordinates of the block rotated clockwise. The center of location for each piece is also predetermined. Whenever a piece is rotated, the blocks either does a *diagonal* rotation or an *orthogonal* rotation as illustrated in figure 9:

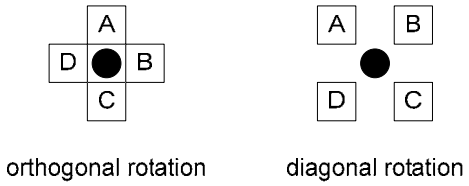

Figure 9: The black circle is the center of rotation. The letters denote the clockwise rotation sequence.

For instance, the rotation sequence of the Z piece is shown below:

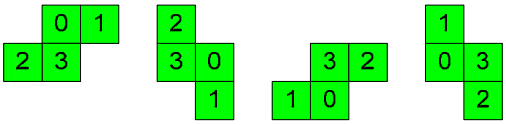

Figure 10: The numbers are block labels. For Z, block 3 is the center of rotation, blocks 0 and 2 perform orthogonal rotations, and block 1 performs diagonal rotations.

Signal Register Module (signal_reg.v)

The signal register retains a high signal (if any) during each *game move* duration. In effect, multiple high signals during a game move duration is only counted once.

Counter (counter.v)

The purpose of Counter Module is twofold. The first is to set the speed of the game. By taking in a 2-bit signal `pace`, the drop rate can be set to three different speeds with a mux: FAST (5 blocks/sec), MED (2 blocks/sec), and SLOW (1 block/sec). The second purpose is to send various signals to the minor and major FSMs to trigger a sequence of events for each game move, which are listed in the following table:

Table 1: Table of Counter Module Output Signals

Signal	**Output To:**
`clean_update`	major FSM
`coord_update`	minor FSM
`check_collision`	minor FSM
`drop_row_update`	minor FSM
`game_update`	major FSM
`swap`	minor FSM
`signal_reset`	minor FSM and `signal_reg` instances

For the functions of these signals, please refer to the corresponding modules for more details.

Self-Overlap Detection Module (self_overlap.v)

This module takes as input the four pairs of coordinates defining a game piece as well as the coordinates of the single block to be compared. If the block is part of the game piece, hence the module name "self-overlap," the module returns a high bit.

Collision Detection Module (cdfsm.v)

The Collision Detection module, `cdfsm`, calculates the distance a piece needs to drop when the drop button is pressed by simulating (for internal calculation, not shown on VGA) the block falling down block by block. The variable `temp_dist` is added to the vertical coordinates of the piece to simulate falling `temp_dist` blocks downward. The module then checks if the piece at the new location overlaps any settled pieces at the bottom. To distinguish overlapping with itself at its original position with other settled blocks, the `self-overlap` module is instantiated for each of the four blocks. Naturally, `temp_dist` increments until the piece either hits a settled piece or the bottom row. At this moment, `final_dist` is set to `temp_dist`, and the game piece is moved down by `final_dist` blocks for the next game move. Figure 11 illustrates an example:

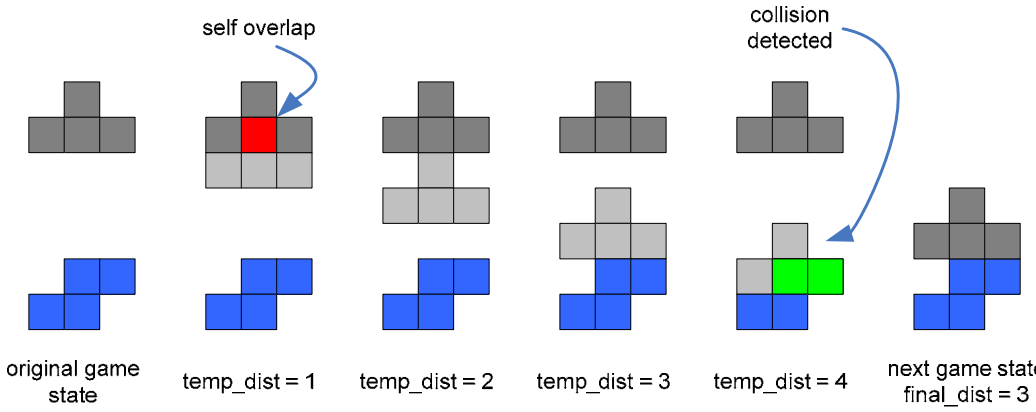

Figure 11: The simulated steps of the T piece are of a lighter grey shade. The red block indicates self-overlap, and the green blocks indicate collisions with the settled S piece.

