A Hand Controlled Digital Audio Synthesizer

6.111 Final Report

December 13, 2006

Behram Mistree

Alexander Sanchez

We built a multimedia digital synthesizer. Using subtractive synthesis, we were able to produce notes that sounded similar to a violin and a guitar. The synthesizer is controlled by a computer keyboard interface and video monitoring interface. The video monitoring interface tracks hand movements reasonably well using a video camera. The hand movements in turn control the pitch of the note that is playing. All movements will be displayed to the computer monitor as feedback to the user. We also investigated the use of a neural network in synthesizing audio. Unfortunately, we were only able to test our neural network in simulation.

Table of contents

List of Figures

List of Tables

List of Equations

Acronyms

ADSR – Attack Decay Sustain Release	
ROM – Read Only Memory	
RAM – Read-Access Memory	
FSM – Finite State Machine	
Hz – Hertz	
mHz – mega-Herz	
BRAM – Block Read-Access Memory	
LED – Light Emitting Diode	
MUX – Multiplexer	

0. Introduction

0.1 Overview of Audio Synthesis System (Alexander Sanchez)

One of the goals of this project is to synthesize audio signals that sound like musical instruments without the use of stored samples. To achieve this goal we investigated the use of to different methods for synthesizing audio.

0.1.1 Subtractive Synthesis (Alexander Sanchez)

The first method implemented was subtractive synthesis. Subtractive synthesis consists of creating instrument like sounds from a combination of basic oscillatory signals such as sine waves, square waves, or sawtooth wave. The main component of the subtractive synthesis method is to apply a volume envelope to a signal. The volume envelope used is often referred to as an Attack Decay Sustain Release (ADSR) Envelope. An ADSR envelope can be used to characterize the variation in volume of a note played on an instrument. For example, a guitar has a very sharp attack and decay, very little sustain, and a gradual release. Applying and ADSR envelope to an oscillatory signal produces a double sided enveloped oscillating signal. This signal can then be feed to the audio out channel.

0.1.2 Neural Network (Alexander Sanchez)

0.1.2.1 Neural Network Overview

The second method of audio synthesis investigated was the use of a neural network in learning how to synthesize audio. The goal of using a neural network is to have the neural network learn how to synthesize audio signals that sounded like musical instruments. In order to achieve this, the network would be trained with a sampled signal for a particular instrument. Once the network has learned to produce that signal within some acceptable margin of error, the network's learning capabilities would be removed so that the system could operate faster in order to produce cleaner sounds.

0.1.2.2 Forward Propagation

The neural network used is a multilayer feed-forward neural network. It consists of 2 inputs, 4 layers of neurons (3 layers of 2 neurons and 1 layer of 1 neuron), and 1 output. It uses forward propagation for computing the output of the system and backwards propagation for learning. Forward propagation consists of a series of computations that are internal to each neuron. First all the inputs to a single neuron are multiplied by weights (one weight is specifically assigned to each input) and then these weighted inputs are summed. This value is then passed into an activation function that produces the output of the neuron. The activation function used in this implementation of a neural network is the sigmoid function since it is differentiable and the derivative of the activation function is necessary in backwards propagation. See figure **0.1** for a mathematical diagram of a neuron. The outputs of the neurons in one level are then passed to the inputs of each of the neurons in the next level and the process is repeated until a final output is produced. See figure **0.2** for a drawing of a simple multi-layer neural network.

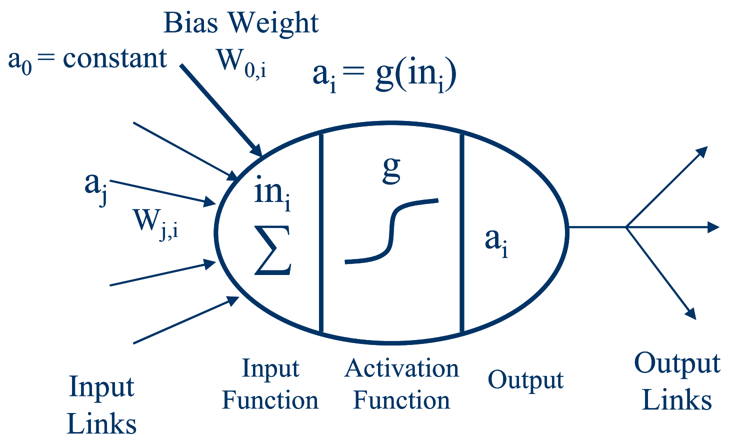


Figure 0.1[1]**:** Mathematical Model of a neuron.

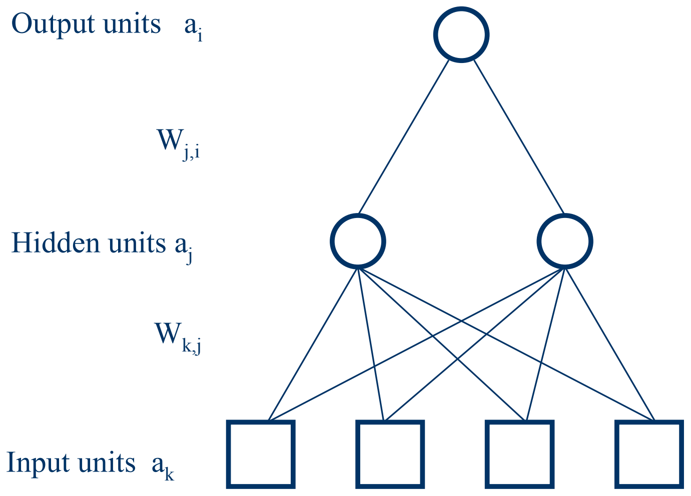


Figure 0.2[1]**:** Drawing of a simple multi-layer neural network.

0.1.2.3 Backwards Propagation

After the final output is produced, the output is compared to a target value and the squared error is computed (see equation **0.1**). This error is then modified by multiplying it by the derivative of the activation function evaluated at the current value of the output of the neuron. This value can be thought of as the delta of the outputs (see equation **0.2**). This delta is then used to adjust the weights. The amount that the each weight is adjusted by in an output neuron is equal to the output of the neuron times the delta of the output times a learning factor alpha. Alpha is a pre-determined constant factor. This adjustment factor is then added to each of the weights of the output neuron (see equation **0.3**).

Equation 0.1[1]: Squared output error vector

$$\mathbf{E} = 1/2 \sum (y_i - a_i)^2 \text{ , where } \mathbf{y} \text{ is desired output vector}$$

Equation 0.2[1]: Delta of outputs (modified error)

$$\Delta_i = \mathbf{E}_i \text{ x } g'(in_i)$$

Equation 0.3[1]: Weight update rule for output neurons

$$W_{j,i} \leftarrow W_{j,i} + \acute{\alpha} \text{ x } a_j \text{ x } \Delta_i$$

Each of the inputs to the output neuron contributed a certain amount to the output error. To account for the contribution to the error for each of the inputs to the output neuron, a separate delta is propagated to each neuron connected to the output layer. The delta propagated to each neuron connected to it is equal to the input to the output neuron received from the neuron in the previous layer times the delta of the output (see equation **0.4**). Once the output neuron has propagated all of its partial delta values the next layer of neurons adjusts its weights (see equation **0.5**) and propagates its partial deltas to the next level and so on until the inputs are reached and there are no more neurons left to propagate to. Once a neuron in a hidden layer receives all the partial deltas from the neurons it feeds its output to it calculates its delta. This delta is equal to the sum of the partial deltas it has received times the derivate of the activation function evaluated at the value of its output. It then adjusts its weights in a manner similar to the output neuron and calculates it partial delta values also in a similar way. Once backward propagation is finished the network can do forward propagation again.

Equation 0.4[1]: Propagation rule for Δ values

$$\Delta_j = g'(in_j) \sum W_{j,i} \Delta_i$$

Equation 0.5[1]: Weight update rule for neurons in hidden layers

$$W_{k,j} \leftarrow W_{k,j} + \acute{\alpha} \text{ x } a_k \text{ x } \Delta_j$$

Each time through this process the weights are adjusted a little bit in order to produce an output that is closer to the target output. The network can be trained until the output produced is within some margin of error.

0.2 Overview of Hand Detection (Behram Mistree)

The principle focus of our project is to construct an instrument that is controlled through a user's hand movements. There are a variety of ways to sense hand motion: buttons, accelerometers, and video detection would all be acceptable. We chose to do hand detection based on video signals because we felt that it would be least restrictive to the user and provide the most interesting feedback.

We implemented and tested two separate methods for detecting hand movements via video. Both incorporated a user's wearing LEDs on his/her hands. The first method looked for strings of similarly colored pixels. Whenever a string was found of suitable length, the detector would check to see if there was an LED already registered in a nearby neighborhood. If there were no LEDs within some set distance, the string of consecutive pixels became registered as a new LED.

This was a fairly naïve algorithm. It was not robust to changes in distance, and random noise and shading often impacted our ability to detect a person's hands.

Noting the success and accuracy of various past and current projects using center of mass calculations to calculate hand positions, we also implemented an algorithm which identified LEDs attached to hands by averaging horizontal and vertical values for pixels whose color values passed particular thresholds. This solution was much more robust and allowed us to experiment with other aspects of the project.

1. Module Descriptions

1.1 Module Description for Subtractive Synthesis System (Alexander Sanchez)

1.1.1 adsr_FSM

The adsr_FSM is a finite state machine that is responsible for controlling the output audio signal and the appropriate ADSR envelope being applied to the oscillating signal. It takes as input an encoding for the desired note (the signal *note*), the signal *instrument*, and the signal *play*. When the adsr_FSM receives a high *play* signal it begins the process of playing a note at the pitch that *note* encodes on the instrument specified by the signal *instrument*. The adsr_FSM contains as parameters the locations of the starting points of the attack, decay, release, and sustain portions for each instrument. A MUX that is controlled by the signal *instrument* is used to decide which address pointers are used. A *ready* signal at 496 Hz is used to sample the ROMs containing the ADSR envelopes. This allows the signal to last long enough to be audible. At every ready signal the address being accessed in the appropriate ROM is incremented by one. If the instrument should change while a note of another instrument is being played, then the address pointers will change and the adsr_FSM will change to the appropriate state to account for the change in instrument. Also, anytime that a high *play* signal is received the current note being played is stopped and a new note is started. The adsr_FSM contains 8 states, although 4 of those states are idle states. The adsr_FSM changes state once *address* equals one of the address pointers, which indicates that it is time to change the portion of the ADSR envelope being used. For example, reaching the sustain address pointer means it is time to leave the attack phase and enter the sustain phase. Once a note has reached the release pointer, the wave form generated goes to zero and audio stops coming out of the module. See figure **1.1** for a block diagam of the adsr_FSM module.

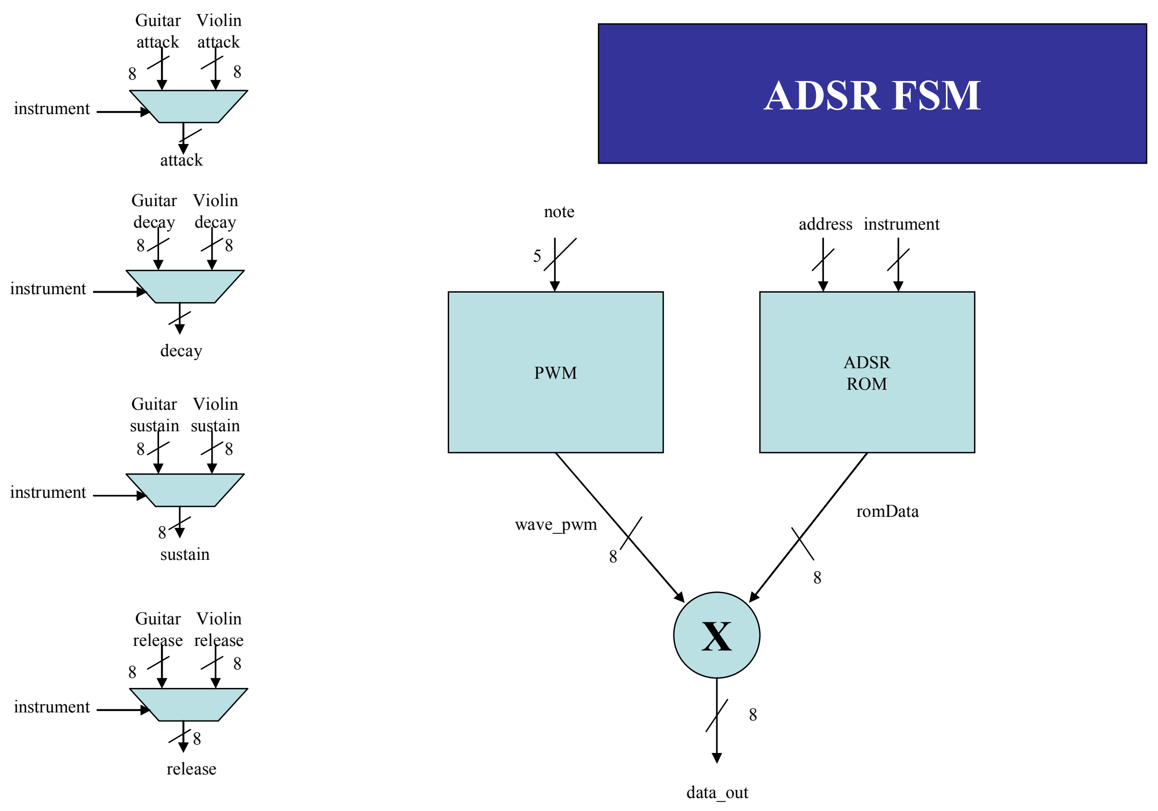


Figure 1.1: Block diagram of adsr_FSM module

1.1.2 note_to_frequency2

The note_to_frequency2 module is responsible for varying the sampling rate of the ROM holding the oscillating waves. By varying the sampling rate of the ROM it is possible to change the frequency of the wave stored in the ROM. This module takes as input the signal *note* which is the encoding used to represent the range of possible notes, see Table **1.1** for the encodings of notes that can be produced by this system. There are two octaves of notes available centered around middle C (261.626Hz).

