Digital Player Piano

Stephen Chait
Joshua Runge

December 13, 2006

Abstract

This project is similar in function to a player piano, taking a written input and converting it to musical notes. More technically, the project takes an image of a piece of sheet music and plays the music via the AC97 codec on the 6.111 labkit. The sheet music is limited in complexity to the level of a simple piano score. An image of the sheet music is stored in memory for processing. Image processing techniques are used to detect features of the image, such as notes, bar lines, and accidentals. The image processing generates a series of notes, including information about their frequencies and durations. This information drives an audio processing component that plays the music via the AC97 codec on the labkit.

Contents

List of Tables

List of Figures

1. Overview

1a. General Overview

The overall project is similar in design to a player piano. Sheet music was scanned and processed to decode information about the written notes. The system is divided into several modules. These include an edge detector, note decoder, and audio processor. Stephen Chait was mainly responsible for the note decoder and audio processor. Josh Runge worked primarily on the edge detector. An overall block diagram of the project is shown below.

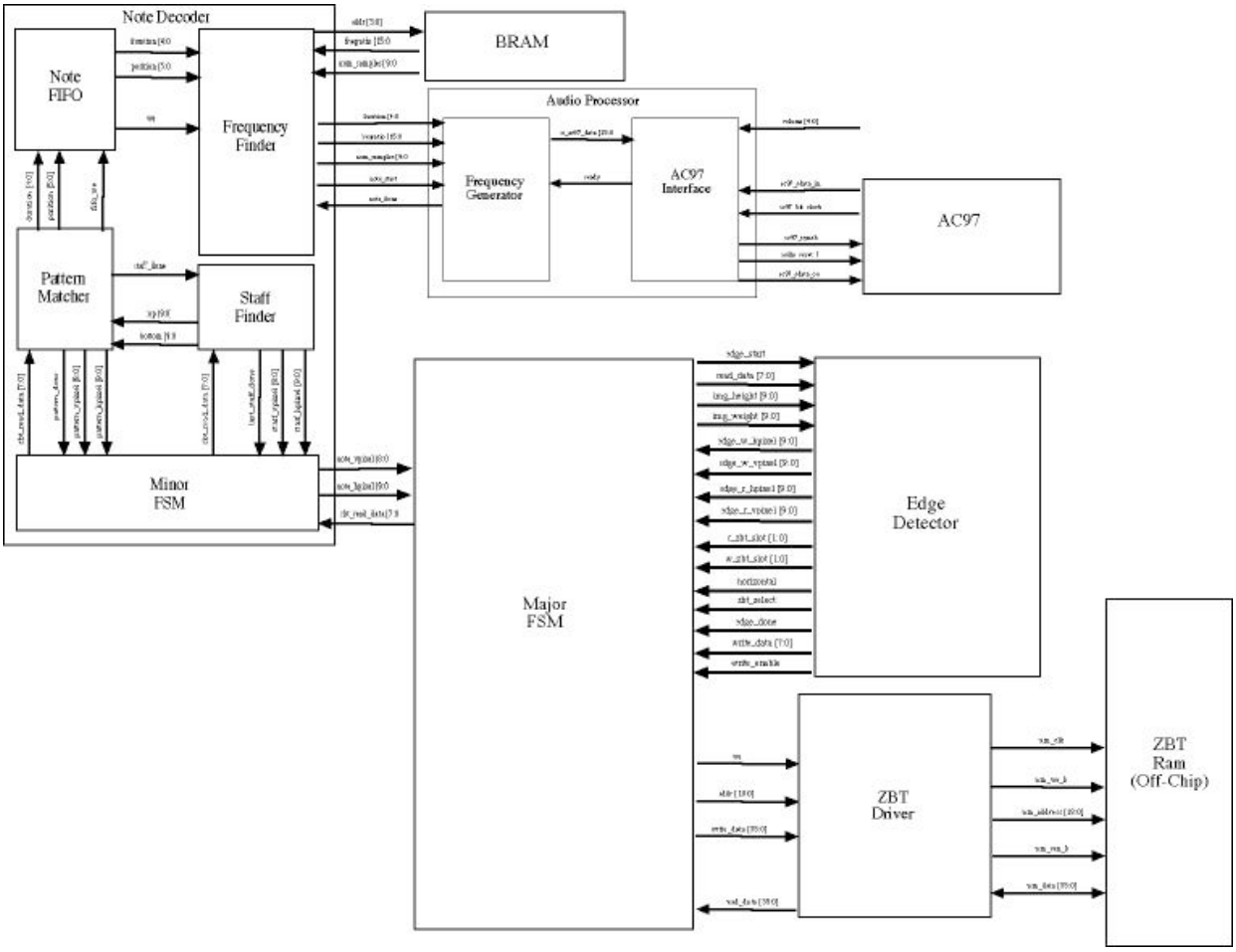

Figure 1: Top-level block diagram

In order to process an image, it is important to first extract the basic structure. Pattern matching on an unprocessed image can be easily confused by changes in lighting and camera orientation. By using edge detection on the image from the camera before passing it to the note decoder, we hope to minimize errors from such external sources.

The edge detector takes the values stored in the ZBT memory from the computer and performs edge detection on the represented image. This will be implemented through a Canny edge detector. The major steps in this process include convolution with a double

differentiated Gaussian filter and thresholding. The resulting image is be stored in the same ZBT memory.