The following waveform depicts the state transitions among states that increment `temp_dist` and check for collisions. Note the incrementing `temp_dist`.

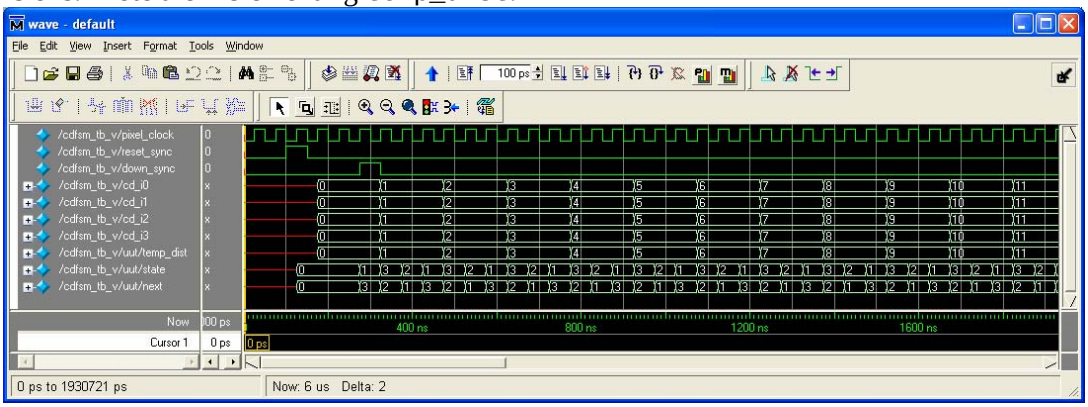

Figure 12: Collision Detection FSM Waveform

Debouncer Module (debounce.v)

This module debounces and synchronizes external buttons signals on the FPGA.

VGA Display Unit (display.v, vga.v, rect.v)

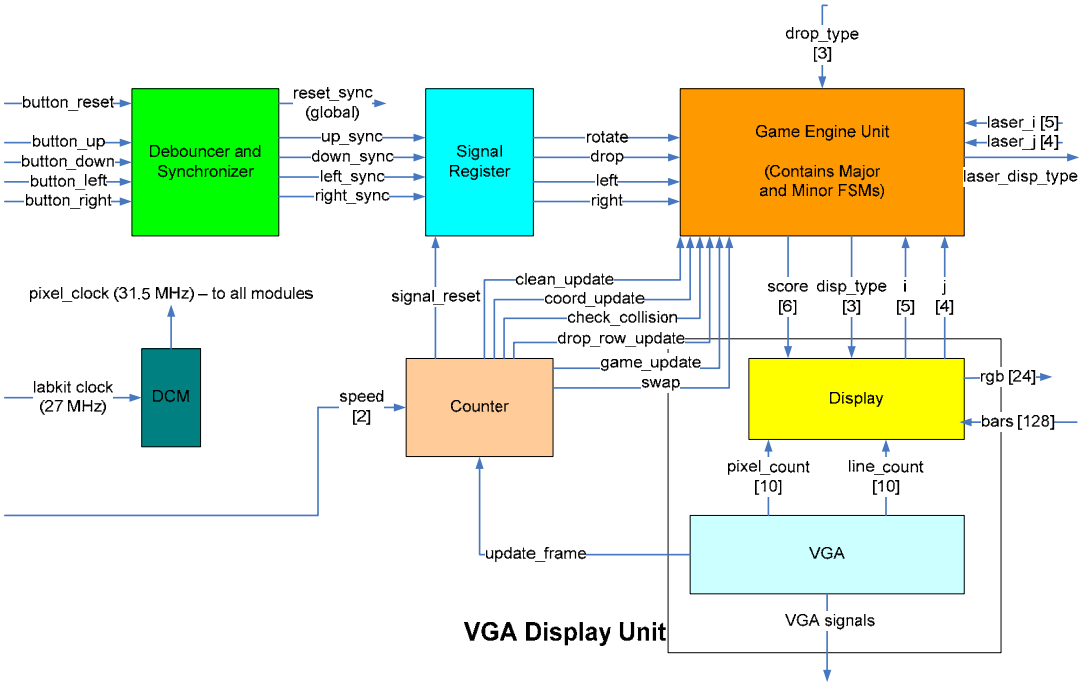

Figure 13: Block diagram containing VGA Display Unit in relations to other modules.