Encoding	**Frequency (Hz)**
0	C 130.813
1	C# 138.591
2	D 146.832
3	D# 155.563
4	E 164.814
5	F 174.614
6	F# 184.997
7	G 195.998
8	G# 207.652

9	A 220
10	A# 233.082
11	B 246.942
12	C 261.626
13	C# 277.183
14	D 283.665
15	D# 311.127
16	E 349.228
17	F 369.994
18	G 391.995
19	G# 415.305
20	A 440
21	A# 466.164
22	B 493.883
23	C 523.251

Table 1.1: Encodings for Frequencies of Playable Notes

1.1.2.1 Solution to Issue of Sampling ROMs Storing Waveforms

The original note_to_frequency module was clocked with a 27mHz clock so the method used for calculating the *freq_count* signal was not consistent since everything else was run off of a 65 mHz clock. To get the module working with a 65mHz clock it became necessary to recalculate the *freq_count* values. The original value was obtained by first dropping the 27mHz clock down to 3,375,000Hz. This signal was then divided by the desired frequency to get the value of *freq_count* for the desired note. However, once we started using the 65mHz clock we realized that the signals generated where no longer at the correct frequency and were highly distorted and didn't resemble the waveforms they were suppose to. This was due to the fact that the 65mHz clock would pick up 2 to 3 positive edges of the control signals generated with the 27mHz clock causing the ROMs to be sampled at the wrong rates. To account for this error the formula for getting the value of *freq_count* was modified. First 3,375,000 is divided by the desired frequency to get a number that represents what the original signal feed into the module had to be divided by to generate the desired frequency, let's call this number f_div. Next divide 27,000,000 by 8 as was done originally and call this number clk_div. Now we again divide clk_div by f_div as before. However, now we multiply this number by 2 to account for the 65mHz clock picking up multiple positive edges, call this number div_adjusted. Finally divide 65,000,000 by div_adjusted to get the value for *freq_count*. Some problems with this method is that this is only an approximations since the 65mHz clock might not be picking up a constant amount of positive edges of the signal generated by the 27mHz clock. This caused the resulting waveforms to be slightly off pitch; however, the waveforms appear to be in relative tuning which results in decent sounds.

1.1.3 sine_generator

The sine_generator module is responsible for generating a frequency variable sine wave. It interacts with the sineROM and the note_to_frequency2 module. The sine_generator takes as input the signal *frequency* which is the encoding of the desired note pitch specified in the note_to_frequency2 module (see Table **S.1**). The sine_generator operates by sending the *frequency* signal encoding to the noteToFrequency2 module and getting the signal *max_freq_count*. *Max_freq_count* is the maximum number for the counter *enable_count* that is used to determine how long to hold the *address* signal to the sineROM at any given address. By changing the value that *enable_counter* counts to you are able to sample the sinROM at different rates and thus get sine wave of different frequencies out of the ROM.

1.1.4 sawtooth_wave_generator

The sawtooth_wave_generator is responsible for generating a frequency variable sawtooth wave. It functions almost identically to the sine_generator except that it interacts with the sawtoothROM.

1.1.5 square_wave_generator

The square_wave_generator is responsible for generating a frequency variable square wave. It functions almost identically to the sine_generator except that it interacts with the squareROM.

1.1.6 synth_channel_selector

The synth_channel_selector is used to select which audio channel is sent to the ac97 audio chip. The synth_channel_selector takes as input four 8 bit signed signals. The last signal is suppose to be a voice channel and the other 3 channels are supposed to be synthesized instruments. The input *note_selector* selects which of the three synthesized signals to play. The possible selections are the signals *note1*, *note2*, *note3*, or the sum of all three waves. The waveform selected by *note_selector* is then passed into another MUX with the voice channel. The input *voice_sel* selects which of the channels gets passed through the MUX. The possible options are the output of the MUX controlled by *note_selector*, the voice channel, or the two channels added. Finally the signal out of the MUX controlled by *voice_sel* is passed into a final MUX controlled by the signal *loop*. This last MUX was to allow for looping. The second input to the MUX was suppose to be the waveform saved into a memory. However, it is also to put a different type of a signal into the input for looping.

1.1.7 adsrROM

The adsrROM module is a wrapper for the guitarROM and the violinROM. It takes as input an 8 bit signal *address* that it then feeds into the guitarROM and violinROM *address* ports. The adsrROM also takes as input the 3-bit signal *instrument* that specifies which instrument should be synthesized, i.e. which instrument ROM needs to be accessed at the moment. The *instrument* signal was made 3 bits in order to allow future expansions with more instruments being synthesized. The adsrROM outputs an 8 bit signed signal *data* that is the data being read out of the appropriate instrument ROM. The adsrROM interacts with the adsr_FSM.

1.1.8 guitarROM

The guitarROM contains the ADSR envelop for the guitar. This envelop was generated using MATLAB. A script was used to create a graph that consisted of 4 parts: attack, decay, sustain, and release. The script can be seen in appendix **a.1**The attack portion of the guitar ADSR is a growing exponential. The decay portion is a linearly decreasing line. The sustain phase is a single point on the graph. The release phase is a slowly decaying parabola. This portions were chosen because a guitar has a quick and sharp attack followed by a quick decay that is then followed by a very minimal sustain and then finally a slow, drawn out release. A plot of the ADSR envelope for the guitar can be seen in figure **1.1**.

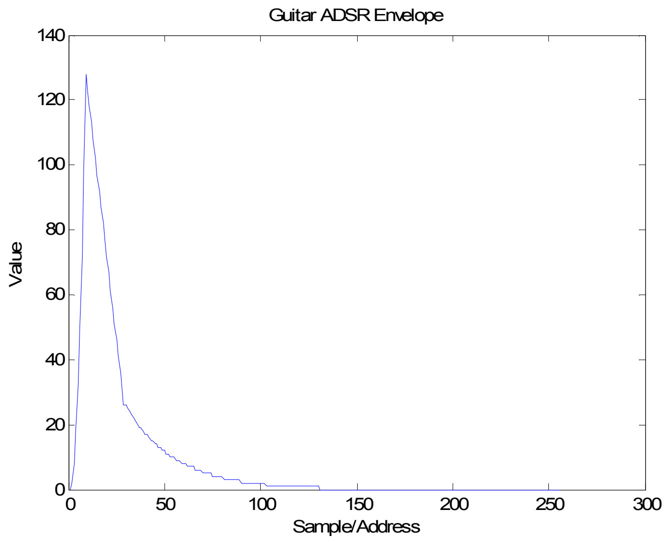


Figure 1.2: ADSR Envelope for the guitar.

1.1.9 sineROM

The sineRom module contains one period of a sine wave at 440Hz. The sineROM contains 256 samples of a sine wave that has been amplified to fit within the range of -127 to 127 so that it would not contain decimal values and fit into an 8bit signed number. It takes as input an 8 bit number for the address of the data desired and outputs an 8 bit signed number *data* that is the data located at *address*. The actual file was created using a matlab script made to compute the sampled values of the 440Hz sine wave and create the look up table.

1.1.10 squareROM

The squareROM contains one period of a square wave at 440Hz. It also contains 256 samples of the waveform and has the same structure as the sineROM. The squareROM was also created using a matlab script.

1.1.11 sawtoothROM

The sawtoothRom contains one period of a sawtooth wave at 440Hz. The sawtoothRO M also contains 256 samples of a sawtooth wave and has the same structure as the sineRom. It was also created using a matlab script.

1.1.12 violinROM

The violinROM module contains the ADSR envelop for the violin. The actual ADSR envelope was created with matlab. The ADSR envelope for the violin was generated in MATLAB using a similar method to that used for the guitarROM. However, a violin has a slow attack, no decay, a very long almost constant sustain, and then a slow decay. The ADSR for a violin looks like a wide upside down U and a plot of the envelope can be seen in figure **1.2**.

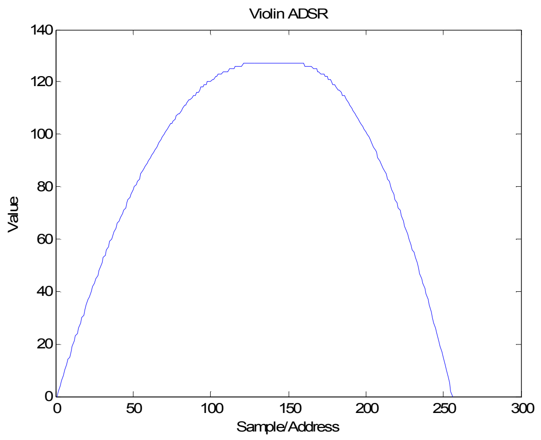
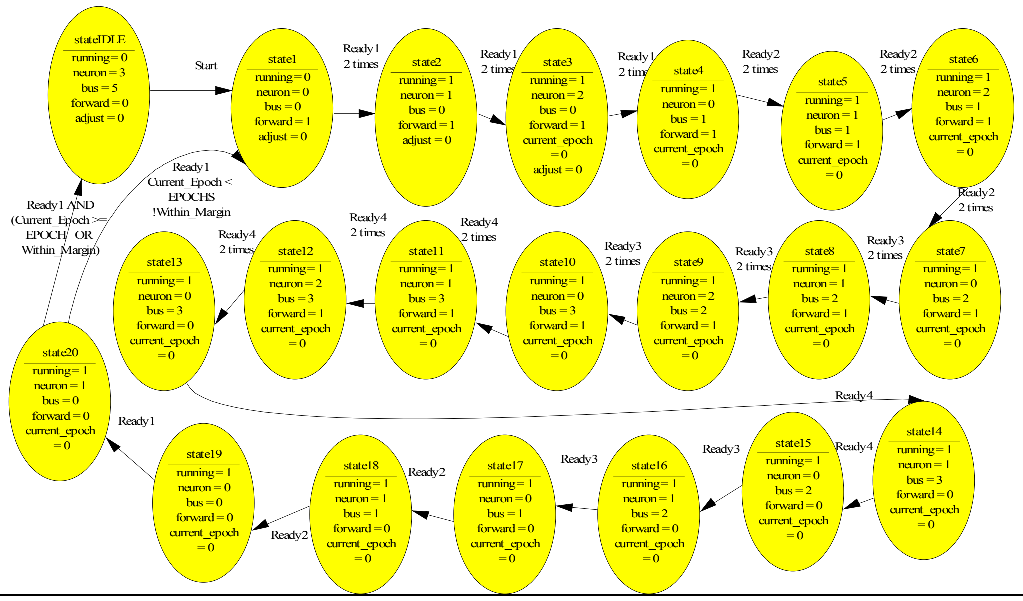


Figure 1.3: ADSR envelope for a violin

1.2 Module Descriptions for Neural Network System (Alexander Sanchez)

1.2.1 propagation_fsm

The propagation_fsm a finite state machine and is responsible for controlling the forward and backward propagation, which neuron_bus is currently active, and generally the whole flow of learning. It consists of 21 states (1 idle state, 12 states for forward propagation, and 8 states of backward propagation) and a state transition diagram can be seen in figure **1.3**. The propagation_fsm interacts with the neurons, neuron_bus, and the neuron_output modules. See Figure **1.4** for a block diagram of the neural network. The propagation_fsm receives as inputs from each of the neuron_bus (the *ready* signal from each of the neuron_buses). Receiving a *ready* signal from the neuron_bus whose ID number matches the value specified by the propagation_fsm's *current_bus* signal causes the state of the propagation_fsm to change.



stateIDLE running = 0 neuron = 3 bus = 5 forward = 0 adjust = 0
Start
state1 running = 0 neuron = 0 bus = 0 forward = 1 adjust = 0
Ready1 2 times
state2 running = 1 neuron = 1 bus = 0 forward = 1 adjust = 0
Ready1 2 times
state3 running = 1 neuron = 2 bus = 0 forward = 1 current_epoch = 0 adjust = 0
Ready1 2 times
state4 running = 1 neuron = 0 bus = 1 forward = 1 current_epoch = 0
Ready2 2 times
state5 running = 1 neuron = 1 bus = 1 forward = 1 current_epoch = 0
Ready2 2 times
state6 running = 1 neuron = 2 bus = 1 forward = 1 current_epoch = 0
Ready2 2 times
state7 running = 1 neuron = 0 bus = 2 forward = 1 current_epoch = 0
Ready3 2 times
state8 running = 1 neuron = 1 bus = 2 forward = 1 current_epoch = 0
Ready3 2 times
state9 running = 1 neuron = 2 bus = 2 forward = 1 current_epoch = 0
Ready3 2 times
state10 running = 1 neuron = 0 bus = 3 forward = 1 current_epoch = 0
Ready4 2 times
state11 running = 1 neuron = 1 bus = 3 forward = 1 current_epoch = 0
Ready4 2 times
state12 running = 1 neuron = 2 bus = 3 forward = 1 current_epoch = 0
Ready4 2 times
state13 running = 1 neuron = 0 bus = 3 forward = 0 current_epoch = 0
Ready4
state14 running = 1 neuron = 1 bus = 3 forward = 0 current_epoch = 0
Ready4
state15 running = 1 neuron = 0 bus = 2 forward = 0 current_epoch = 0
Ready3
state16 running = 1 neuron = 1 bus = 2 forward = 0 current_epoch = 0
Ready3
state17 running = 1 neuron = 0 bus = 1 forward = 0 current_epoch = 0
Ready2
state18 running = 1 neuron = 1 bus = 1 forward = 0 current_epoch = 0
Ready2
state19 running = 1 neuron = 0 bus = 0 forward = 0 current_epoch = 0
Ready1
state20 running = 1 neuron = 1 bus = 0 forward = 0 current_epoch = 0
Ready1 AND (Current_Epoch >= EPOCH OR Within_Margin)
Ready1 Current_Epoch < EPOCHS !Within_Margin

Figure 1.4: State Transition Diagram for the propagation_fsm.