The Note Decoder takes the modified raster image from the edge detector and determines the sequence of notes and rests, and the pitches and durations of each. It includes several smaller modules essential to its operation. The Staff Finder identifies the break between staves by looking for large sections of white space. Since we are using the constraint that all notes must be on the staff (no ledger lines), there will be a number of rows between each pair of staves that contain no edges. The module determines the first and last rows of each staff, and the locations of each line on the staff. It then sends the staves one at a time to the Pattern Matcher. The Pattern Matcher looks down each staff and identifies patterns of edges that represent notes or rests. Based on the edge pattern, it determines the duration of the note. Using the location of the lines determined by the Staff Finder, it determines the physical placement of the note on the staff (e.g. middle space, 2^{nd} line from the top). It sends the note's duration and placement to the Frequency Finder. The Frequency Finder determines the frequency of each note based on the placement of the note on the staff. Since all notes must be on the staff, the range of notes is limited (about 3 octaves if we use both treble and bass clef), so a lookup table is feasible.

The Audio Processor generates sampled sine waves at the frequencies specified by the Note Decoder. It sends samples to the AC97 codec in a loop that represents one period of the sine wave. Since the AC97 codec outputs samples at 48 kHz, it is necessary to keep the sampling rate at a constant 48 kHz. This means that the number of samples in the loop is w ,where w is the desired frequency. The Audio Processor also uses a tempo clock to determine how long to hold each note. The tempo clock is an enable signal that is generated by dividing the 27MHz clock, can be controlled by the user.

2. Edge Detector

2a. MATLAB Implementation

The first several versions of the edge detector were created in MATLAB. Initially the built in functions for edge detection were used to find a desirable threshold level. Next, functions were created to perform the independent steps of the Canny filter. Most of the focus was put into the design of the convolution module as this was anticipated to be the most complicated hardware design problem. At this point in the process, a design choice was made to eliminate non-maximal suppression. This operation would have required the calculation of arctan, which was considered too much additional complication for the hardware design. A comparison of an image with and without non-maximal suppression is shown below.

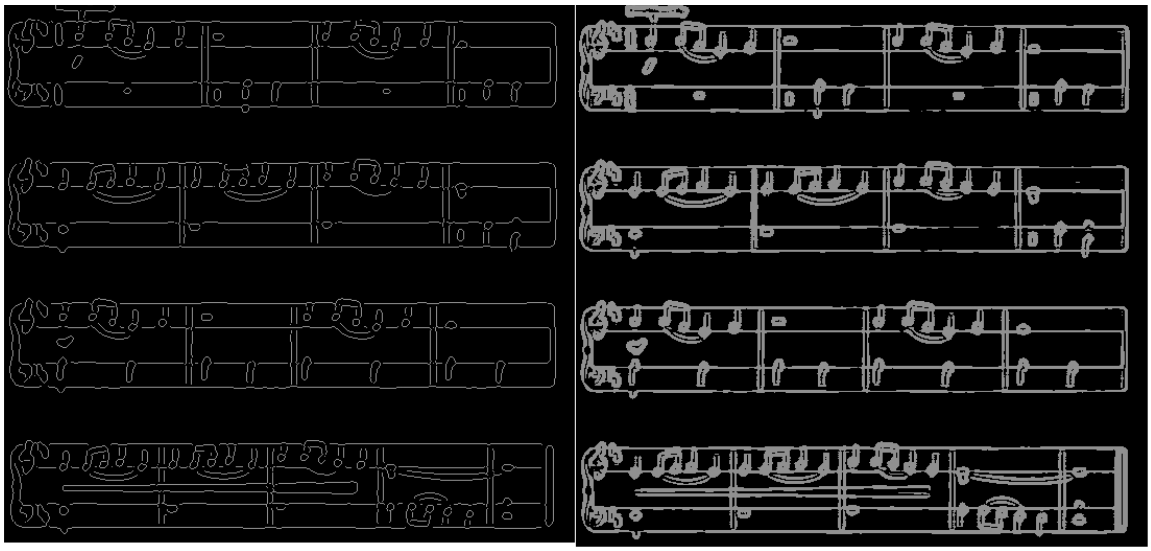

Figure 2: Results using and not using non-maximal suppression

It was convenient to have the MATLAB implementation for comparison throughout the design process because of the large amount of data involved in the project. Correct operation of the edge detector is difficult to determine without at least a 13x13 matrix of pixel values, 169 values. Computation of these values by hand would have been painful and prone to error. For each of the stages, it is possible to have the exact values that would be generated by a working edge detector.

2b. Verilog Implementation

Top-level Edge Detector

The top-level edge detector module handles communication with the major FSM for our project and through it the ZBT RAM. It outputs horizontal and vertical pixel values for reading and writing, which are translated to ZBT memory addresses. The ZBT RAM delays the read pixel values one clock cycle, which is compensated for by the introduction of old_hpixel and old_vpixel registers. The module also outputs a zbt_select value and reading and writing zbt_slots which will be described below.

An FSM is included in the top-level module to control communication with the filter and magnitude/threshold modules. This FSM progresses through five states representing operation performed and the direction of processing. The DX and GX state represent convolution in the x direction with a differentiated Gaussian and normal Gaussian respectively. The GY and DY states are similar but in the y direction. The MAG state performs an iterative square root of the sum of the squares of the above two operations. The ZBT RAM and slots used in each state as well as the transition conditions are included on the state diagram below. Both ZBT RAMs are used and the states alternate RAMs for reading and writing to ensure both operations can occur in the same clock cycle.