The VGA Display Unit consists of two large modules, Display and VGA. Figure 13 above shows both modules (lower right) in relation to other modules found in the Game Engine Unit.

Display Module (display.v)

This module instantiates all the rectangles composing the game window and spectrum analyzer on the VGA monitor. A block's color is determined by a case selection on the block's corresponding three-bit color type stored in `map`. The gaming window consists of 25 rows and 10 columns. A screenshot of the VGA is shown in figure 14, with the Tetris game on left and spectrum analyzer on the right.

:

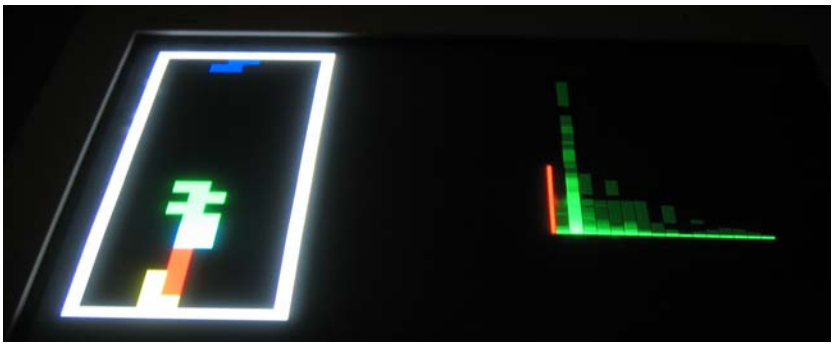

Figure 14: Screenshot of the VGA Display

VGA Module (vga.v)

The VGA module is responsible for providing various correct syncing and blanking signals, both horizontal and vertical, to the VGA display. Most importantly, this module also outputs to several other modules a high `update_frame` signal with a pulse width of one pixel clock cycle whenever the `vsync`

signal goes low. Other modules use this signal to prepare the next frame during the vertical blanking period between frames. The timing diagram is shown below:

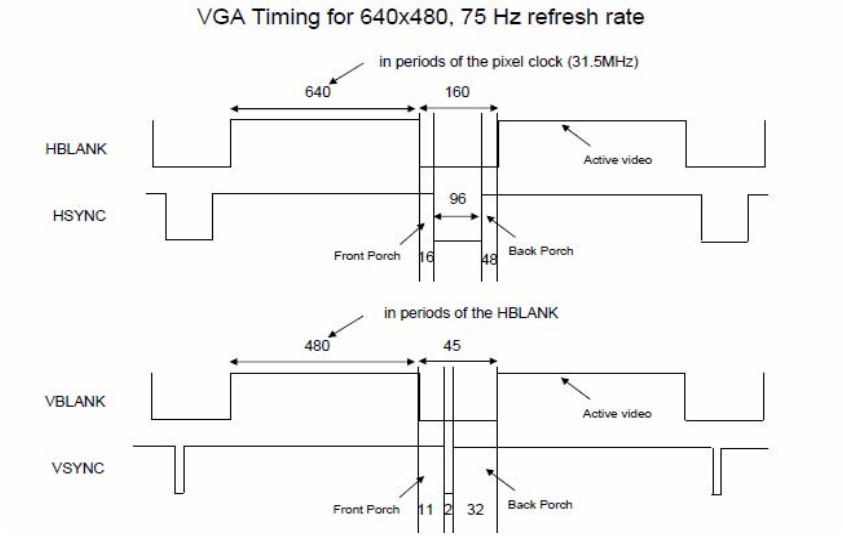

Figure 15: VGA Timing Diagram

The screen has 640X480 pixels, and the pixel clock generated by the DCM is 31.5 MHz, resulting in a 75 Hz refresh rate. The structure of the VGA Module code is more like a nested for loop: pixel_count increments from 0 to `PIXELS`-1 for each line, and line_count increments from 0 to `LINES`-1 for each frame.

Rectangle Module (rect.v)

Given the upper left-hand corner coordinates of a rectangle, the width and height, and a color, this module determines if a given pixel falls within the rectangular region. If so, the pixel is set to the input color; else, it is set to black. Because all objects displayed on the VGA are composed of rectangles, this module essentially provides the display functionality at the lowest level.