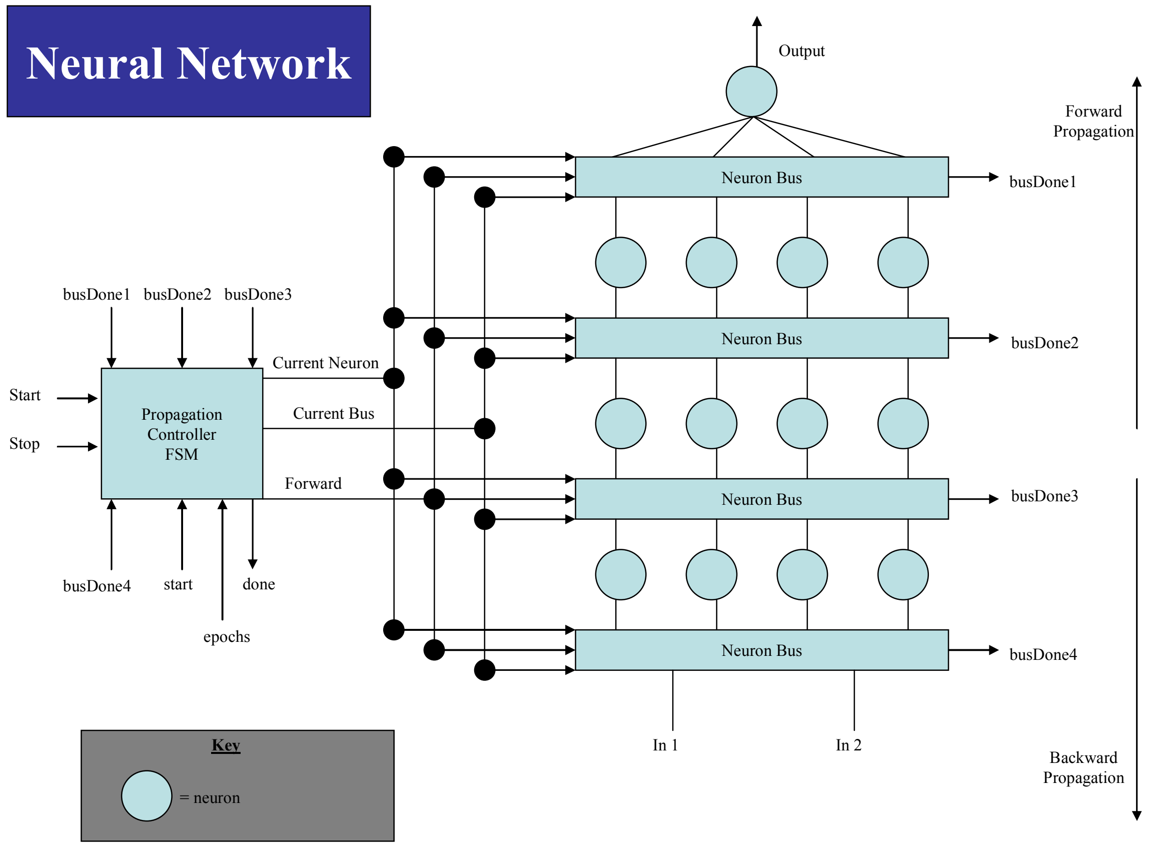


Figure 1.5: Block diagram of neural network.

The propagation_fsm generates several output signals. The *forward* and *adjust* signals control whither forward or backwards propagation is running. When *forward* is high the neural network runs the forward propagation algorithm and when the *adjust* signal is high the neural network runs the backward propagation algorithm. Only one of these signals may be on at any given moment.

The *current_neuron* signal is used to control which neuron is passing its outputs onto the outputs of the current neuron_bus at any given time as well as lets the neurons in the next level know which neuron in the previous level the current input is coming from.

The propagation_fsm also keeps track of what epoch the neural network is currently on as well as when the neural network should stop training. The stopping conditions for the neural network are to either reach the maximum number of epochs specified by its parameters or to receive a high *within_margin* signal from the neuron_output. A high *within_margin* signal means that the output error has fallen within an acceptable error margin and thus training can stop.

1.2.2 neuron_bus

The neuron_bus module is used to decrease the amounts of wires neccesary between modules. Since all each neuron in a single level are suppose to be connected to each neuron in the level before and after it, a great deal of wires would be required to make all of the required connections in a multilayer neural network. In order to decrease the number of wires between each level a bus is used. Each neuron in a single level will feed its output into the appropriate input of the neuron_bus. Then based on the *current_neuron* signal the neuron_bus will decide which neuron's outputs to allow to pass.

The neuron_bus is a two-way bus. During forward propagation it passes the output of the neurons (i.e. the output of the each neuron's activation function) and during propagation it passes each neuron's partial deltas. For backward propagation, the neuron_bus has four 32-bit signed input and two 32-bit signed outputs. Each neuron generates 2 partial deltas (one for each neuron in the next level). *Out_backward_1* goes to the neuron on the left in the next level and *out_backward_2* goes to the neuron on the right in the next level. So first the neuron_bus passes the first partial delta for each neuron through the 2 outputs of the bus and then it passes the second partial deltas for each neuron in the level. Every time that the neuron_bus has new data ready on its outputs and it is the current bus specified by the propagation_fsm, the neuron_bus will generate a high *ready* signal that gets passed to the propagation_fsm.

1.2.3 Weight Control System

1.2.3.1 weight_manager_controller

The weight_manager_controller module is a finite state machine used for controlling the loading and saving of weights from memory. It consists of 15 states (1 idle state, 7 loading states, and 7 saving states). In state 1 neuron 0 is loading its initial weights from memory, in state 2 neuron 1 loads its initial weights, and so on for each of the 7 neurons. The weight_manager_controller interacts with each of the weight_managers for each of the neurons. See Figure **1.5** for a block diagram of the weight control system. When the weight_manager_controller receives a high *start_load* signal it will begin telling neurons to load their respective weights. The weight_manager_controller sends the *current_unit* signal to each of the neurons where *current_unit* specifies the unit ID number of the current unit that is being allowed access to the memory. After a weight_manager has finished loading its respective weights from memory it sends a *load_done* signal to the weight_manager that causes weight_manager to advance to the next state.

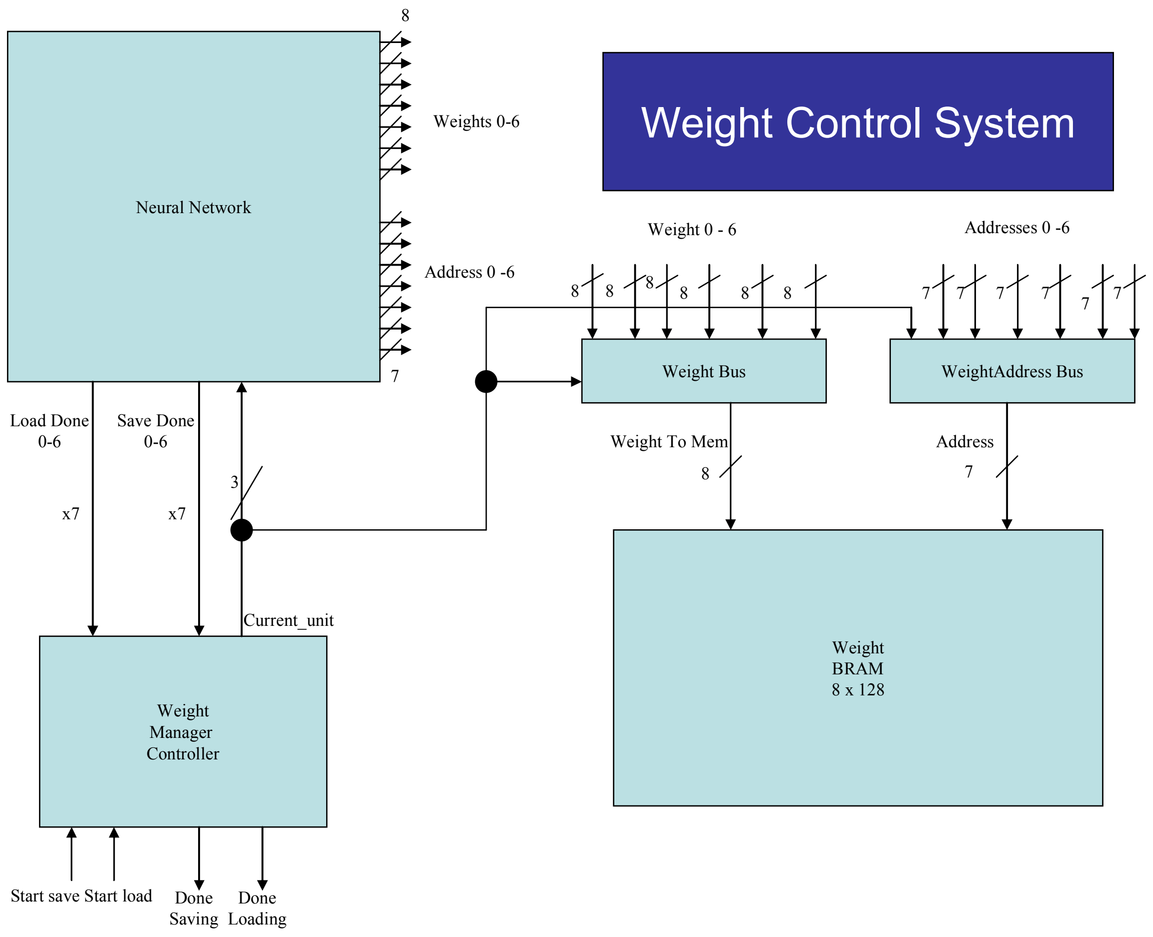


Figure 1.6: Block diagram of Weight Control System.

As for saving, state 8 corresponds to neuron 0 saving the value of its current weights to memory, state 9 corresponds to neuron 1, and so on. The weight_manger_controller enters state 8 upon receiving a high *start_saving* signal. As with loading, weight_manager_controller sends each individual weight_manager the *current_unit* signal so that each weight_manager knows when it is its turn to access the memory. When a weight_manger finishes saving its weights it sends the weight_mangaer_controller a high *saving_done* signal. The weight_manager_controller can do only one thing at a time, it is not possible for the weight_manager to save and load at the same time. Also, the weight_manager controller can't be interrupted by receiving another *start_load* or a *start_save* signal as once it receives one it will proceed to the end of the load or save process before being allowed to accept another signal.

The weight_manager_controller controls access to the memory by sending the *current_unit* signal to the weight_bus and weight_address_bus modules. This allows these two modules to know which neuron should currently be allowed access to the memory.

1.2.3.2 weight_manager

The weight_manager module is responsible for keeping track of the current values of the weights of a neuron. It has three functions. It can load weights from a BRAM, save weights to a BRAM, and load/save weights from the modules that use/adjust the weights within the neuron. Loading and saving weights to and from a BRAM is somewhat tricky since the weights are 32 bit signed numbers and the BRAM stores 7 bit signed numbers. The reason for this restriction is that the serial port sends 8-bit data and the processing we do on that data ignores the most significant bit, thus reducing the available data sent via serial to a computer to 7-bits. This means that the weights must be stored in among 5 address. Bits 31-28 of a weight are stored in bits 3-0 of the first address. Bits 27-21 of the weight are stored in the second address, bits 20-14 are stored in the third address, bits 12-7 are stored in the fourth address, and bits 7-0 are stored in the fifth address.

To load the weights from a memory, weight_manager uses a counter that acts as a pseudo finite state machine to keep track of what bits of the weight it is loading and what weight it is currently on. Once all of the bits have been loaded for each of the weights, weight_manager sends a high *load_doneI* signal to the weight_manager_controller module.

Saving is done in a similar fashion. A counter is used as a pseudo finite state machine to keep track of which bits of which weight and to what address in memory the module is sending data to. After all the data is written to the bram the module will send a high *save_done* signal to the weight_manager_controller module.

The weight_manager also interacts with the other modules within the neuron and neuron_output modules. Modules can request the current values of weights from the weight_manager and the weight_manager will send the modules the requested data. The module has two ports for interacting with other modules within the neuron. One port is an output port called *current_weight* that sends the currently requested weight to a module. Whenever the weight_manager is not loading or saving and the value of *weight_request* changes then weight_manager changes the value of *current_weight* to the appropriate weight. The module that sends the *weight_request* signal to weight_manager is weight_multiplier and weight_multiplier_output. However, the weight_adjuster modules also take the same weights as inputs as the weight_multiplier modules. The other port for interacting with modules within the neuron that weight_manager has is an input port called *current_weight_out_adj*. This port is feed by the weight_adjuster modules. When the weight_adjuster modules are finished adjusting weights they then send the weight_manager a high *adjusted_weight_ready* signal, an *adjusted_weight* signal and a *current_adj_weight* signal. These signals tell the weight manager which weight is being adjusted and what its new value is. The weight_manager then overwrites the old value of the weight being adjusted with its new value.

1.2.3.3 weight_address_bus

The weight_address_bus module is used to control which neuron has access to the address port on the BRAM containing the weights. It takes as input the *current_unit* signal from the propagation_fsm. The *current_unit* signal tells the weight_address_bus which module should be accessing the weights BRAM at any given moment. Whenever the *current_unit* signal changes, on the next positive edge of the clock the address on the *address* output port of the module will change to the address being requested by the neuron specified by the *current_unit* signal.

1.2.3.4 weight_bus

The weight_bus module controls which neuron has access to the data_in port at any given time. It works in a similar fashion to the weight_address_bus. It should be noted that the weight_bus takes as inputs 7 bit signed numbers whereas the weights used are 32 bit signed numbers. This just means that each neuron has to send each weight in five 7 bit parts to the weights BRAM. The weight_bus also takes as input the *current_unit* signal from the propagation_fsm and whenever this signal changes on the next positive edge of the clock the *weight_to_mem* output will change to the current 7-bit portion of a weight that the current neuron specified by *current_unit* is currently changing.

1.2.3.5 weight_system_test_fixture

The weight_system_test_fixture is a test fixture used for testing the interfacing of all of the components of the weight control system (weight_managers, the weight_manager_controller, the weight_address_bus, the weight_bus, the initial weights bram, and the labkit to pc via serial modules) The test was to load some initial weights from the memory, increase them by one, save the adjusted weights to memory, and then transmit the newly saved weights to a laptop via the labkit's serial port.

1.2.4 Neurons

1.2.4.1 neuron

The neuron module is the basic computational component of the neural network system. Its main purpose is to abstract away the various computations involved in forward and backward propagation. The neuron module functions in two separate modes, forward and backward propagation. During forward propagation the neuron takes as inputs the outputs of all the neurons in the previous layers and performs various calculations with those values in order to get out a single 32-bit signed value that it then propagates to all the values in the next layer. During backward propagation the neuron takes as input the delta from the layer it sent its outputs to during forward propagation. Then the neuron will perform various calculations with those deltas. Next the neuron will adjust the values of its weights and calculate its new delta. Finally the neuron propagates its new delta to the next level. See figure **1.6** for a block diagram of a neuron.