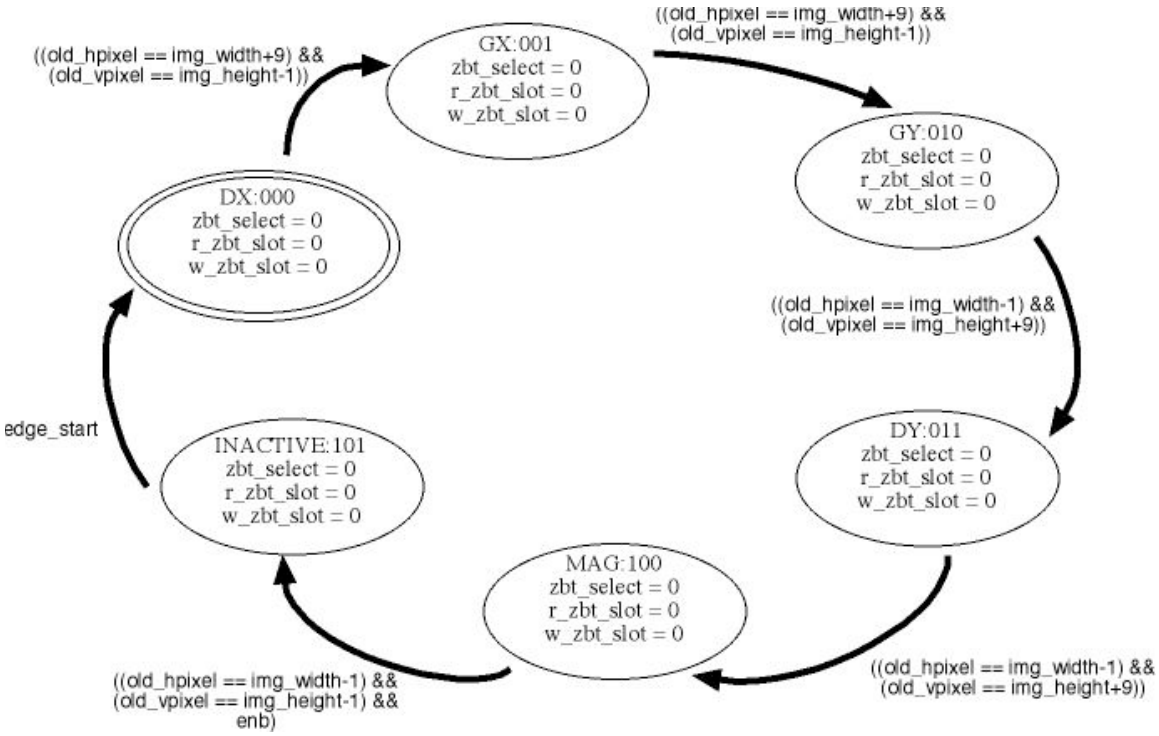

Figure 3: State diagram for edge detector FSM

There are also some important timing issues addressed by the FSM. Since the filter has a built in delay caused by the pixel width of the convolution, the hpixel and vpixel values for writing must be delayed 9 clock cycles from the read values. A simulation of the reading and writing pixel values for the GX stage is shown below. Also there are tight timing constraints on the MAG state. The two input data values must occur on successive clock cycles corresponding to the start signal and the clock period afterward. The write-enable signal must occur after the transition of done to 1 but before the enable pulse which increments the hpixel and vpixel values.

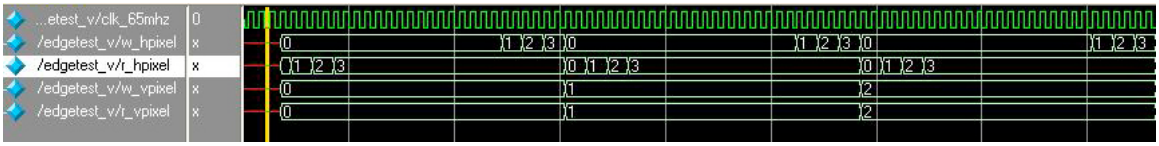

Figure 4: Simulation showing hpixel and vpixel during filter operation

Filter Module

The filter module performs the convolution the filters based on state inputs. The filter contains 13 internal 8-bit registers for input values, basically an input vector, corresponding to the 13-pixel width of the filter. On reset, the first 7 input registers are loaded with the first pixel value to extend the image and allow a more reasonable first convolution output. On successive input values, each input register is shifted right and the new value added to the first register. The input registers are multiplied, on each cycle, by the filter values corresponding to their position in the input vector. These

values are stored in 13 16 bit sum registers. These filter values depend on the diff input, as shown in the tables below. Also on each cycle, the sum registers are added together and the highest 8 bits are presented on the output. The filter produces a valid signal 9 clock cycles after the initial reset, indicating that the input vector is full of real values and the output corresponds to the sequence of input values.

Filter[i]	0	1	2	3	4	5	6	7	8	9	10	11	12
diff = 0	2	9	28	67	124	180	204	180	124	67	28	9	2
diff = 1	7	22	55	99	123	90	0	-90	-123	-99	-55	-22	-7

Table 1: Filter coefficients for Gaussian and differentiated Gaussian

The use of a differentiated Gaussian filter introduced some interesting challenges regarding signed integers. For the input image, values are stored as unsigned 8-bit values from 0-255. On convolution with the differentiated Gaussian, the values are remapped to signed 8-bit values from -127 to 127. Since these values had to be used by the filter again after the first state, signed integers had to be used in the internals of the filter. The conversion was performed based on the past state. If, the previous state performed a differentiated operation, then the input was sign extended to 8 bits before the multiplication. Since the filter never produced values above about 90, the highest order bit could be used for this operation.

Magnitude/Threshold Module

The magnitude/threshold module is mostly an iterative square root procedure. When the start signal goes high, the value on the data input is stored in the register data1. On the next clock cycle, the value on data is, if it is negative, converted to a positive number, squared, added to the square of the data1 register, and stored in data2. It is possible to determine if the number is negative because the second input is from a differentiated operation. Positive numbers will never go above about 90 so the highest order bit is indicative of a negative number. The iterative calculation of the square root is then allowed once this converted value is stored in data2. When the calculation is finished, the done output goes high. The output of this module is high if the magnitude is above the threshold of 12.