Laser Display Unit (display_enable.v, fsm.v, fancy_dots.v, scan_sampler.v)

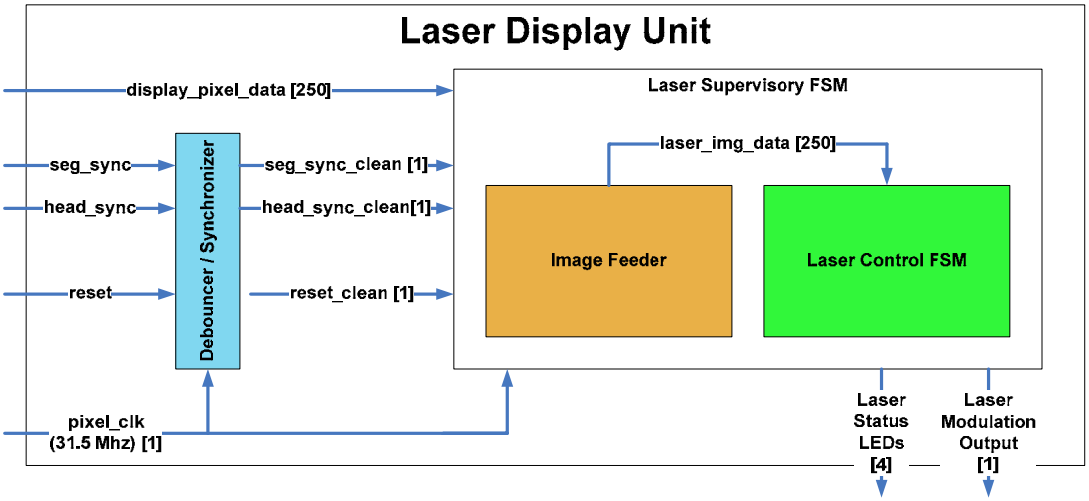

Figure 16: Laser Display Module Block Diagram

The laser display module was built to provide an alternate visualization of the Tetris game playing field. The motivation behind this endeavor was primarily driven by the technical challenge it presented, but it was also reasoned that the demonstration of a long-range laser projection system would be a worthwhile endeavor in itself. The display itself is based on a hybrid electro/mechanical mirror assembly which uses a Dremel motor to drive a 10-sided mirror head. Laser light shines onto the head which spins at a variable rate, typically between 15-20 rpm. As the head rotates, the laser is turned on and off at very specific times to create the effect of an images as the laser dot sweeps across the projection surface in two dimensions. Persistence-of-vision from the bright laser dot enables users to perceive a continues image once the mirror head is spinning fast enough.

Since this mirror head assembly is powered by a simple DC motor whose RPMs fluctuate somewhat randomly, it is necessary to synchronize the modulation of the laser with the mechanical mirror head assembly. This is accomplished through the use of two infrared emitters and detectors positioned on either side of the mirror head. Holes have been pre-drilled in specific patterns around the mirror head so as to ensure that the infrared light is transmitted through the assembly at highly specific points during the rotation of the assembly. This "break-beam" configuration outputs either a low voltage level when the infrared detector is exposed to the emitter, and remains high at other times. This opto-mechanical setup effectively functions in much the same way as the horizontal sync and vertical sync pulses of a typical VGA display. While seemingly intuitive, this component is absolutely critical to reliably displaying stable video data on the projection surface.

The entire assembly is shown below. Note the two black-tipped IR photodiodes located just to the right of the aluminum mirror head. The laser emitter is located just left of the top center. The design is such that the laser shines towards the camera and bounces off each of the 10 mirrors in turn, ultimately projecting forwards as shown in a subsequent picture.

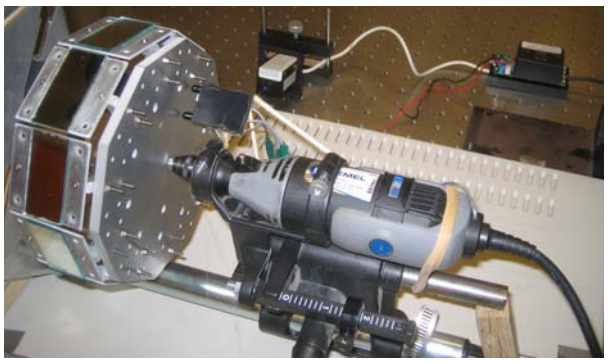

Figure 17: The laser, spinning mirror head, and IR sync assembly.