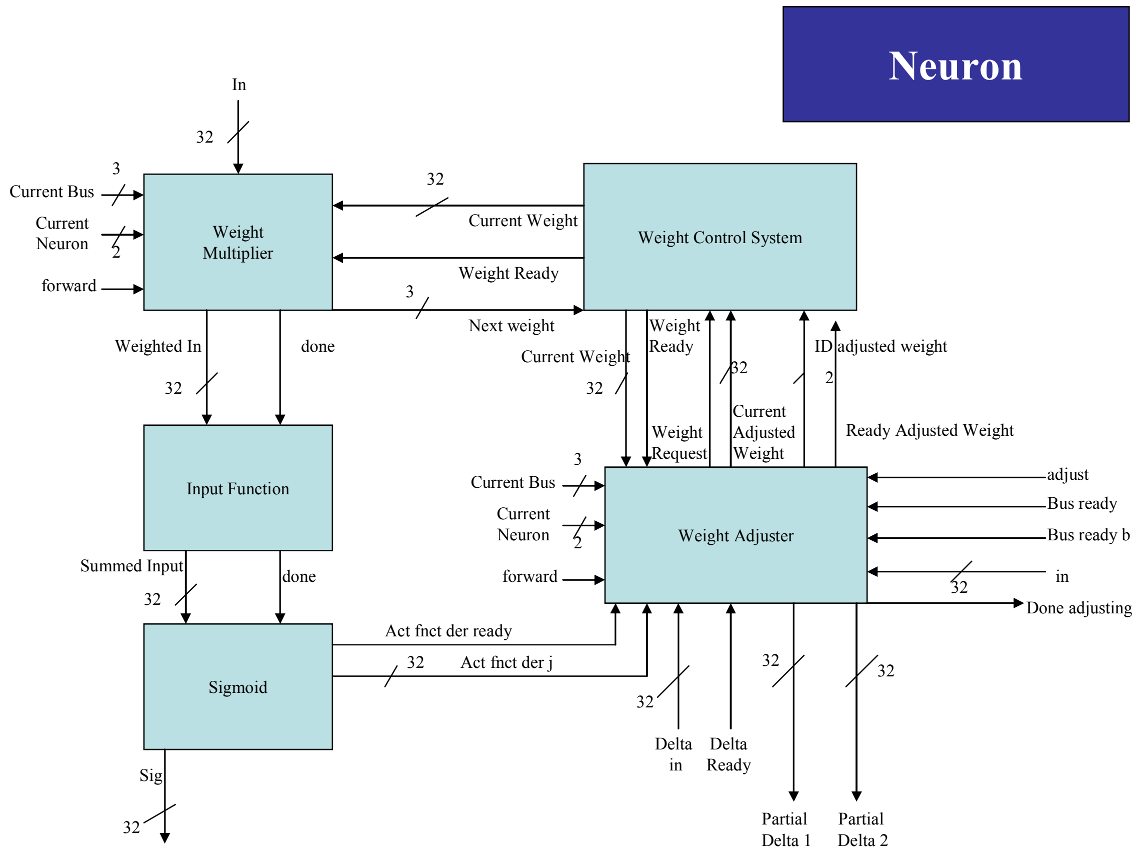


Figure 1.7: Block diagram of a neuron.

1.2.4.2 weight_multiplier

The Weight Multiplier module applies weights to the inputs of a neuron. The weight multiplier module takes as input either the inputs to the system or the output of each neuron in the previous layer. There is a specific weight in each neuron that corresponds to each input that it receives. Each neuron also has a bias weight that is multiplied by a constant input of 1. In order to assure that each weight is properly loaded and that each input value that is received is correct, the module waits until all the weights have been loaded and the inputs have been received before performing any multiplications.

First the weight multiplier module communicates with the neuron_bus module that drives the inputs of the weight multiplier. The neuron_bus tells the weight_multiplier when it has a value ready at its outputs. When the weight_multiplier receives this signal it loads the value of the output of the neuron_bus into a register and then waits for the next input from the neuron_bus. Once all the inputs have been receive, the weight_multiplier then communicates with the weight_manager module to load the current value of the weights into registers. First the weight_multiplier tells the weight_manager module the ID number of the weight it wants (0 = bias weight, 1 = weight 1, 2 = weight 2). The weight_multiplier then finds the value for the appropriate weight and sends the weight_multiplier module a high control signal once it has found the appropriate weight and has put the weight on its output to the weight_multiplier. The weight_multiplier then stores this weight into the appropriate register and tells the weight_manager the ID of the next weight it wants. Once all the weights have been loaded the weight_multiplier multiplies the inputs by the appropriate weights and stores the values in registers. It then puts these values on its output to the input_function module one at a time and signals to the input_function when a value is ready at its outputs

1.2.4.3 input_function

The input_function module is used to sum the weighted inputs to a neuron. This module interacts with the weights_multiplier module and the sigmoid module. It takes as inputs the *weighted_in* output from the weights_multiplier. The input_function module waits until it receives a high *done* signal from the weights_multiplier to store the current *weighted_in* value into a register. Once input_function has received 3 *weighted_in*s it will signal that is done summing the weighted inputs by making *done* high and placing the sum of the weighted inputs on its *summed_input* output.

1.2.4.4 sigmoid

The sigmoid module serves as the neural network's activation function. The sigmoid function is a differentiable threshold function. It is a constant high value for numbers greater than some threshold, a constant low value for numbers below some threshold, and takes on different values for numbers between the two thresholds. The graph of the sigmoid function looks like an elongated 'S'. The sigmoid module takes as input the *summed_input* output of the input_function module. The sigmoid function is then evaluated when input_function sends sigmoid a high *done* signal. The sigmoid function is then evaluated at *summed_input* as well as the derivative of the sigmoid function at *summed_input*. Since the sigmoid function involves an exponential division by a number that is possibly not a power of 2 a look up table was used to compute the function. By using a lookup table an output value is reached in a reasonable amount of time. Once the function is evaluated the sigmoid function sends a high *done* signal and outputs the value of the evaluated function as *sig* and its derivative as *sigDer*. These outputs are used to let the system know when a layer of neurons are finished with forward propagations. *sig* is passed as an input to neuron_bus which it then passes to the inputs of the next layer of neurons.

1.2.4.5 weight_adjuster

The weight_adjuster has more or less the same functionality as the weight_adjuster_output module. The big difference is that the weight_adjuster_output module gets its delta inputs from the output_errror module whereas the weight_adjuster gets its delta inputs from the neuron_bus. This means that weight_adjuster has to wait to receive both partial deltas from the propagating level. The neuon_bus has two outputs for backwards propagation, one for the partial delta propagating to each of the neurons in the level. Since each neuron is receiving 2 partial deltas it needs to wait for both. When weight_adjuster receives a *bus_ready* high signal from neuron_bus while *adjust* is high, it will store the incoming delta value as in a register the first time and then add the incoming delta value to the first value the second time to get the total delta. After the second delta is received the internal *delta_prop_recieved* signal is set to high and the module then functions like the weight_adjuster_output module.

1.2.4.6 neuron_output

The neuron_output module is the output neuron of the neural network and is responsible for generating the output of the system as well as calculating the output error and beginning the backward propagation. As far as forward propagation is concerned it functions almost identically to the neuron module with the exception that it uses the weight_multiplier_output and the sigmoid_output to account for the state transition of the propagation_fsm from forward to backward propagation. Since the neuron_output module is not feed any partial deltas it has to compute its own delta based off of the output error that it computes. See Figure **1.7** for a block diagram of neuron_ouput.

Neuron Output

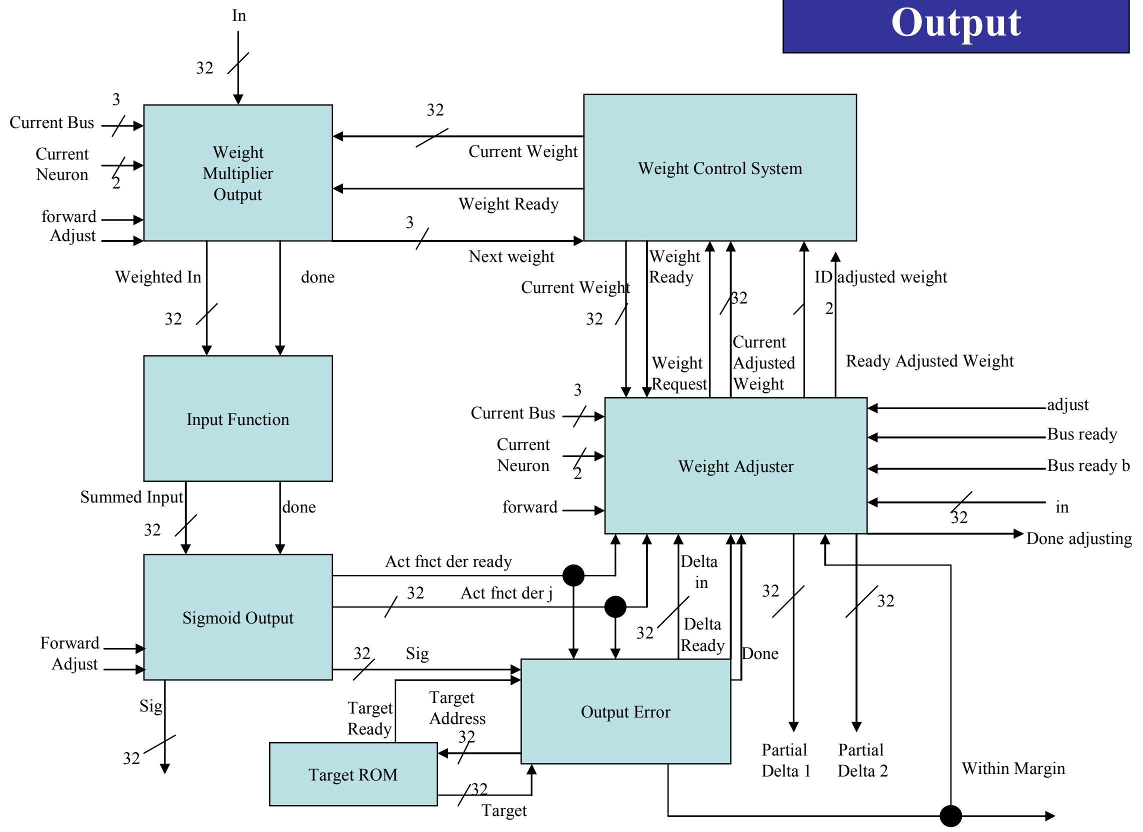


Figure 1.8: Block diagram of neuron_output.

1.2.4.7 weight_multiplier_output

The weight_multiplier_output module has the same functionality of the weight_multiplier module except that the actual multiplications occur at a different time. The weight_multiplier module computes the multiplications during the time that the *forward* signal is asserted high and the weight_multiplier_output module makes the computations while the *adjust* signal is asserted high. The reason for this difference is the propagation_fsm. Once the last neuron_bus sends the last input to the neuron_output the *forward* signal is asserted low and the *adjust* signal is asserted high. This doesn't leave enough clock cycles for the weight_mutliplier module to finish all of the necessary multiplications. This would cause the weight_multiplier to never finish and assert its finished signals high and would thus stop the training of the neural network prematurely. In order to prevent this, the weight_multiplier_output defers its computation of the necessary multiplication to when *forward* is asserted low and *adjust* is asserted high so that it will have the necessary number of clock cycles to finish.

1.2.4.8 sigmoid_output

The sigmoid_output module has all the same functionality as the sigmoid module except that it operates at a different time. Due to the implementation of the propagation_fsm, the sigmoid would receive *summed_input* from input_function on the last clock cycles of the *forward* signal being high. This causes the sigmoid module not to evaluate. In order to get the module to evaluate it was necessary to make the sigmoid_output module become functional when the *adjust* signal is high. The output *sig* is sent to the output_error module and is also the output to the entire system. The ouput *sigDer* is sent to the output_error module as well as the weight_adjuster_output module. After sigmoid_output exerts a high *done* signal, backward propagation can be thought of as having officially begun.

1.2.4.9 output_error

The output_error module calculates the squares output error of the system as well as the delta of the output. It takes as input the *done* signal from the sigmoid_output module and the *forward signal*. The output_error module keeps track of how many samples of the target have been compared to the output of the system. Every time that output_error finished cycling through the samples, when *forward* is high output_error will calculate the total average error over all the samples in the target. If this average is within some acceptable margin of error, then the *within_margin* signal is asserted high to signal that training can stop. While the *adjust* signal is high and *withing_margin* is low the output_error module calculates the squared output error of the system and the delta of the output. Once *done* from sigmoid_out is asserted high output_error begins making its calculations. The process of calculating the squared error and the deltas has been pipelined. First the differences between the output of the system and the target sample value is computed . Then on the next clock cycle this difference value is squared. On the next clock cycle the squared value is multiplied by the *sigDer* value from the sigmoid_outpput module to get the output delta value. On the last clock cycle the output delta value is divided by two and put on the *delta* output of the output_error module and the value of the squared difference divided by two is put on the *error* output. The internal *running_sum* of the output error is incremented by *error*. Also on this last clock cycle, *done* is asserted and sent to weight_adjuster_output module.

1.2.4.10 weight_adjuster_output

The weight_adjuster_output is used in backward propagation to calculate the new values of the weights based on the delta of the output and then propagate the partial deltas to each of the neurons in the next level. It receives as input the *done* signal from output_error, the *delta* signal from output_error, and the current inputs to the output neuron from the neuron_bus. First, while *forward* is asserted high weight_adjuster_output waits for the appropriate neuron_bus to become active. Once the appropriate neuron_bus is active, weight_adjust_output behaves just like weight_multiplier_output in that it loads the current inputs into registers. Once a*adjust* is asserted high weight_adjuster_output begins to load the current value of the weights for the output_neuron. Whenever weight_adjuster_output receives *weight_ready_adj* from the weight_manager, it stores the weight into a register and increments *weight_request_adj* by one. This signal is the id of the current weight being requested for adjusting. Once all the weights have been loaded the internal signal *weights_loaded* is assert high and loading of weights stop. When the *delta* from the output_error is ready, weight_adjuster_output stores the value of *delta* into a register and asserts the internal signal *detla_recieved* high. When the signal *done* is asserted high from the sigmoid_output, weight_adjuster_output stores the value of *sigDer* into a register and asserts the internal signal *der_recieved* high.

Once *der_recieved, delta_recieved, and weights_loaded* are all asserted high the process of adjusting the weights begins. This process is also spread out over a series of clock cycles. On the first clock cycle the adjustment factor is finished being computed and the current adjustment factors are multiplied by the current delta. The partial deltas are also started to be computed when the weights are multiplied by the current delta. On the next clock cycle the partial deltas are finished being computed when the current partial delta values are multiplied by the stored value of the derivative of the sigmoid function. The weights are also adjusted by the adjustment factor on this cycle. On the next clock cycle the new value of the bias weight is put on the *current_adjusted_weight* output and *ready_adjuste_weight* is asserted high. This allows the weight_manager to store the new value of the bias weight. Two clock-cycles later *current_adjusted_weight* is changed to weight two and weight_Manager stores this new value for weight 1. Again two clock cycles later the process is repeated for weight 2. Finally, on the last clock cycle the partial deltas are sent on *partial_delta_out_1*and *partial_delta_out_2* and *done* is asserted high. The partial deltas are sent as inputs to the appropriate neuron_bus.