3. Major FSM

The main function of the Major FSM, built in to the top-level module, is to integrate all of the modules with each other and with the ZBT. The Major FSM has three states: LOADING_DATA, EDGE_DETECTOR, and NOTE_DECODER. The states simply indicate which module is running at the time, and therefore which address inputs and write enables should be sent to the ZBT. The FSM starts out in the LOADING_DATA state on reset. When the loading_done signal is asserted, it transitions to the EDGE_DETECTOR state and asserts the edge_start signal, telling the Edge Detector to start. When edge_done is asserted, the FSM transitions to the

NOTE_DECODER state and tells the Note Decoder to start by asserting the decoder_start signal. When the decoder_done signal is asserted, the FSM waits for the user to press Play and then asserts the playback_start signal, telling the Frequency Finder and Audio Processor to start. No change of state is needed because the Frequency Finder and Audio Processor do not access the ZBT.

The interface to the ZBT is a bit more interesting. Because of the many intermediate images used by the Edge Detector, it is necessary to use both ZBTs, as well as to use several "slots" within each ZBT. The write_zbt and read_zbt signals indicates which ZBT (0 or 1) is being written to and which is being read from. The write and read data are multiplexed to either ZBT0 or ZBT1 using write_zbt and read_zbt. Also, an offset is added to each address depending on the edge_slot_select signal.

Finally, the Edge Detector and Note Decoder work only with 8-bit pixel values. The ZBT, however, has 36 bits in each memory location, making it necessary to store four pixel values in each location of the ZBT. Thus the FSM needs to select the correct 8 bits out of the 36-bit data coming to and from the ZBT. For reading, this is relatively simple; the FSM simply selects the appropriate 8 bits based on the low-order 2 bits of the edge_hpixel and note_hpixel inputs. Since the image is stored row by row (with the four pixels in each ZBT location representing pixels from the same row), it is the horizontal index that determines which 8 bits is selected.

Writing is trickier, because the FSM needs to write 8-bit values to the ZBT without changing the other 28 bits in that particular ZBT address. Since the Edge Detector does not always move horizontally across rows, the FSM cannot simply concatenate the 8-bit values and write every four clock cycles. Before it writes, the FSM must read from the address it is about to write to, and writes can only happen every other clock cycle. The 36-bit zbt_overwrite register stores the data from the address about to be written to. Then the appropriate 8 bits of the zbt_overwrite register are replaced by the 8-bit write data from the Edge Detector.

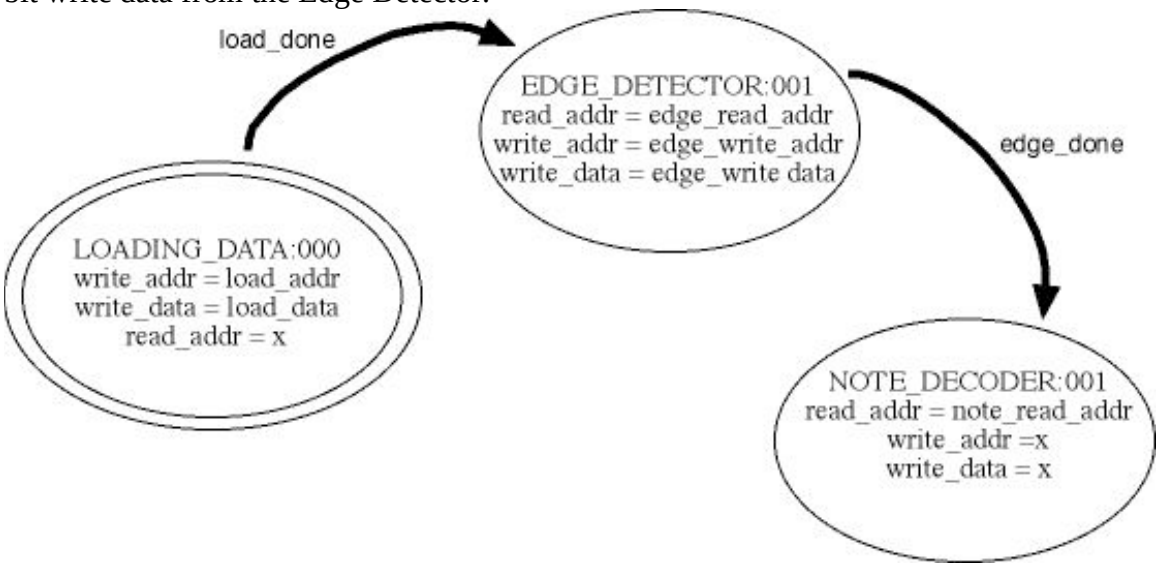

Figure 5: Major FSM state diagram

4. Note Decoder

The function of the Note Decoder is to read the edge-detected image data out of the ZBT RAM and determine the sequence of notes in each staff. The Note Decoder has three main modules: the Staff Finder, the Pattern Matcher, and the Frequency Finder. The Staff Finder finds the top and bottom lines of each staff, the Pattern Matcher determines the sequence of notes in a staff, and the Frequency Finder determines the frequency of each note based on its position on the staff. The Note Decoder operates one staff at a time. As soon as the Staff Finder finds the first staff, the Pattern Matcher decodes it. When the Pattern Matcher is done decoding, the Staff Finder starts finding the next staff, and so on. The Frequency Finder starts after both the Staff Finder and Pattern Matcher have finished.