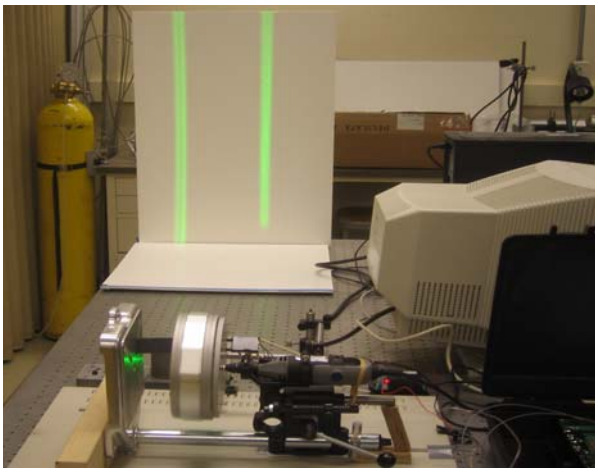

Figure 18: The laser display in operation, projecting two lines onto a remote surface.

As mentioned above, the laser is modulated at the appropriate times by the Labkit so as to create the effect of discrete pixels as the beam scans across each of the 10 mirrors, each of which is calibrated to a slightly different angle (approximately 0.5 degree increments). A resolution of 10 x 25 pixels at a frame-rate of up to 30 fps was selected in order to optimize the Tetris aspect ratio – this is the classic configuration. This also satisfies laser switching constraints (the laser can be modulated at up to 10kHz, and we push 7.5kHz at full frame rate).

Below is an image of the laser display surface. The test pattern for each column is an alternating series of on and off pixels. 10 lines should be visible (there is some overlap in this image), but this was mostly corrected for the final project demo.

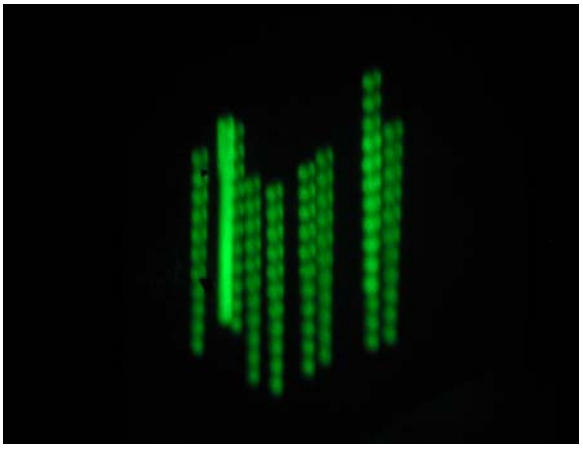

Figure 19: The projected laser raster test pattern prior to mirror alignment.

As with the audio system, Verilog integration was reasonably straightforward. The module display_enable.v is used for clock dividing purposes to give the laser modulation FSM appropriate enable pulses. The file fsm.v handles laser specific control tasks such as modulating the laser in real-time, displaying appropriate "pixels" based map data from the Tetris engine using as an index the current laser x and y positions. These coordinates are inferred from the hsync and vsync pulses mentioned above, along with extrapolated timings between sync pulses. The module fancy_dots.v allows for the display of arbitrary decimal display of binary values in the decimal range of 0-9999. It is used to show the "Frames Per Second" (in the rightmost 4 'hexdisplay' digit slots) that the mirror head is currently spinning at (averaged over 1 second). The last module, laserscan_sampler.v, is concerned with debouncing and filtering the IR synchronization pulses for use in fsm.v. This essentially helps limit the effects of IR noise and slightly improves sync capabilities overall.

III. Testing and Debugging

Game Engine and VGA Display

Visual Testing and Debugging

Testing the visual elements were quite straight-forward, as the VGA display itself provided enough feedback for error detection and improvements. Using the software design principle of spiral iterations, a static "screenshot" of the game display was first created with hard-coded positions and pieces of the blocks. Then the move left, move right, and drop-one-row-per-time-interval functionalities were implemented. Rotating a piece proved to be much harder; the simple way would be to mathematically multiply the piece's current coordinates by a rotational matrix, but that was infeasible, as I was using unsigned representation. Eventually, rotation patterns of blocks among the seven pieces were classified

into *orthogonal* and *diagonal* rotations, handled by the `rotate_ortho` and `rotate_dia` modules, respectively. Despite the brute-force approach, both modules ended up working reliably.