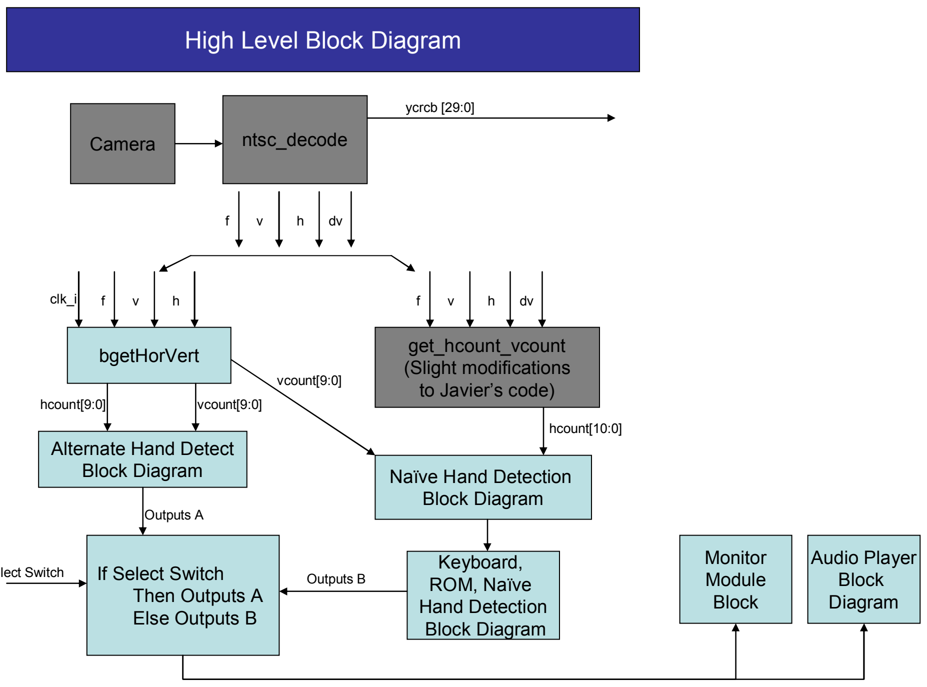
1.2.4.11 network_test_fixture

The network_test_fixture is the testing fixture used for testing the integration of all the neural network components into the full neural network. First the network_test_fixture consisted of two constant inputs being feed into the neural networks inputs and trying to get the neural network to learn to output a constant number. The neural network was able to learn to achieve this. It quickly converged to a number within 3 of the target number and then stopped as 3 was within the chosen error margin of 10 for the test. Next the network_test_fixture was feed the values in the and input look up tables and the target values were taken from the target and look up table. The target was to have TRUE be 127 and FALSE be 0, however the neural network learned TRUE to be 127 and FALSE to be -127. This could have been due to the implementation of the sigmoid function. The range of values for which the sigmoid function was not constant might have been too small. As the error on this training data would jump from 0 (when the output was 127) to a very large number (when the output was -127) . If FALSE would have been defined as -127 the network_test_fixture would have been able to properly learn the function. It should be noted that all of these test were only run in simulation as there was not enough time to test the network in hardware on the labkit.

1.3 Block Diagrams for Hand Detection, Serial, etc. (Behram Mistree)

Please note that all gray blocks came from external creators. All pale blue blocks were written by us.

1.3.1 High Level Block Diagram



High Level Block Diagram
Camera
ntsc_decode
ycrcb [29:0]
f
v
h
dv
clk_i
bgetHorVert
get_hcount_vcount (Slight modifications to Javier's code)
hcount[9:0]
vcount[9:0]
hcount[10:0]
Alternate Hand Detect Block Diagram
Naïve Hand Detection Block Diagram
Outputs A
Select Switch
If Select Switch Then Outputs A Else Outputs B
Outputs B
Keyboard, ROM, Naïve Hand Detection Block Diagram
Monitor Module Block
Audio Player Block Diagram

Figure 1.9: High Level Block Diagram

1.3.2 Keyboard, ROM, Naive Hand Detection Block Diagram

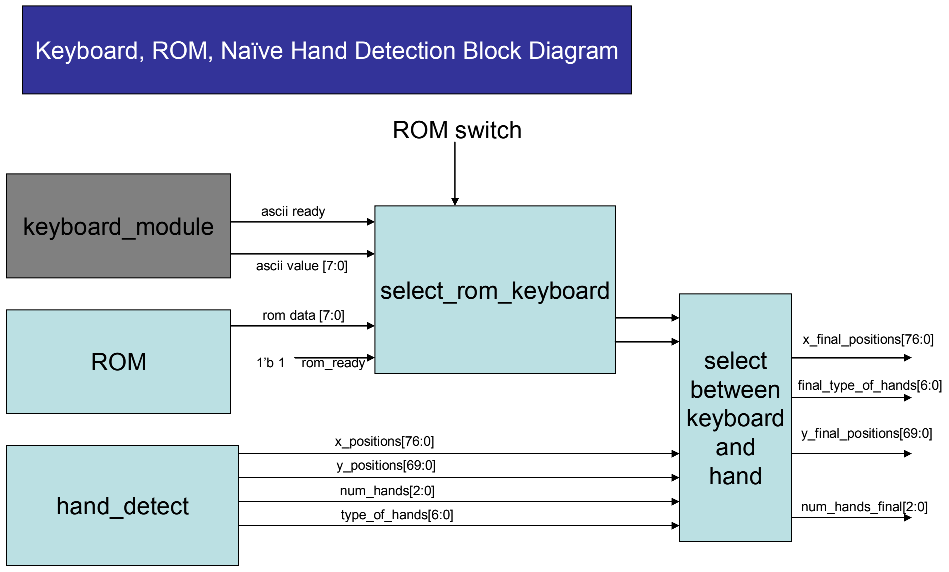


Figure 1.10: Keyboard, ROM, Naïve Hand Detection Block Diagram

1.3.3 Naive Hand Detection Block Diagram

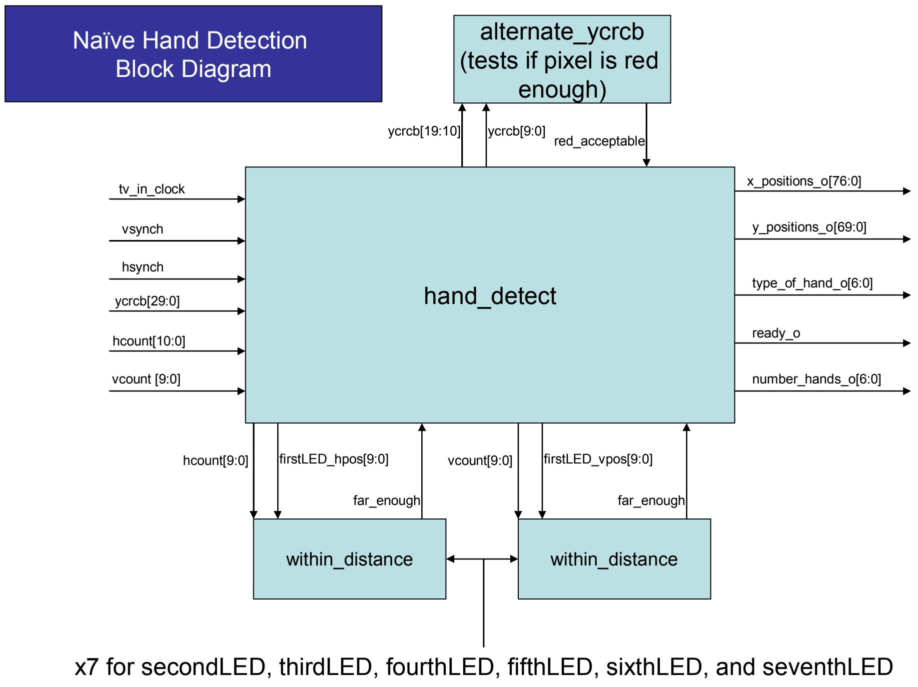
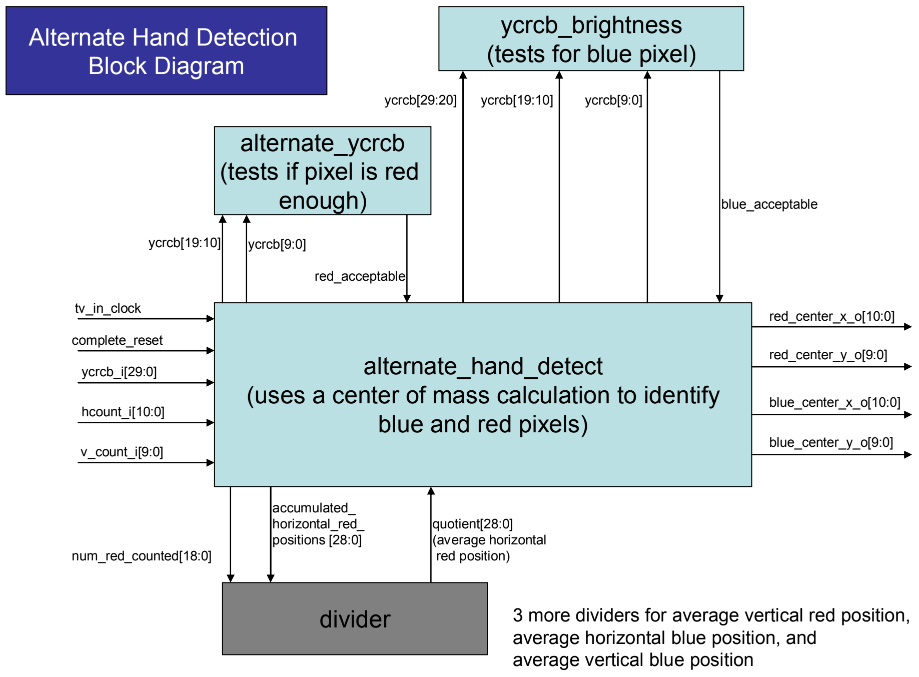


Figure 1.11: Naïve Hand Detection Block Diagram

1.3.4 Alternate Hand Detection Block Diagram



Alternate Hand Detection Block Diagram
ycrcb_brightness (tests for blue pixel)
ycrcb[29:20]
ycrcb[19:10]
ycrcb[9:0]
alternate_ycrcb (tests if pixel is red enough)
blue_acceptable
ycrcb[19:10]
ycrcb[9:0]
red_acceptable
tv_in_clock
complete_reset
ycrcb_i[29:0]
hcount_i[10:0]
v_count_i[9:0]
alternate_hand_detect (uses a center of mass calculation to identify blue and red pixels)
red_center_x_o[10:0]
red_center_y_o[9:0]
blue_center_x_o[10:0]
blue_center_y_o[9:0]
accumulated_ horizontal_red_ positions [28:0]
quotient[28:0] (average horizontal red position)
num_red_counted[18:0]
divider
3 more dividers for average vertical red position, average horizontal blue position, and average vertical blue position