A minor FSM interacts with these three modules and determines which signals to send to the major FSM. The minor FSM has three states: S_INACTIVE, S_STAFF and S_PATTERN. These states indicate which Note Decoder module is running at the time, and thus which pixel addresses should be sent to the major FSM. A separate state for the Frequency Finder is not necessary, since the Frequency Finder does not read from the ZBT.

The minor FSM starts out in the S_INACTIVE state. The decoder_start signal, an input from the major FSM, indicates that the Edge Detector is done and the Note Decoder can start. When this signal is asserted, the minor FSM transitions to the S_STAFF state. When the staff_done signal is asserted by the Staff Finder, the minor FSM transitions to the S_PATTERN state. Similarly, when the pattern_done signal is asserted by the Pattern Matcher, the minor FSM switches back to the S_STAFF state.

Eventually, the Staff Finder will reach the bottom of the page and assert the last_staff_done signal, indicating that all staves have been found. When this signal is asserted, the minor FSM goes back to the S_INACTIVE state and asserts the decoder_done signal. Note that there will be no notes between the bottom of the last staff and the bottom of the page, so it is not necessary to go to the S_PATTERN state when last_staff_done is asserted.

When the Note Decoder is done and the user presses the Play button, the playback_start input (from the major FSM) is asserted. This signal tells the Frequency Finder to start sending frequency data to the Audio Processor. Since the Frequency Finder does not read from the ZBT, the minor FSM does not need to change state.

4a. Staff Finder

The Staff Finder looks through the image, determines the location of the top and bottom lines of each staff, and sends those values to the Pattern Matcher. It does this by scanning each row of the image, looking for edge pixels. An edge pixel is defined as a pixel with value greater than 0 but less than 255 (since 255, the maximum value, denotes the first or last row or column on the page). If the Staff Finder finds more than 300 edge pixels in a row, it counts that row as a staff line (see figure).

The Staff Finder receives a staff_start signal from the Note Decoder, which tells the Staff Finder to start and sets its active bit to 1. The top_found bit indicates whether the top of the current staff has been found yet. This bit is initialized to 0 on reset and is

set to 1 when the first staff line is found. When the next staff line is found, the top_found bit will be 1. Thus the line represents the bottom of the staff and the top_found bit is set back to 0.

Additionally, since the boundaries of the staff have been found, the Staff Finder passes control to the Pattern Matcher by setting its active bit to 0 and asserting the staff_done signal, which is sent to the minor FSM in the Note Decoder. The Staff Finder saves the values of the top and bottom signals as well as the values for old_hpixel and old_vpixel. When the Pattern Matcher finishes a staff and the Staff Finder starts again, it starts 16 pixels below the bottom of the previous staff (since some staff lines may be more than one pixel wide).

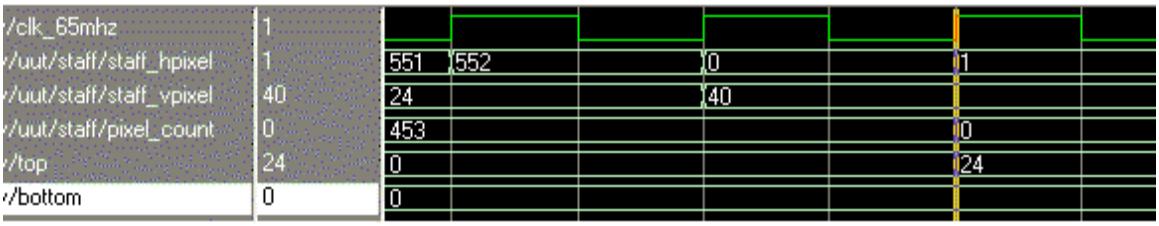

Figure 6: The Staff Finder finds the top of the staff. The row has 453 edge pixels in it.

4b. Interfacing with the ZBT

The Staff Finder and Pattern Matcher use the old_hpixel, old_vpixel, and data_valid signals to deal with the additional one cycle of latency in the ZBT RAM. When going across a row, values can be read out of the ZBT every clock cycle if one accounts for the latency. The modules do this by storing the previous pixel addresses in the old_hpixel and old_vpixel registers. Because of the latency, the data that will come out on the next clock cycle corresponds to the old_hpixel and old_vpixel addresses, not the current addresses. Thus the tests for end conditions use only old_hpixel and old_vpixel.

However, when switching to a new row (or column), the first data value that comes out after the switch may not be meaningful. The value of the last pixel in a row often helps determine which row or column the Staff Finder or Pattern Matcher goes to next (this is especially true in the Pattern Matcher). However, since the addresses must always be one cycle ahead of the data, the data for the last pixel in a row will not come out until the next address has been clocked in. This next address will not be the beginning of the next row or column the Staff Finder or Pattern Matcher wants to scan, resulting in the next data value being invalid. Therefore, whenever the Staff Finder or Pattern Matcher finishes a row, the data_valid signal is deasserted for one cycle.

We realize that the optimal solution to this problem would be to determine the most probable next address and deassert the data_valid signal only if the most probable path is not taken. However, we did not have time to make this optimization. The Staff Finder and Pattern Matcher simply go one pixel past the end of each row and column.

4c. Pattern matcher

The Pattern Matcher takes in the top and bottom lines of a staff and determines the sequence of notes and other features (stems, flags, barlines) on that staff. After edge detection without non-maximal suppression, notes and other features show up as blobs of nonzero values. The Pattern Matcher looks for rectangular blobs of mostly nonzero values. Based on the height and width of each blob, the Pattern Matcher determines if the feature is a note, stem, flag, or barline.