Non-visual Testing and Debugging

Although most of the project could be tested visually, subtle bugs which were not visually intuitive required ModelSim simulations. For instance, a timing issue was suspected in the interface between the random number generator and the game controller, since the shape of dropping piece did not match the correct color. With ModelSim analysis, however, a cycle delay turned out to be the cause. Another module that required ModelSim simulation was the collision detection module. Sometimes a piece would freeze or pierce through some settled blocks at the bottom, but with the waveform shown in figure 12, the sequence of FSM states that resulted the drop can be monitored.

Internal Game Representation

The state of the game is kept and updated in the three-dimensional array `reg [2:0]map[0:24][9:0]`, an inferred memory at runtime. During the development stage, many issues became apparent, either involving `map` or caused by `map`; most were fixed, but two were unable to be resolved in time for check-off.

Because I did not realize beforehand that Veriolog cannot port 3-D arrays between modules, I had to make significant changes to my original design during the development stage. Even though having only one implementation of `map` greatly reduced the number of wires, changing the indices while setting or getting the colors of (sometimes non-adjacent) many blocks during "map sweeping" operations inevitably caused delays. Furthermore, my first design of `map` involved heavily on variable bit selects within arrays, but errors stated that part-selects must be constant. Fortunately, I was able to fix the problem by using *indexed part selects*, a new syntax added in Verilog Standard 2000.

Enforcing Modularity

Because both of us were working in parallel, we defined a simple interface during the initial design state in order to facilitate the merging process at the end. To simulate signals from the other person's module, switches were used. For example, the speed of the game could be manually set with switches, thereby decoupling the game engine module from the audio module completely. During the merging process, another switch was used as the mux selector bit to toggle between manual and audio inputs for the speed.

Unresolved Issues

Two issues were unable to be resolved in time for check-off: occasional freezing during drop operations and eliminations of completed rows. The first one was particularly hard to debug, since a piece only froze roughly one in four times. In view of the algorithm used to calculate the dropping distance, we suspected that the blocks froze because it thought there exists another block right beneath it. Given the fast changes in indices while reading and writing blocks, glitches and delays were possible.

The second problem involved elimination of full rows. Row elimination could not be implemented due to multi-source errors. Using only one map would be extremely tricky, since that would involve generating new data from old data on the map and then rewriting those old data with the new data. Thus, an algorithm was devised to implement a second map, `map2`, that uses `map` to generate the state after full rows are eliminated, and then swapping its contents back into `map`. Theoretically, the algorithm precluded more than one source pointing to each cell, but perhaps due to dynamic allocation of cell locations (indices used to write into `map2` were represented by variables, since they could not be pre-determined), the FPGA mistook that as multi-sourcing.

Audio Processor and Laser Display

FFT Interfacing

A major challenge in developing the Audio Processor module was related to interfacing difficulties with the Coregen FFT Module. Although the data sheet for this module was itself highly detailed, it was still unclear how the data should be formatted before feeding it into the FFT module and how the output data should be processed. Moreover, the entire process was complicated by the somewhat confusing coregent configuration GUI which didn't map exactly to the module's datasheet in some instances due to minor typos in pin-out labeling.

Nonetheless, I was eventually able to get the module working and integrated with my system as it was intended. Several of the problems and solutions are presented below.

The first problem I encountered was that coregent would not instantiate the FFT. After struggling for a hours with naming schemes and computer configurations, it was pointed out that I had a filename with a space somewhere down the directory tree. Fixing the issue was as simple as replacing the space with an underscore. Determining that this was the problem, however, was very nontrivial and required me to seek advice from a number of different people before the solution was found.

Next I had difficulties with clocking the FFT. At first I had a slight typo in my clock signal which Verilog did not catch, resulting in no data output despite all my attempts at probing signals.

Once these problems were sorted out and I started receiving real-time FFT data, I experienced difficulties with input magnitude scaling and frequency ranges, and it took some tweaking to normalize the signals and extract meaningful data from the FFT. Ultimately, this required me to use 1024 buckets as opposed to the initial plan of 16. Of the 1024 buckets, I found the most meaningful data was encoded in the lower 16 buckets, due to the fact that the vast majority of audio data is represented in the sub kilohertz frequency range. Moreover, the magnitudes of these data had to be scaled appropriately in order to display correctly since the human ear response is nonlinear with a preference for the mid and low frequencies contained in typical audio data.

These issues were eventually sorted out, but the above problems encountered along the way resulted in unexpected delays in the overall project, compressing the schedule with which we had allocated time to system integration. As a result, we nearly ran out of time at the end.