Figure 1.12: Alternate Hand Detection Block Diagram

1.3.5 Audio Player Block Diagram

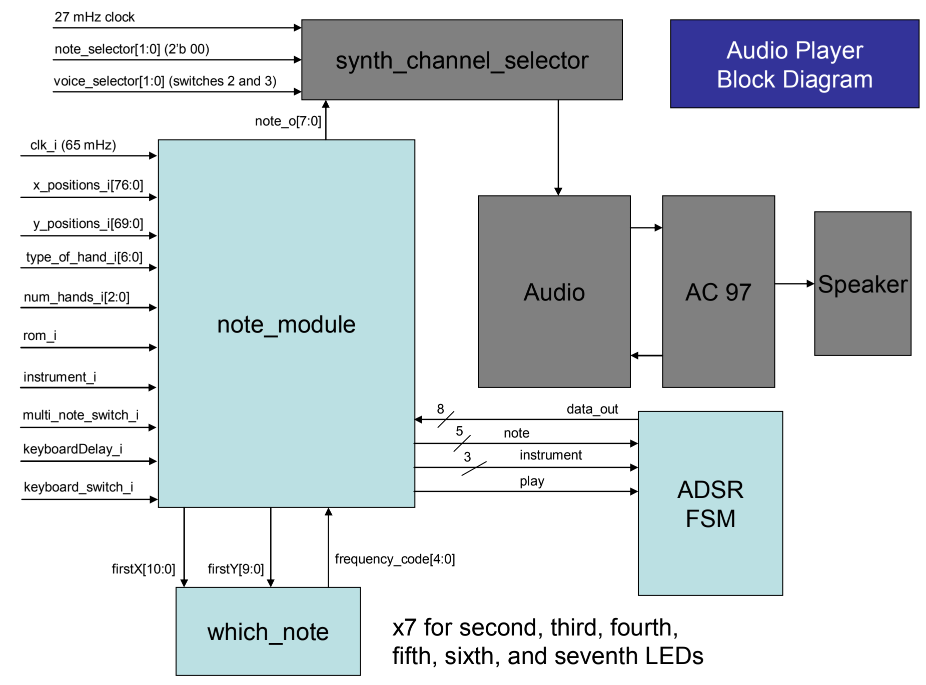


Figure 1.13: Audio Player Block Diagram

1.3.6 Monitor Module Block Diagram

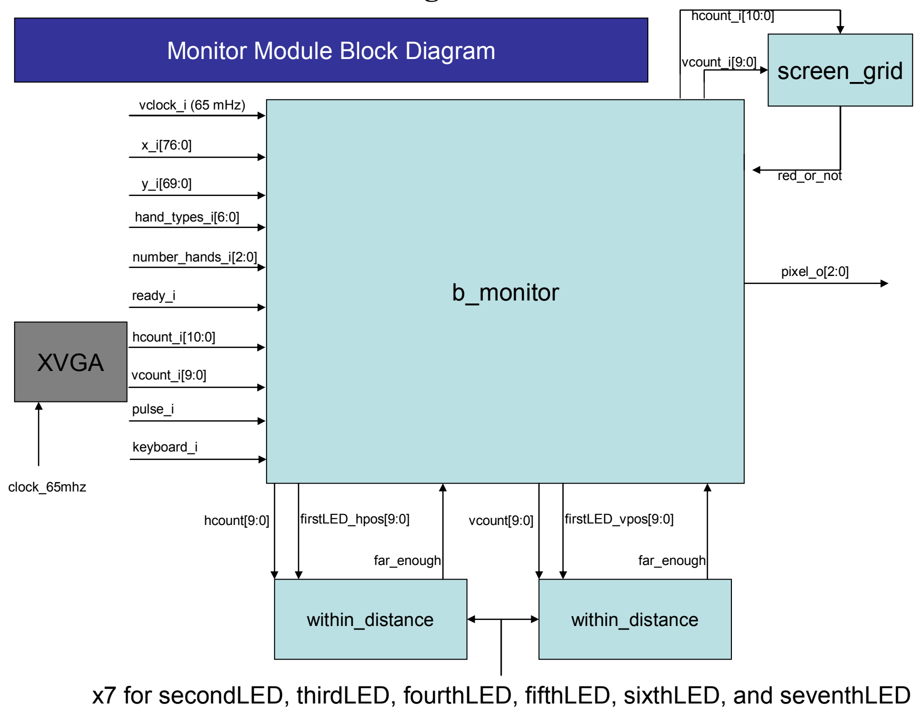


Figure 1.14: Monitor Module Block Diagram

1.3.7 Serial Block Diagram

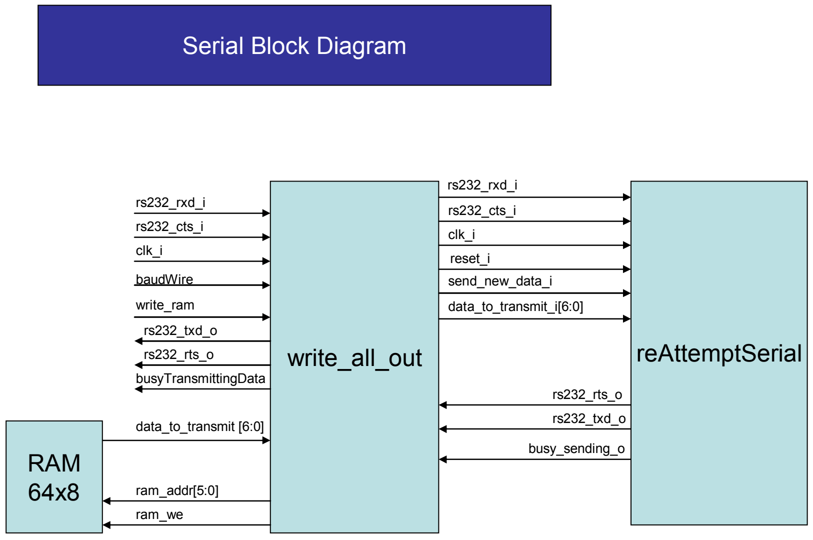


Figure 1.15: Serial Block Diagram

1.4 Module Desccriptions for Hand Detection, Serial, etc. (Behram Mistree)

1.4.1 hand_detect

At a high level, the hand_detect module is simply supposed to take in camera data and output the horizontal and vertical positions of any LEDs within the camera data. To accomplish this task, the hand_detect module takes in YcrCb data directly from the camera. The hand_detect module filters each pixel that comes in as either suitably red or not suitably red. The hand_detect module keeps track of the last ten pixels recorded. If all of the last ten pixels were categorized as suitably red, the hand_detect module does two things: (1) it checks whether the current position is adequately different from an LED position it already has stored, and (2) it checks whether the maximum number of LEDs has not yet been detected for this frame of camera data.

If both of these conditions are met for the aforementioned tenth consecutive suitably red data point, hand_detect stores the horizontal and vertical positions of the tenth point a distinct LED. In addition, it increments the stored number of LEDs the module believes it has seen as well as multiplying.

After a full frame of data, a ready signal is asserted which guarantees the validity of the outputs. This ready signal is high for one clock cycle, after which the data continues to change.

The hand_detect module takes as inputs *vclk_i*, *ycrcb_i*, *hcount*, *vcount*, *hsynch*, and *vsynch*.

The *vclk_i* input is a simple clock signal timed to the 27 mHz camera's clock. The *ycrcb_i* input is a 30-bit long value that is updated on every clock edge. It contains the luminosity (10 most-significant-bits), red chrominance (middle 10 bits), and blue chrominance (10 least-significant-bits) of each pixel. *hcount* is an 11 bit number that corresponds to the horizontal position within a frame of camera data of the data that is in *ycrcb_i*. Similarly, *vcount* is a 10-bit number which corresponds to the vertical position of the data that is in *ycrcb_i*. *hsynch* and *vsynch* are each 1 bit numbers that correspond to the horizontal and vertical syncing of the data represented by *ycrcb_i*. When *hsynch* or *vsynch* are high, we know that the data that is being presented in *ycrcb_i* is invalid, and should not be used for calculations.

The hand_detect module issues five relevant outputs: *x_positions_o*, *y_positions_o*, *type_of_hand_o*, *ready_o*, and *number_hands_o*.

The hand_detect module is capable of detecting up to seven distinct LEDs. *x_positions_o* is a 77 bit number that corresponding to all of the potential seven LEDs' horizontal positions. The 11 least-significant-bits map to the horizontal position of the first LED detected, the next 11 bits map to the horizontal position of the second LED detected, etc.

Similarly, *y_positions_o* is a 70 bit number that corresponds to all of the potential seven LEDs' vertical positions. The 10 least-significant-bits map to the vertical position of the first LED detected, the next 10 bits map to the vertical position of the second LED detected, etc.

number_hands_o is a 3 bit output. Its value corresponds to the number of distinct LEDs that were detected in a full scan of the data. For example, if we detected 4 distinct LEDs after scanning the data, *number_hands_o* would equal 3'b 100.

The original specification of our design called for detection of two, differently colored LEDs. The purpose of the 7 bit output *type_of_hand_o* is to distinguish which type of LED was detected. That is, if the least-significant-bit of *type_of_hand_o* is high, we know that if LEDs were detected (ie *number_hands_o* greater than 0), the first LED that was detected was red. Similarly, if the second bit of *type_of_hand_o* is low, we know that if at least two LEDs were detected (ie *number_hands_o* greater than 1), the second LED detected was blue. (Please note that for simplicity, we later decided that all LEDs should just have a default type value of 1.)

1.4.2 alternateHandDetect

The alternateHandDetect module is responsible for taking in a stream of camera data and returning the centers of the LEDs detected within that data. Please refer to the block diagram labeled Alternate Hand Detection Block Diagram for specific connections.

The alternateHandDetect module filters out red pixels (with appropriate brightness and red chrominance as determined by the ycrcb_acceptable module) and filters the data for blue pixels (with appropriate brightness and blue chrominance as determined by the ycrcb_brightness module).

The module keeps a running count of the number of red pixels detected as well as their horizontal and vertical positions. As you can see from the diagram labeled Alternate Hand Detection Block Diagram, we divide these values using a separate, pipelined divider module that returns a 29-bit number.

Because division is an intensive process, the actual result of the division takes 31 clock cycles to propagate. When we sample the results of this division, we reset the red and blue pixel accumulators. Therefore, we could potentially miss 31 pixels worth of data.

Dividing the horizontal and vertical positions by the number of red pixels detected gives the center of mass of red pixels. A symmetric method is used to determine and return the center of mass of the blue pixels. After a full set of clock cycles, the center of mass of red pixels is returned and the center of mass of blue pixels is returned.

The relevant inputs of the alternateHandDetect module are *clk_i*, *complete_reset_i*, *ycrcb_i*, *hcount_i*, and *vcount_i*.

The *clk_i* signal represents the 27 mHz clock coming in from the camera. The *ycrcb_i* input is a 30-bit long value that is updated on every clock edge. It contains the luminosity (10 most-significant-bits), red chrominance (middle 10 bits), and blue chrominance (10 least-significant-bits) of each pixel. The *hcount_i* and *vcount_i* signals represent the horizontal and vertical position of the pixel that corresponds to each YCrCb value.

The *complete_reset_i* signal is simply an input that allows the user to reset all his or her registers to an initial state.

Although there are numerous debugging outputs which were primarily used to demonstrate functionality, the relevant outputs of the alternateHandDetect module are *blue_center_x_o*, *blue_center_y_o*, *red_center_x_o*, and *red_center_y_o*.

The *blue_center_x_o*, *blue_center_y_o*, *red_center_x_o*, and *red_center_y_o* outputs correspond to the horizontal and vertical components of the red and blue centers of mass of a previous frame's video data. All these outputs are held static, except during a one-clock-cycle refresh where they are updated with the values of the newest frame of data that has been processed.

As you can see from the block diagram labeled High Level Block Diagram, the four outputs of this function are fed into something of a mux which determines whether to send on the information from the ROM, keyboard, or other hand detection module to the b_monitor and note_module modules for display and sounding respectively.

1.4.3 b_monitor.v

The monitor module was designed to take in the x and y positions of up to seven distinct LEDs and display their positions on a grid on the computer screen. The simple logic describing the grid is contained in the screen_grid module. To see the interface to this module and other details of the b_monitor module, please reference the block diagram labeled Monitor Block Diagram.

The b_monitor module takes in as inputs *x_i*, *y_i*, *hand_types_i*, *number_hands_i*, *ready_i*, *vclock_i*, *hcount_i*, *vcount_i*, *pulse_i*, and *keyboard_i*.

The monitor module is clocked differently from the camera. Specifically, while the camera takes in a 27 mHz clock, the clock input of the b_monitor module, *vclock_i*, is clocked at 65 mHz.

x_i is a 77 bit input that maps to the horizontal centers of each LED recorded. Because the module was designed to support up to seven distinct LEDs, the 11 least-significant-bits of *x_i* correspond to the horizontal position of the center of the first LED detected by one of the hand detection modules, its next 11 bits correspond to the horizontal position of the center of the next LED detected, etc.

Similarly, *y_i* is a 70 bit input that maps to the vertical centers of each LED recorded. (The 10 least-significant-bits correspond to the vertical position of the center of the first LED detected by one of the hand detection modules, the next 10 bits correspond to the vertical position of the center of the next LED detected, etc.)

Our design called for our hand detection to distinguish seven distinct LEDs of two different colors. *hand_types_i* is a seven bit number. The value of each bit corresponds to what type of LED is detected. That is, if the least-significant-bit of *hand_types_i* is high, we know that if LEDs were detected (ie *number_hands_i* greater than 0), the first LED that was detected was red. Similarly, if the second bit of *hand_types_i* is low, we know that if at least two LEDs were detected (ie *number_hands_i* greater than 1), the second LED detected was blue.

As you can see from the block diagram labeled High Level Block Diagram, *x_i*, *y_i*, *hand_types_i*, and *number_hands_i* come from either the alternate_hand_detect module or the ROM, keyboard, or the naïve hand detection module.

hcount_i and *vcount_i* are 11 and 10 bit signals respectively. *hcount_i* represents the value of the horizontal value of the particular pixel that is being painted on the screen, while *vcount_i* represents the vertical value of the particular pixel on the screen. As you can see from the block diagram labeled Monitor Module Block Diagram, these signals come directly from the XVGA module. In essence, if *hcount_i* and *vcount_i* are within a sufficient radius of the horizontal and vertical positions of detected LEDs, a black pixel is painted. Otherwise, a white pixel is painted.

pulse_i is a single bit pulse signal. It is asserted high for only one 65 mHz clock cycle. The time between these pulses can be thought of as the minimum duration a note must be held. (For our project, the time between *pulse_i*'s pulses was roughly an eighth of a second.) Clocking changes in notes to these pulses helps reduce noise in the display.

Because aspects of the hand-detection system were susceptible to random noise, while music is being played based on hand positions, the monitor module only displays data when at least two different LEDs are seen. However, when displaying notes from the keyboard or a ROM input, the data is pure, and we do not have any problems with random noise. As such, we do not need to require that more than one hand is detected when displaying note values from a keyboard or ROM input. Therefore, we use a one-bit input called *keyboard_i*. When *keyboard_i* is high, it indicates that we are either reading note data from a ROM or keyboard. When low, *keyboard_i* indicates that we are displaying note values from hand detection and need to require that more than one hand is seen.

The only non-debugging output in the b_monitor module is the three-bit value *pixel_o*. *pixel_o* corresponds to the RGB value that you want the monitor to display at the point specified by *hcount_i* and *vcount_i*.

1.4.4 note_module

note_module is responsible for taking in a list of LED positions and converting those to a single synthesized sound wave. As you can see from the block diagram labeled Audio Player Block Diagram, this synthesized sound wave output is then sent out through speakers and to the user.

Also, please note in the Audio Player Block Diagram that note_module makes use of the verilog module which_note. which_note is a simple look-up table that maps x and y positions to frequency values that are then mapped to a waveform using ADSR FSM which is finally played using Synth Channel Selector.

note_module takes as input *clk_i*, *x_positions_i*, *y_positions_i*, *type_of_hand_i*, *min_note_duration_i*, *num_hands_i*, *rom_i*, *instrument_i*, *multi_note_switch_i*, *keyboradDelay_i*, and *keyboard_switch_i*.

clk_i is the same 65 mHz clock that is used by the b_monitor module.

x_positions_i is a 77 bit input that maps to the horizontal centers of each LED recorded. Because the module was designed to support up to seven distinct LEDs, the 11 least-significant-bits of *x_i* correspond to the horizontal position of the center of the first LED detected by one of the hand detection modules, its next 11 bits correspond to the horizontal position of the center of the next LED detected, etc.

Similarly, *y_positions_i* is a 70 bit input that maps to the vertical centers of each LED recorded. (The 10 least-significant-bits correspond to the vertical position of the center of the first LED detected by one of the hand detection modules, the next 10 bits correspond to the vertical position of the center of the next LED detected, etc.)

Our design called for our hand detection to distinguish seven distinct LEDs of two different colors. *type_of_hand_i* is a seven bit number. The value of each bit corresponds to what type of LED is detected. That is, if the least-significant-bit of *type_of_hand_i* is high, we know that if LEDs were detected (ie *num_hands_i* greater than 0), the first LED that was detected was red. Similarly, if the second bit of *type_of_hand_i* is low, we know that if at least two LEDs were detected (ie *num_hands_i* greater than 1), the second LED detected was blue.

min_note_duration_i is a single bit pulse signal. It is asserted high for only one 65 mHz clock cycle. The time between these pulses can be thought of as a the minimum duration a note must be held. (For our project, the time between *min_note_duration_i*'s pulses was roughly an eighth of a second.) Clocking changes in notes to these pulses helps reduce noise in the note audio.

instrument_i is a single bit value that chooses between synthesizing note audio as a violin (*instrument_i* is a logical 1) and a guitar (*instrument_i* is a logical 0).

multi_note_switch_i is a single-bit input that allows a user to select whether to apply an audio effect. When *multi_note_switch_i* is high, the audio waveform is played in "tremolo" (a musical term for playing a note in rapid succession over and over). When *multi_note_switch_i* is high, the note is played regularly.

Because aspects of the hand-detection system were susceptible to random noise, while music is being played based on hand positions, the note_module module only displays data when at least two different LEDs are seen. However, when displaying notes from the keyboard or a ROM input, the data is pure, and we do not have any problems with random noise. As such, we do not need to require that more than one hand is detected when displaying note values from a keyboard or ROM input. Therefore, we use two one-bit inputs called *keyboard_i* and *rom_i*.

When *rom_i* is high, it indicates that we are reading note data from the ROM. When low, *rom_i* indicates that we are playing notes from either hand detection positions or the keyboard.