The operation of the Pattern Matcher is governed by a 7-state FSM. Each state represents one stage of the pattern-matching operation. The states are as follows:

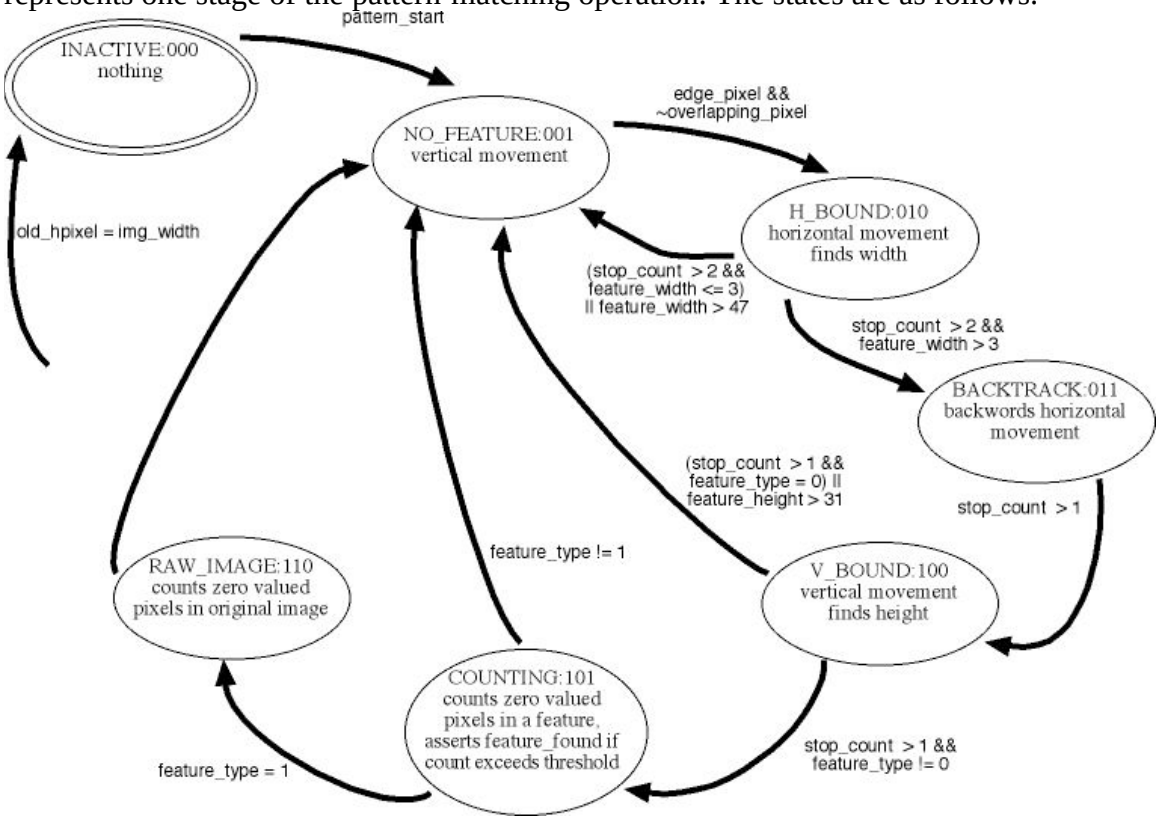

Figure 7: Pattern Matcher FSM state diagram

The INACTIVE state indicates that the Pattern Matcher is not running. The module starts in this state. When the pattern_start input is received from the Note Decoder's minor FSM, the Pattern Matcher transitions to the NO_FEATURE state.

The NO_FEATURE state represents the start of the Pattern Matcher process and indicates that the module is looking for the possible start of a feature. In this state, the Pattern Matcher proceeds vertically, column by column, so that the overall movement down the staff is horizontal. When a nonzero pixel is found that does not overlap with a previous feature, the Pattern Matcher transitions to the H_BOUND state. This nonzero pixel represents the possible start of a rectangular blob. On the transition to H_BOUND, the registers h_start and v_start are set to the pixel addresses of this start pixel. Also, the registers feature_height and feature_width are set to 0.

The H_BOUND state determines the width of the rectangular blob that starts at (h_start, v_start). The Pattern Matcher proceeds horizontally from the start pixel, incrementing feature_width for every nonzero pixel it finds. When it finds three consecutive zeroes (indicated by the value of stop_count), it stops. If feature_width is greater than 3, the Pattern Matcher transitions to the BACKTRACK state and returns to the start pixel. Otherwise, it assumes this has insufficient width to be a real feature. It transitions to the NO_FEATURE state and returns to one below the start pixel. In addition, if feature_width exceeds 47 at any point, the Pattern Matcher assumes it is on a staff line and transitions to the NO_FEATURE state.

The BACKTRACK state is like the H_BOUND state, except the Pattern Matcher proceeds horizontally backwards from the start pixel. The feature_width register is incremented for every nonzero pixel found. The procession stops when two consecutive zeroes are found, at which point the Pattern Matcher returns to the start pixel and transitions to the V_BOUND state. The BACKTRACK state is necessary because of the overlap between flags and stems. Since the start pixel cannot overlap with a previous feature, the start pixel for flags will be somewhere in the middle of the flag. The BACKTRACK state thus is necessary to determine the true width of the flag.

The V_BOUND state determines the height of the rectangular blob. In this state the Pattern Matcher proceeds vertically, incrementing feature_height for every nonzero pixel and stopping when two consecutive zeroes are found. At this point, if the feature width and height are in acceptable ranges for a note head, a stem, or a flag, the Pattern Matcher transitions to the COUNTING state. Otherwise, it transitions to the NO_FEATURE state. Also, if feature_height exceeds 31 or the pixel address goes too far past the bottom of the staff, the Pattern Matcher assumes it is a barline and the feature_found signal is asserted.