Laser Scan Synchronization

In theory the laser scan synchronization was simple, and was designed in the following way: a vsync pulse is triggered once per revolution to signify the start of the scan, and hsync pulses trigger every time the laser starts scanning on a new mirror (of the ten total) so as to synchronize the display at the start of each column, since the laser scans top to bottom, left to right.

During manual slow-speed testing, these pulses and synchronizations work flawlessly. However, during full frame-rate operation (15-20fps), mechanical vibrations arose and created a situation in which the labkit periodically missed hsync pulses 1-2 times per second, resulting in a "lateral jitter" or seemingly random frame shifts from side to side. The result of this was a somewhat nauseating effect which somewhat degraded the quality of the overall laser display, but it was posited that a future robust implementation of the mechanical mirror head assembly would completely eliminate this situation. As this was not the primary focus of 6.111, the problem was reduced as much as possible then simply ignored when it was determined that no amount of digital correction could compensate for a purely mechanical problem. Despite this setback, the laser display did in fact work most of the way in which it was supposed to.

This problem was diagnosed with the aid of a digital oscilloscope, and a photo of the hsync and vsync traces is shown below. Note that the vsync line pulse occurs once every ten hsync pulses when the system is operating correctly, which is the situation depicted in this image.

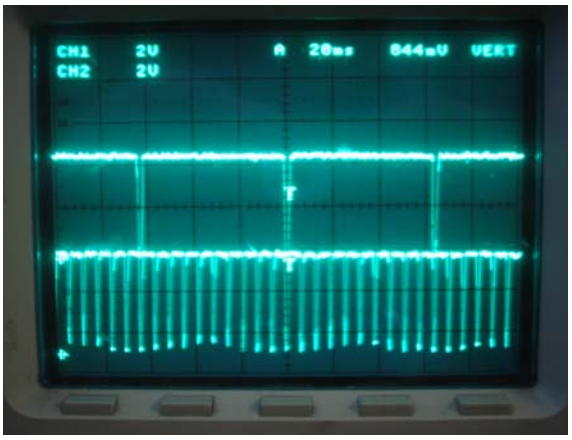

Figure 20: The hsync (bottom) and vsync (top) IR break-beam pulses on the oscilloscope.

Laser/VGA Timing Differences

Due to the fact that the VGA display updates at 75Hz and the laser display updates at 15-20Hz, one cannot use the same signal to directly draw to both the VGA monitor and the laser display. To correct for this, a complex buffering system was first designed in which the laser control module kept two local copies of the tetris map data in ram – one to update with the high speed (75Hz) map updates occurring in the Tetris game engine, and another dedicated to serving up the map data to the laser display FSM.. This idea essentially parallels the concept of common double-buffering practices, but it turned out to be unnecessarily complex for Verilog coding purposes. Halfway through the development, a most simpler solution was identified, involving a continuous assignment between the primary map representation and the (x, y) coordinates of the current laser position. It was confirmed that glitching would not occur since the combinational logic in this setup could at worst allow a pixel to be drawn one frame in the past, which at 75Hz is undetectable to any observer. Thus, recognition of a simpler solution during development forced us to chain this part of our design, but we feel it was well warranted despite the fact that it occurred relatively late in the project development phase.

IV. Conclusion

Overall we were quite satisfied with our project by the end. Almost all the important elements worked successfully at the end, with the exception of a couple small details. Although it became clear by the end of the term that nothing would be perfect, we were happy to have accomplished the bulk of a challenging project.

Specifically, all for major sections were considered a general success, though some to a greater degree than others. This included the Audio Processor module, the Tetris Game Engine, the VGA Display System, and the Laser Display System.

Having worked for 2 months almost continuously on this project, we have realized a number thing about large-scale project development. One, it is absolutely critical to design code before diving in and implementing functionality. Second, it is important to think through all the small details before plunging in. And third, it is necessary to have backup plans in the inevitable event that something fails to go according to plan. Finally, it is also absolutely critical to factor in debugging time to the development process. It is one thing to plan for design and implementation time, but it is equally important to assume that things won't work perfectly the first time and consequently to budget substantial time for such issues as debugging, system integration and code merging.

We are already much better prepared to approach difficult, large engineering problems. Although we definitely made mistakes along the way, our 6.111 project overall can be considered a success. As such, future engineering endeavors can expected to go a little more smoothly based on our experiences with 6.111.