When *keyboard_switch_i* is high, it indicates that we are reading note data from the keyboard. When low, *keyboard_switch_i* it indicates that we are playing notes from either hand detection or the ROM. We distinguish between *rom_i* and *keyboard_switch_i* because keyboard data requires a specific delay to ensure that the correct note value gets sounded.

This delay is represented outside of note_module. Essentially, we delay the time at which the keyboard's values are sampled. This delay is represented by *keyboardDelay_i*. *keyboardDelay_i* goes high for one clock cycle two clock cycles after the ascii data from the keyboard is ready. When *keyboardDelay_i* is high, the data from the values of *x_position_i*, *y_position_i*, *type_of_hand_i*, and *num_hands_i* are sampled and their values converted to a note value to be played.

Unfortunately, despite our efforts, there is a one note delay between values that you play and values that you hear. For instance, if you hit an 'A' and then a 'B' and then a 'C' on the keyboard. When you hit 'B,' you will hear the note corresponding to an 'A'; when you hit 'C,' you will hear the note corresponding to 'B.'

There is only one non-debugging output from note_module: *note_o*. *note_o* is an 8-bit value that corresponds to the waveform that will be played to the user.

1.4.5 bgetHorVert

The bgetHorVert module takes in data from the decoded camera signal and outputs the horizontal and vertical pixel position that that data corresponds to.

As seen in the block diagram labeled High Level Block Diagram, the bgetHorVert module takes as input four one-bit signals *clk_i, f_i, v_i,* and *h_i*. *clk_i* corresponds to the 27 mHz clock from the camera. *h_i* corresponds to a pulse whose rising edge indicates that data coming from the camera is associated with a new horizontal line of camera data (as shown in the online labkit information: http://www-mtl.mit.edu/Courses/6.111/labkit/appnotes/xapp286_04.pdf**)**. *v_i* corresponds to a pulse indicating that the data about to be presented comes from a new vertical line. *f_i* indicates the field value of the particular vertical row being scanned. The camera data is interleaved: all even rows of data are presented and then all the odd rows of data. When *f_i* is high, it implies that the row of data being read is even. When *f_i* is low, it implies the row of data being read is odd.

The bgetHorVert module outputs two 10-bit signals: *hor_o* and *vert_o*. *hor_o* and *vert_o* correspond to the horizontal and vertical positions of the data that are being presented. They are updated on each clock cycle.

Roughly speaking, *hor_o* corresponds to the number of clock cycles that have passed since the last time *h_i* was pulsed. We say roughly speaking here because we know that the camera's field of view is only 750x625 pixels. Therefore, whenever *hor_o* exceeds 749, we set it equal to a default, constant value greater than 750.

vert_o is a double count of the number of times *h_i* was pulsed between changes in *f_i*. Again, because we know the vertical position of the camera's field of view cannot exceed 625, if our count exceeded 625 *vert_o* was uniformly set to a constant value greater than 625.

1.4.6 write_all_out

The writeAllOut module is responsible for writing the contents of a ram out through a serial port to a computer. As you can see from the Serial Block Diagram, writeAllOut accomplishes this task by controlling a smaller module, reAttemptSerial, which writes an individual 7-bit number across the serial line. (Please see next module description for more information on reAttemptSerial.)

Because our RAM was larger than the number of relevant values it contained, writeAllOut only writes a range of values out through the serial port (in our case, values with addresses from 5 to 56). However, two parameters corresponding to the first address that you want your RAM to write from and the last address you want your RAM to write from can be specified to increase or decrease the range of values that are being written out from your RAM.

The relevant inputs that we receive (all labeled on the Serial Block Diagram) are *rs232_rxd_i*, *rs232_cts_i*, *clk_i*, *baudWire*, *write_ram*, and *data_to_transmit*.

We ignored all incoming data because we assume that the computer we write to will not be simultaneously trying to send data back to us. Therefore, although we receive *rs232_rxd_i* and *rs232_cts_i* as inputs, we ignore them.

clk_i is our module's clock. It is the generic 65 mHz clock from the labkit. We choose to transmit at the baud rate of 57,600 because it is an integer multiple of our 65 mHz clock. Therefore, we are limited in our data transmission rate to one character every 1125 peaks from *clk_i*. Because it would be cumbersome to incorporate such a delay directly into all of our circuitry, we take in the input *baudWire*. *baudWire* goes high for one clock cycle of our 65 mHz clock whenever 1125 peaks of the 65 mHz clock have expired.

When the input *write_ram* goes high, writeAllOut initializes its parameters to begin transmitting data. When *write_ram* goes low after going high, writeAllOut begins writing all its data to the serial port.

The input *data_to_transmit* is the seven-bit data value to transmit across the serial line.

The writeAllOut module produces four outputs: *rs232_txd_o*, *rs232_rts_o*, *busyTransmittingData*, *ram_addr*, and *ram_we*.

rs232_txd_o corresponds to the binary piece of data that is currently being written across the serial port. *rs232_rts_o* corresponds to the ready-to-send signal that the computer is expecting. Because our serial communication is remarkably one-sided (the labkit is only writing to the computer instead of listening to it), we just generically hold this signal at an arbitrary constant value.

busyTransmittingData is a signal to whatever module is controlling writeAllOut. *busyTransmittingData* is high whenever writeAllOut has not finished writing data after *write_ram* was asserted.

ram_addr is a 6 bit number. It corresponds to the address of the RAM with the data that writeAllOut is currently writing to the serial port.

ram_we is the write enable signal that goes to the RAM. Presumably, the user does not want to be writing data to the ram while he or she is writing data from the RAM to a serial line. Therefore, *ram_we* is constantly grounded.

1.4.7 reAttemptSerial

As shown in the Serial Block Diagram, the module reAttemptSerial is controlled by writeAllOut. The function of reAttemptSerial is to send a 7-bit piece of data specified by writeAllOut across the serial port to a computer whenever requested to begin this transmission by writeAllOut.

reAttemptSerial's inputs are: *rs232_rxd_i*, *rs232_cts_i*, *clk_i*, *send_new_data_i*, and *data_to_transmit_i*.

We ignored all incoming data because we assume that the computer we write to will not be simultaneously trying to send data back to us. Therefore, although we receive *rs232_rxd_i* and *rs232_cts_i* as inputs, we ignore them.

clk_i is the clock signal on which reAttemptSerial's timing is based. Instead of being a 65 mHz clock signal, *clk_i* is clocked at 57,600 Hz so as to agree with our established baud rate.

send_new_data_i is a one-bit signal. When it goes from high-to-low, reAttemptSerial begins transmitting the seven-bit number represented by *data_to_transmit_i* to the computer.

As seen in the Serial Block Diagram, reAttemptSerial has three outputs: *rs232_txd_o*, *rs232_rts_o*, and *busy_sending_o*.

rs232_txd_o is a single-bit value corresponds to the binary piece of data that is currently being written across the serial port. *rs232_rts_o* corresponds to the ready-to-send signal that the computer is expecting. Because our serial communication is remarkably one-sided (the labkit is only writing to the computer instead of listening to it), we just generically hold this signal at an arbitrary constant value.

The output signal *busy_sending_o* goes high when reAttemptSerial begins transmitting data. It remains high until reAttemptSerial is done transmitting the data.

2. Testing and Debugging

2.1 Testing and Debugging Issues for Subtractive Synthesis (Alexander Sanchez)

2. 1.1 General Testing Strategy for Subtractive Synthesis

The general strategy used for testing and debugging was to first test the modules in behavioral simulations in ModelSim and once the modules passed these simulations they were tested on the labkit. Testing on the labkit for audio signals and waveforms consisted of displaying the signals on the logic analyzer and comparing the displayed waveforms to the expected waveforms. Audio signals under went additional testing in that they also had to sound good, so the audio signals would be feed into the ac97 chip on the labkit and listened to on headphones to see if the signals sounded properly.

2.1.2 Issues Sampling Stored Waveforms

The major bug encountered with the subtractive synthesis portion of the project was getting the oscillating signals stored in the ROMs to play at the right frequency. Originally the 27mHz signal was dropped to a lower frequency signal and that lower frequency signal was used as a ready signal to the modules involved with extracting the waveforms out of the ROM. This worked fine when we were using the 27mHz clock at the start of the project. However, we eventually decided to use the 65mHz clock in order to get cleaner signals with more resolution. This caused some major problems with the oscillating signal generators since they were using a ready signal that was based off of a 27 mHz clock. The problem was that the ready signal didn't always line up with the positive edge of the 65mHz clock. This would cause the signal generators to get 2 or 3 ready signals a clock cycle and would thus change the rate the roms were being sampled at and in turn the frequency of the oscillating signals. The solution to this problem was described in detail in the module description for note_to_frequency2.

2.1.3 Issues Involving Signed Number Calculations

Another issue was with using signed numbers. Originally all of the registers that were used in any way with the oscillating signals were declared signed, however, the wires that were assigned to these registers were not declared as signed and so the signals that came out of the oscillating signal generator modules were in an unsigned form. When these signals were later added in the synth_channel_selector to get polyphonic sounds with and without voice over the resulting signal would be heavily distorted. This bug took a long time to track down since all of the appropriate registers were declared as signed. Originally the problem was thought to have to do with a timing constraint not being satisfied, however, after starting to develop the neural network and a having to deal a great deal with signed numbers, it was discovered that wires had to be declared signed in addition to the registers they were assigned to. In the end, the solution to this bug proved to be a quick fix of adding a signed declaration to all the appropriate wires. After making this fix, the polyphonic signals with and without voice became very clean.

2.1.4 Issues Creating ADSR Envelopes

Creating the ADSR envelopes for the guitar and violin also proved to create many issues. Although there was much information to be found online about what an ADSR envelope is[2], there was very little information about how to generate an accurate ADSR envelope for an instrument. Eventually a website was found that described a process for creating linear and exponential envelopes in MatLab[3]. An article describing subtractive synthesis was found at wikipedia.org that had audio samples of the signals at various points in the process of subtractive synthesis[4]. After much studying of these signals and of the process for creating envelopes in MatLab, attempts were made at creating an ADSR envelope for a guitar in MatLab. This proved to be a grand procedure of trial and error. Eventually the audio signals that we were trying to match were opened in Audacity which allows the user to clearly see the waveform of the audio signal. Eventually, after many tries, an envelope was created that could synthesize a guitar sound that was reasonably close to the audio sample we were going off of. See figure **2.1** for a plot of the ADSR envelope of a guitar

2.1.5 Issues with the ADSR FSM

2.1.5.1 Issues With Note Duration

The adsr_FSM module also presenting some interesting issues that need to be debugged. First there was the issue of how long the envelope should last, as the duration of the envelope being applied to a signal would define the duration of a note being played. Originally, the address that was being access from the adsrROM was being incremented by one every clock cycle. However, with a 65mHz clock, 256 samples would be played at a rate that would not be audible. To solve this problem we introduced a counter that would produce an enable signal after counting off a certain number of clock cycles of the 65mHz clock. This enable signal would then tell the adsr_FSM when to increment the address being accessed from the adsrROM. However, the question then arose of what the counter should be counting to. This problem was solved by a trial and error process. Being as we wanted an audible signal, this process took longer than desired as we would change the duration of the counter, recompile the Verilog, and then upload the code to the labkit and see if we could hear the signal. After trying many different values we decided on using 17'b111_111_111_111_111_11. Using this for number we were able to create a 496Hz enable signal. With this enable signal, a note that goes through the entire ADSR envelope with no interruptions would last for approximately 0.51 seconds.

2.1.5.2 Issues with Sounding Repeated Notes

The issue of making a note sound repeatedly proved to become a problem when we tried integrating the hand detection system with the sound synthesizes portion. The adsr_FSM was tested using a button on the labkit as the play signal input to the adsr_FSM. However, due to the fact that it is very hard to hit a button and make it give a high signal for exactly one clock cycle, the adsr_FSM had very poor repeated note functionality. Essentially you couldn't play the same note twice, another note had to be played first. Although, you could play one note and then interrupt it by playing a new note. This resulted in not all of the notes that the hand detection generated to be played. However, we soon realized that it was possible for the play signal generated by the hand detection module to last for exactly one clock cycle. With this constraint it became possible to change the adsr_FSM to allow for repeated notes.

2.2 Testing and Debugging Issues with the Neural Network (Alexander Sanchez)

2.2.1 General Testing Strategy for the Neural Network

The general testing and debugging strategy used for the neural network was to first run behavioral simulations in ModelSim for each individual module. Once the modules were running satisfactory in ModelSim it became time to integrate them and run more behavioral simulations in ModelSim. The last part of the testing strategy we planed was to test the full neural network on the labkit. However, we were only able to test one portion of the system on the labkit and were unable to test the entire neural network on the labkit. Because of this we were also unable to test integrating the neural network with the hand detection modules. The portion that we were able to test was the weight management system that loaded weights from a bram, changed the weights, saved the new weights in the bram, and then transmit the values of the weights to a computer via the labkit's serial port.

2.2.2 Issues with Scaling Numbers

The neural network provided us with an abundance of interesting issues to tackle and debug. The biggest issue encountered with the neural network was deciding on which bits to take out of a number in order to get a smaller number. For example, what bits should be taken out of the multiplication of two 8 bit signed numbers in order to get an 8 bit signed number out? Originally, we were striving to use as few wires as possible. Since we wanted the end result of the neural network to be an 8 bit signed signal that could be feed to the ac97 chip as an audio signal, we thought it would be a good idea to try and only pass 8 bit signed signals between modules. However, we soon realized that this required a lot of bookkeeping in order to make sure that the signed bit was not lost in the reductions that took place between levels.

2.2.3 Issues with Signed Number in Case Statements

The first implementation of the sigmoid function involved centering the function on zero and using a case statement to select the output. However, we soon realized that negative inputs to the sigmoid function were returning the constant high value (127) instead of the constant low value (-127). It turns out that case statements in Verilog don't support negative decimal numbers. To get around this we had to put the negative values in binary form in the case statement since the case statement would treat all cases as unsigned numbers. This provided a fix to the problem of getting the right value out of the sigmoid function. The waveforms for the simulation of the sigmoid module can be seen in figure **2.1**.

