# Digital Player Piano

## Project Proposal

Stephen Chait, Josh Runge

**Abstract**

This project will be similar in function to a player piano, taking a written input and converting it to musical notes. More technically, the project will take an image of a piece of sheet music and play the music via the AC97 codec on the 6.111 labkit. The sheet music will be limited in complexity to the level of a simple piano score. An image of the sheet music will be recorded by a digital camera and stored in memory for processing. Image processing techniques will be used to detect features of the image, such as notes, bar lines, and accidentals. The image processing will generate a series of notes, which will include information about their frequencies and durations. This information will drive an audio processing component that will play the music via the AC97 codec on the labkit.

**List of Modules and Division of Labor**

In order to design and test our player piano system, it will be divided into several modules. These will include a camera interface, edge detector, rotation checker, note decoder, and audio processor. Initially, we will focus on the design of the edge detector, note decoder and audio processor. We intend to scan the sheet music and upload it from a computer to the 6.111 labkit for processing. If time allows, we will implement the camera interface and rotation checker modules, to make the system more user friendly. Stephen Chait will be mainly responsible for the note decoder, audio processor and camera interface. Josh Runge will work primarily on the edge detector and rotation checker.

**Camera Interface**

The Camera Interface module takes in the analog input from the camera, converts it to a grayscale raster image, and stores it in a ZBT memory.

This module will be based on the zbt_6111_sample module provided by the 6.111 staff, in which one ZBT memory is used as a video frame buffer, with 8 bits of grayscale data stored per pixel. Thus each address in the ZBT memory will store the information for 4 pixels.

When taking a picture of the sheet music, we will mount the camera on a tripod in order to ensure that the camera will always be the same distance from the music, and thus that the scaling of the image will always be the same. At least initially, we will restrict ourselves to a certain size and spacing on the sheet music, to make the image processing a lot less complicated.

We will use the ADV1785 chip on the labkit to convert the analog input from the camera to a digital signal, and then use Javier Castro's ntsc_decode module to decode the ADV1785 output bitstream. This module outputs 30-bit YCrCb data (luminance and

chrominance) as well as a vertical sync, horizontal sync, and a field tag. The ADV1785 does not output raster data, but instead alternates between outputting only odd pixels and only even pixels. The field tag distinguishes an odd field from an even field.

Since we will be doing multiple image processing operations on the image that will involve large blocks of pixels, we need to store the entire image in memory. The image will be on the order of 512x512 pixels, and we will use an 8-bit grayscale value to represent each pixel, so 2 million bits will be needed to store the image. This fits easily in a ZBT RAM, but would require 128 BRAMs.

We will thus use the zbt_6111 module to drive the ZBT memory. This module accommodates the two-sample delay that is necessitated by the pipelining of the ZBT memories, as well as provide the phase-shifted clock needed to drive them. In addition, the ZBT memories have an extra-long hold time, and this module makes sure that the hold time is not violated.

Finally, the ntsc2zbt module takes the NTSC output from the ntsc_decode module and manipulates it so that it can be stored in the ZBT memory. In particular, it decodes the even and odd field data, converting the NTSC data into a raster image. It also groups the pixel data into groups of 4 pixels (32 bits) for storage in one address of the ZBT memory.

**Edge Detector**

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 will take the values stored in the ZBT memory from the image capture module and perform 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 will be stored in the same ZBT memory.

The edge detector will be broken into several parts including the convolution module, threshold module and filter look-up table. The most complex part will be the 1D convolution module. This will convolve a double differentiated Gaussian filter over the image to detect edge structure. Zero crossings will indicate edges. Since the convolution in 2D is completely separable into two 1D operations, the convolution operation will likely be implemented as a series of shifts and adds along rows and columns of the image.

The filter will be approximately 15-20 values wide. This will be stored in a look-up table (LUT) since it is a small set of values. During each operation on a pixel, the convolution will have to operate on pixels 7-10 pixels to the left and right of (or above and below) the current pixel. The result of this convolution will then be passed to a threshold filter.

The threshold filter will take in a value and determine if it is close enough to zero to represent an edge. This level will be determined by trial and error and might be user adjustable if time allows.

A minor FSM will be included in this module for control purposes. The FSM will manage the look-up of values from the ZBT memory and the filter LUT. It will decide

which values to load from the ZBT memory for each convolution. This will be done through the zbt_6111 module as in the camera interface. It will also manage the communication between the convolution module and threshold module. Finally, it will write the value from the threshold module to the ZBT memory.

**Rotation Checker**

After the image structure is determined by the edge detector, the rotation checker will prepare the values for the note decoder. It is likely that during scanning, the sheet music was rotated slightly. This will pose problems for a pattern matcher that expects horizontal staff lines. The rotation checker will determine the degree of rotation of the paper and correct the edge detected image.

The plan for implementing this module is to create staff paper with a horizontal line near the top to aid in determining rotation and scaling. The line will be the width of a staff, which will give us scale for the image, assuming that the width and height of the staffs is always the same ratio.

This will also provide a simple check for rotation. The first edge pixel found when scanning from the top of the image will be the corner of one side of the line. This will determine which direction to rotate. We can then find the first blank line after this edge pixel to determine the bottom edge. This will determine how much to rotate. The rotation can then be done by shifting pixel values horizontally and vertically based on the magnitude and direction of rotation.

**Note Decoder**

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.

The Staff Finder will identify 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 will determine 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 Lookup Table.

The Lookup Table 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.

**Audio Processor**

The Audio Processor will generate sampled sine waves at the frequencies specified by the Note Decoder. It will do this by sending 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 will be necessary to keep the sampling rate at a constant 48 kHz. This means that the number of samples in the loop will be, where *w* is the desired frequency.

The function of the Audio Processor, then, will be to calculate $\sin(2\pi n * w/48000)$ for all values of n between 0 and 48000/*w*. If we use both treble and bass clef, the lowest note on the bass clef has frequency 87.31Hz, thus 550 samples would be required. The samples are 20-bit values, and there are about 40 notes in our range. Thus we could use a lookup table, but it would take about 16 BRAM's. A better solution, especially since these values need to be output at only 48000 Hz, would probably be to use the CORDIC algorithm (Coregen has one) to calculate the sample values as they are output. The lookup table would then need to store only the values of 48000/*w*, which are merely 40 10-bit numbers.

The Audio Processor will also use a tempo clock to determine how long to hold each note. The tempo clock will be an enable signal that is generated by dividing the 27MHz clock, and it will be controlled by the user.

**Testing**

Since edge detection algorithms can be implemented relatively easily in Matlab, we will scan in an image of the sheet music we are using and run the Matlab algorithms on the scanned image. We will test our edge detection and note decoder algorithms by comparing their output to the output of the Matlab algorithms. This way we will be able to tell relatively easily whether our modules are finding edges and detecting notes correctly.

# Overall Block Diagram