The COUNTING state counts the number of zero pixels within the boundaries of the rectangular blob determined by the start pixel, the height, and the width. If at any point the number of zero pixels exceeds the threshold, the Pattern Matcher determines that the blob is not a valid feature and returns to the NO_FEATURE state. If it reaches the end of the blob, the feature_found signal is asserted. The feature type is determined by the length and width of the feature. A note head corresponds to feature type 1.

When a feature is found, the boundaries of the feature are stored in registers to check for future overlaps. If the feature is not a note head, the Pattern Matcher transitions to the NO_FEATURE state and resumes scanning at the pixel just below the bottom-left corner of the feature. If the feature is a note head, the Pattern Matcher first transitions to the RAW_IMAGE state before going back to NO_FEATURE. The RAW_IMAGE state is just like the COUNTING state, except the Pattern Matcher is looking at the original image instead of the edge-detected image. The purpose of this is to determine whether the note is filled in or not, which is essential in determining the type of note.

The Pattern Matcher uses a FIFO to store the notes (Figure). When a note head is found, the boundaries of the note head are stored in the head_h_start, head_v_start, head_h_end, and head_v_end registers. The position of the note is determined by the relative position of head_v_start, head_v_end, and the top and bottom staff lines. The duration of the note is determined by the proximity of stems (indicating that the note is not a whole note) and flags (indicating eighth notes) as well as whether the note is filled in. The duration is stored as the number of 16^{th} notes equivalent to the note (for example,

a quarter note's duration is 4). When the next note head is found, the duration and position of the current note (the one whose boundaries are stored in the four registers) are put into the FIFO.

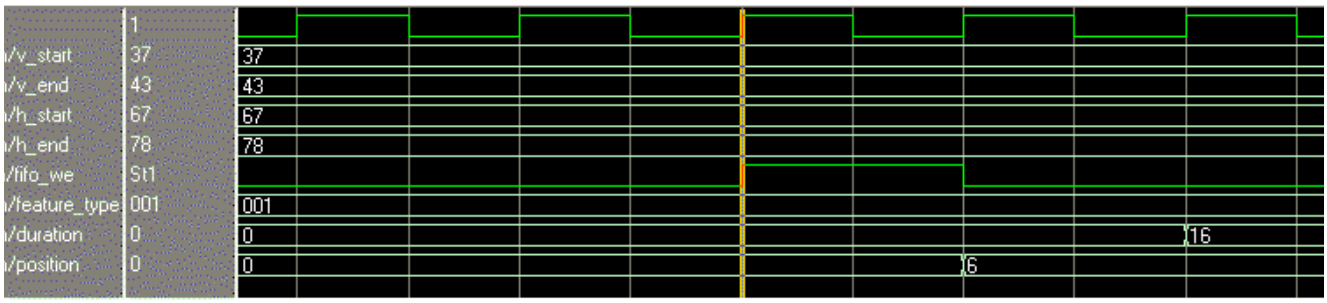

Figure 8: The Pattern Matcher finds a note. The fifo_we signal is asserted and the position is calculated from the four boundary values.

4d. Frequency Finder

The Frequency Finder reads the position data out of the FIFO and determines the note's frequency using a lookup table. The lookup table has three addresses for each named note: one for sharp, one for natural, and one for flat. This has some redundancy but takes care of the cases where the sharp of one note and flat of the next note are not the same (i.e. between E and F). The frequency finder uses the key signature input from the user to determine whether the note should be sharped or flatted.

Since the Audio Processor has a fixed output rate of 48 kHz, the lookup table does not output the actual frequency of the sine wave corresponding to the note. Instead, it outputs two values: freqratio and num_samples. Because of the sampling rate, the Audio Processor needs to generate a sine wave with frequency $w/48000$, where w is the frequency of the note. Thus the value of freqratio is $w/48000$, shifted left by 16 bits in order to represent decimal values with adequate precision. Also, the Audio Processor works by generating the samples in a loop corresponding to one period of the sine wave, with the number of samples in the loop determining the period. Thus the Audio Processor also needs the period of the sine wave, determined by $48000/w$. This is the value of num_samples.

The playback_start signal tells the Frequency Finder when to start reading values out of the FIFO. The note_done input from the Audio Processor indicates when the Audio Processor is done reading the current note. When note_done is asserted, the Frequency Finder reads a new value out of the FIFO.

5. Audio Processor

The Audio Processor generates sampled sine waves corresponding to the frequency of each note. It does this by sending samples to the AC97 codec in a loop that represents one period of the sine wave. Because of the 48 kHz sampling rate, as mentioned above, this requires calculating $\sin(2\pi n * w/48000)$ for all values of n between 0 and $48000/w$. The original intent of this project was to be able to read both treble and

bass clef. This required a range of about 40 frequencies, ranging from about 87 to over 600 Hz. Since the 87 Hz sine wave requires 550 samples, and the samples are 20-bit values, storing the sample values in a lookup table would require at least 16 BRAM's. To avoid using this much space, we used a CORDIC core to calculate the sample values.

To keep track of the samples, the Audio Processor uses the register sample_count, which increments in a loop from 0 to num_samples. Immediately after each ready pulse from the AC97, it begins calculating the next sample value. It multiplies freqratio by sample_count and then by the constant TWOPI_SH9, which is 2π shifted left by 9 bits. (We realized, too late, that the TWOPI_SH9 constant could have been incorporated into the lookup table to save a multiplication). Since freqratio has a 16-bit shift and TWOPI_SH9 has a 9-bit shift, this product is equal to 2^{25} times the desired argument of the sine function. Since the CORDIC core takes inputs in 2QN form (for a 10-bit input, this basically means the input needs to be shifted 13 bits to the left), the Audio Processor throws out the high-order 12 bits of the product. Finally, the angle is converted to an equivalent angle in the range $(-\pi, \pi)$ and fed into the CORDIC.

To control the duration of each note, the Audio Processor uses the register count_16th_note, which increments on every pulse of the user-controlled tempo clock. (The tempo clock is simply a clock divider, with the divisor controlled by switches on the labkit). When count_16th_note reaches the value of the duration, the note_done signal is asserted and count_16th_note is reset to zero.

6. Serial Data Loader

The Serial Data Loader was the one module that we could not manage to get working. It was an afterthought from the beginning, since the point of our project was the image processing and audio processing. We assumed that there would be some easy way to load our data onto the ZBT when the time came. When we couldn't find a serial/ZBT interface on the 6.111 website, we tried to build one ourselves, using a terminal emulator to load a text file onto the ZBT. But we quickly ran into a problem.

In order to transmit 8-bit grayscale values, we needed the full range of values from 0 to 255. But the terminal emulator would only accept valid ASCII characters, represented by values from 32 to 126. We considered using several characters to represent one 8-bit value. Since the range of valid ASCII characters is less than 128 values, we realized we needed at least three characters per 8-bit value. Additionally, we needed some way to represent values less than 32. The hack we came up with was to add three ASCII character values and then subtract 96, giving us a range from 0 to 282. We used a MATLAB script to generate an appropriate sequence of three ASCII characters for each grayscale value in the image.

After spending several days unsuccessfully testing this data loader, we finally realized the fundamental problem with it. We were trying to use an enable pulse to match the bitrate of the serial stream. However, because the serial data has no clock, we could never get it exactly synchronized. Bits were being shifted or skipped when the enable pulses lined up with transitions of the serial bitstream, resulting in huge chunks of invalid data.

Finally, we found a Verilog serial interface on the Internet that used a process of oversampling and recentering to solve the synchronicity problem. We built our data

loader around this module, reading in three bytes at a time and computing the grayscale value by summing the three bytes and subtracting 96. However, we did not have time to test it extensively, since we had already spent about 40 more combined hours on it than we were planning to. When we were testing the note decoder, we discovered that the output from the ZBT was different from the actual pixel values in the image. But we had no time to fix it.

7. Testing and Debugging

As was stated before, the greatest single testing procedure was checking the output of all the modules with the MATLAB implementation created in the design stages. All outputs could be check for accuracy, assuming correct inputs. Generating the correct inputs was one of the most interesting testing challenges.

Initially, inputs to the edge detector filter module were manually entered in a simple test bench. This allowed testing of the Gaussian convolution as long as more than 13 pixel values were entered. This could be compared against the convolution of a single row in MATLAB.

Before we could test the modules in simulation, we needed a way to simulate the ZBT. To do this, we wrote a simple module that introduced an additional clock cycle of latency to the memory reads and writes. We then simulated the ZBT with a BRAM, knowing that ModelSim could simulate an arbitrarily large memory. We copied the HDL functional module from one of our other memories and adjusted its parameters so that we would have a 1Mx8 memory. Since the Edge Detector and Note Decoder only worked with 8-bit values anyway, there was no point in making it 36 bits wide. We then used MATLAB to transform our image matrix into a vector of binary values, thus generating the memory initialization file for our simulated ZBT.

In later stages, the top-level edge detector module was designed and tested to ensure that it produced the correct values for the major FSM. This simulation was unable to test the actual operation of the filters or magnitude module, but at least it was able to show that the pixel values were progressing correctly. In the ideal case this module would also have been used to test the operation on the labkit with real data. Due to errors in the serial module, however, this was not possible.

Because of the sheer amount of logic involved, the Note Decoder was the hardest module to debug, other than the Serial Data Loader. Even after we got it working in simulation, it took several long days to get it working on the labkit. Because we made significant use of initial statements in our simulation, it took awhile to make sure that values were getting initialized properly. Early on, we noticed a bug in the state transitions for the Pattern Matcher. However, a thorough analysis of the logic revealed nothing. Then we checked the warnings and saw a warning about a combinational loop that involved several of the signals that affected the Pattern Matcher's state. After many hours of scanning the code and looking at schematics, we fixed the combinational loop. But the Pattern Matcher state transitions still only worked about a third of the time. Often, the Pattern Matcher would transition incorrectly to the inactive state and get stuck there.

Finally, we put a timing constraint into our UCF file. The constraint passed, but just barely. We decided to slow down the clock from 65mhz to 54mhz and then to 27mhz. At 54mhz the Pattern Matcher state transitions worked half the time; at 27mhz

the bug was fixed. It seemed that we were overclocking the Note Decoder. The logic did not have enough time to finish, leading to the invalid state transitions.

The Frequency Finder and Audio Processor were relatively easy to debug. Once we got them working in simulation, it was very easy to get them working on the labkit.

8. Conclusion

Obviously, our biggest mistake was grossly underestimating the amount of time it would take to build the Serial Data Loader. We budgeted about a half a day; it ended up taking seven days and we still didn't get it to work. We would have saved a lot of time if we had realized the synchronicity problem earlier in the process.

Also, it was very surprising how much trouble we had getting the Note Decoder to work on the labkit after we had it working in simulation. In the earlier 6.111 lab exercises, neither of us had any trouble once everything was simulating correctly. However it took several days to solve the timing issues in the Note Decoder.

Other than the data loader, however, we got our project to work. It was simply unfortunate that the Edge Detector and Note Decoder both depended on the data loader to generate valid output.