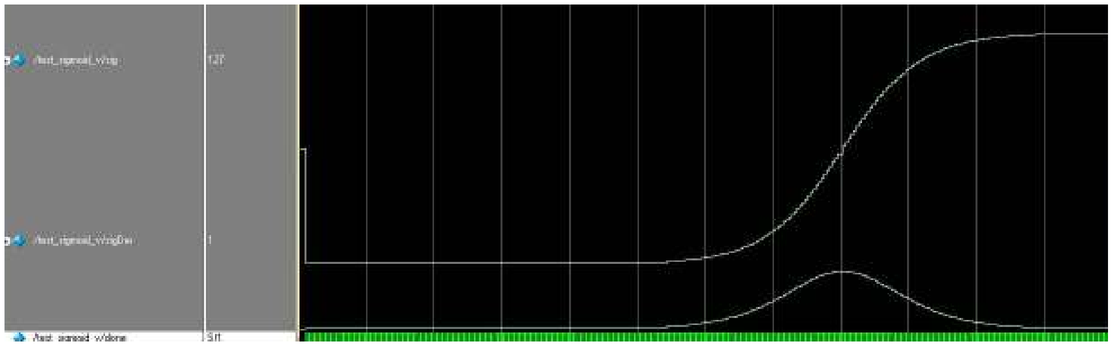


Figure 2.1: Simulation Waveform of the Sigmoid Module

2.2.4 Possible Solution to Issues with Signed Numbers

Eventually, though, we decided that it would be easier to get 8 bit numbers out of larger bit numbers if we only used unsigned numbers and then converted the final output of the system to an unsigned number by subtracting off some constant factor. This engineering decision immediately eliminated all the bookkeeping associated with keeping track of the signed bits. It also made getting a scaled 8 bit unsigned number out of a larger bit unsigned number a lot easier as we would only have to take the 8 most significant bits. This soon proved to be deceiving as it introduced huge precision errors that would cause the neural network to quickly converge to zero as the weights became overloaded and also converged to zero.

2.2.5 Precision Error Issues

The precision error wasn't fully realized until all the modules in the network had been written and integrated into a test fixture that we were running simulations on. The first few simulations we ran with a very small learning factor of 0.5 and saw that the weights and the output would go to zero quickly. We then tried a really large learning factor of 100 and the neural network produced the same convergence problems. We then tried expanding the range of non-constant outputs of the sigmoid function but this resulted in the same results.

We then realized that the weights wanted to changed by small amounts, but by using 8 bit unsigned number it became nearly impossible to accurately represent really small decimal changes in the weights which caused the deltas to get large and the weights to go to zero. The solution we decided on for this problem was to make all the registers and wires used in the computations in the forward and backward propagation algorithms 32 bit numbers. We also realized that if all the numbers were going to be the same bit size, then there would be very little, if any, bookkeeping necessary for keeping track of the signed bit. So we finally decided on making all the registers and wires involved in the forward and backward propagation algorithms 32 bit signed numbers. In order to get an 8 bit audio signed audio signal we planned on taking the 8 most significant bits out of the 32 bit output of the neural network, and if that signal ended up being too low we would take the next 8 bits in order to scale the magnitude up.

2.2.6 Issues with Constraints on Size of Stored Weights

Although using 32 bit signed numbers would get rid of the precision errors we were having before, we soon ran into yet another problem with the size of numbers we were using. The labkit to computer via serial port mechanism we were using was strictly restricted to transmitting 8 bit numbers where the most significant bit is a garbage bit and only bits 0 through 7 can be used for storing data. This constraint on the amount of data we could transfer was one of the original reasons we decided on using 7 bit weights. However, if we used 32 bits numbers everywhere except the weights we would still have precision errors if we would try to adjust the weights by very small amounts, which is what the backward propagation algorithm does.

The way we solved this problem was to make the weights 32 bit signed numbers but store the weights in 5 separate 8 bit locations where bit 8 is unused and is some garbage value. This change increased the necessary size of the BRAM used to store the weights from 8x21 to 8x128 since there are 21 weights and each would require 5 locations to store the entire weight. In order to implement this design change it became necessary to alter the design of the weight_manager module, although the changes made were small. The loading and saving sequences were simply increased and carefully bookkeeping was taken to ensure that the right bits of the weights were written and loaded from the correct address. Fortunately, the weight_manager module and the weight_manager_controller module were abstracted from each other enough so that the changes made to the weight_manager module had no effect on the weight_manager_controller, weight_address_bus, or the weight_bus modules.

2.2.7 Simulation Results of Neural Network

Once all of these changes were made it was time to test the neural network in simulation again. During the simulations we tried to run the neural network for several hundred epochs only to realize that the neural network would stop running after a few cycles. The bug, however, was quickly found to be a timing issue with resetting a control signal that resulted in the propagation_fsm module getting stuck in a single state.

In general, timing constraints proved to be very tricky to get right in the neural network. We found that there were a lot of hidden delays associated with always blocks and accessing memories that we had not originally accounted for.

Once the timing constraints were fixed we again started to run simulation. This time we tried to train the network to take in two constant inputs and produce some constant output. In this case we arbitrarily choose 1000 and 350 as the inputs and 90 as the output and set the error margin to 10. The neural network then was able to get the value of 87 out and then stop since it was within the acceptable error margin. In fact, it learned to compute this very fast as it only took 4 epochs. The waveforms of this simulation can be seen in figure **2.2**. We then simulated the neural network trying to learn to compute the Boolean AND function. However, we found that the network wouldn't converge and the partial deltas would stop propagating after a few epochs. It turned out that after switching to 32 bits we forgot to change the size of the adjusted weights that were being saved. This resulted in 6 bit weights being save which again introduced precision errors that caused the weights and deltas to go to zero. After changing the size of the weights being adjust to the proper 32 bit size we again reran the simulation of learning the AND Boolean function. The waveforms for this simulation can be seen in figure **2.3**.

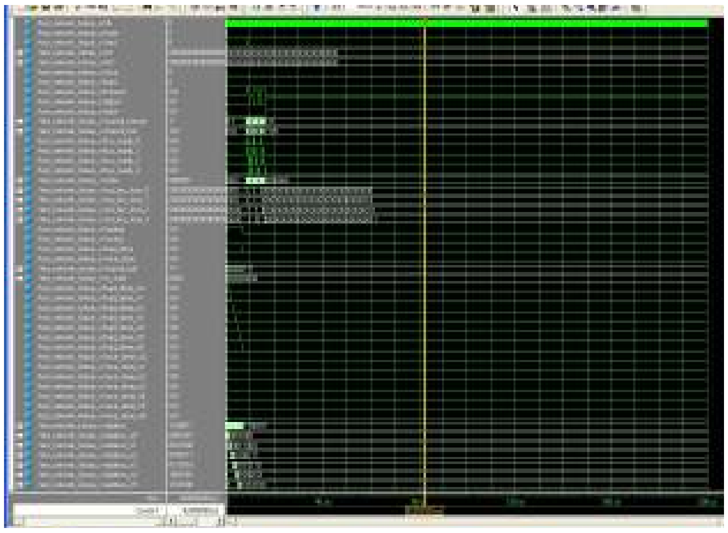


Figure 2.2: Simulation waveform of neural network being trained to computer a constant value from two constant valued inputs.

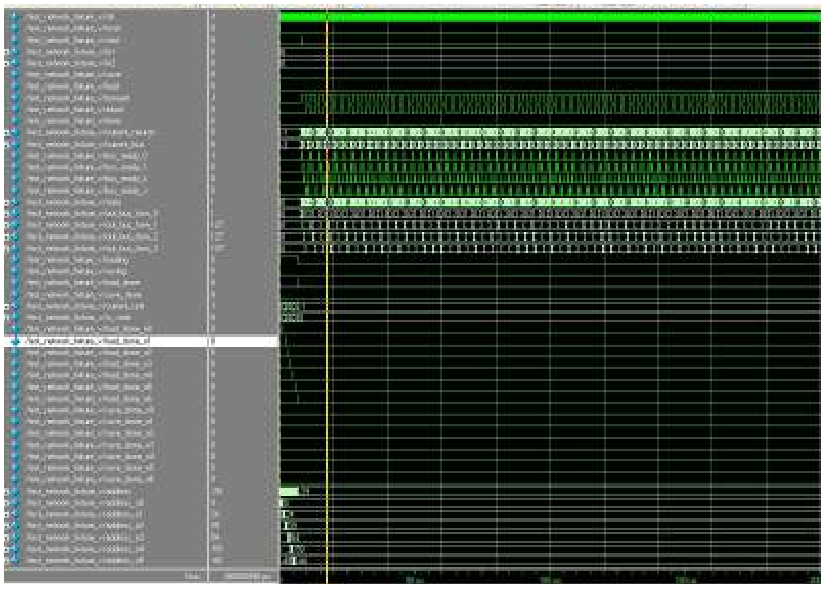


Figure 2.3: Simulation waveform of neural network being trained to learn the Boolean function AND.

This time we got an oscillatory behavior out of the system. The neural network would oscillate between output 127 and -127 which happen to be the maximum and minimum values outputted by the sigmoid modules. However, in the AND function we defined TRUE as 127 and FALSE as 0. This caused the squared output error to oscillate between 0 and a large number as the output oscillated between -127 and 127. This oscillation caused the average error to never be below the margin and for the system to converge. However, if we were to define TRUE as 127 and FALSE as -127 the neural network would have learned how to compute that.

2.2.8 Issues with Compiling the Neural Network

Satisfied with the simulations we then decided to try the test the neural network on the labkit using the labkit to computer via serial mechanism to save the weights. However, we ran into a bizarre compilation error that prevented us from compiling the Verilog and testing it on the labkit. The error that we got had to do with an illegal mux operation. It is possible that one of the multiplications in a case statement was not completing fast enough and this was causing the error. However, we were unable to track down the exact location of the bug.

2.3 Testing and Debugging Hand Detection and Serial (Behram Mistree)

2.3.1 Initial Hand Detection Testing and Debugging

We demonstrated functionality of our initial hand detection algorithm by painting squares to the screen on top of the image the camera was reading. These squares were associated with the positions that the hand detection module detected LEDs. For short distances and correct lighting conditions, the algorithm worked adequately.

Debugging was difficult with the initial hand detection module. For a while it was difficult to discern what our sources of error were. Were we not getting data because we were not associating the right horizontal and vertical positions with the right YCrCb values? Or was it because our thresholds for detecting red LEDs were unreasonable?

For a long time, there appeared to be a scaling issue with the data: although LEDs were detected, they were not painted in the right positions. This scaling issue occurred because I was using some of Javier's old code incorrectly to decode the horizontal and vertical positions of YCrCb values coming from the camera. Coupling the horizontal positions found using the old code with our own new code, bgetHorVert, which did a great job finding vertical positions solved this problem.

2.3.2 Alternate Hand Detection Testing and Debugging

Noting other groups' successes with taking weighted averages of pixel data, we implemented a second algorithm for collecting pixel data and began to test it. We used a similar approach to testing our second hand detection module to testing our first: we painted squares on top of real-time data being received by the camera. These squares corresponded to the positions that LEDs were supposed to have been detected. By comparing the placement of these squares to their real-world counterparts in the image, we tested our algorithm.

Unfortunately, our alternate hand detection algorithm did not work completely correctly. Specifically, when we interfaced the alternate hand detection module with the horizontal positions from get_hcount_vcount, the square representing our pixel would not be drawn in the correct place. We tried to solve this problem by implementing our own function, bgetHorVert, which would take outputs from the NTSC decode module and convert them into a horizontal and vertical position detected by the camera. The vertical position was almost spot on. However, the horizontal position still left much to be desired: there was a large, constant horizontal offset between the where the camera was recording the LED and where our algorithm detected the LED.

Thinking that such an offset might simply be a product of some linear shift, we tried subtracting off the constant value and retesting to see if everything was fixed. Unfortunately, when we subtracted the offset, it appeared that our square was being drawn horizontally twice as far as it should be.

We tested and retested our divider. We tested the system that painted pixels to the monitor. We worked and reworked our horizontal and vertical counters. All to no effect.

The only explanation that we can plausibly accept is that the values received from the NTSC decode did not correspond to those that we expected from reading about field line decoding from http://www-mtl.mit.edu/Courses/6.111/labkit/appnotes/xapp286_04.pdf.

At the time, we could not figure out any plausible reason why that would be the case. In retrospect, we realize that the discrepancy might have to do with the NTSC decode's *dv* (data valid) output. In calculating the horizontal pixel position of the camera, we increment for every clock cycle after the NTSC decode module's *h* output goes high. However, if the data is not valid while we are incrementing, it might explain why we were seeing the offset that we were.

In addition, taking into account the data valid signal would not affect our vertical count at all, explaining why we kept getting a poor horizontal position even while we got great vertical positions.

2.3.3 Serial Module Testing and Debugging

We tested the serial module by loading a .coe file onto a RAM with specific values. We wrote a MATLAB function that read data from the serial port. When the data read by MATLAB matched the data that the RAM was initialized to, we declared success.

Reading the RS-232 specifications, it seemed as though our module would have to perform some form of elaborate handshake just to get the computer to listen to it. For a while, we tried to implement this elaborate handshake but were unsuccessful. After re-reading the specifications though and talking with Prof. Terman, we discovered that we did not need a handshake, we just needed to spew data at the computer in a properly clocked manner.

Other issues involved sending stop bits. Although every source about RS-232 proclaimed that I needed a stop-bit, none of them actually told me whether that stop bit would be high or low. (They also did not tell me that I needed to start transmissions with a high bit.) Fortunately, I sat next to a group who was doing serial project that also involved the serial port. After exchanging information with them and displaying serial transmissions from my computer to my labkit on the logic analyzer, we were soon able to amend any errors, and use the RS-232 ports to transmit data to the computer.

3. Conclusion

Overall, the final project we choose proved to be satisfactory. It provided us with a very formidable challenge and a great opportunity to learn more about the design of complex digital systems. Even though we did not get the results we wanted, the project can still be considered a success. We accomplished many tasks and took a stab at implementing an innovative digital design.

The hand detection modules backed with the subtractive synthesis modules performed basic functionality satisfactory. The system was able to synthesize guitar and violin in response to the position of LEDs picked up by the camera. The system performed fairly well although there was room for improvement. The mechanism for sampling the stored oscillating signals was not perfect and could have been improved. However, we also wanted to try and implement a neural network which limited the amount of time we could spend perfecting the subtractive synthesis method we used.

The neural network proved to be a very formidable challenge. We came extremely close to finishing the implementation. The simulations we ran in ModelSim showed that the neural network was learning, although it was not always learning correctly. A possible way to improve the ability of the neural network to learn could be to increase the threshold of the sigmoid modules. Our implementation had a fairly small threshold considering we were using 32 bit numbers. We did notice improvements in performance when we would increase the range of the module.

Event though we didn't get to test the neural network on actual hardware I'd have to say that the overall project was a success. The project can be considered a success in the sense that we learned a great deal about designing complicated digital systems. The neural network quickly became very complicated as the individual components were integrated and the entire network was built up. During our design of the neural network we also learned a great deal about dealing with signed numbers and choosing the right bits out of a number in order to get a smaller scaled number.

At the days end we learned a great deal and had a lot of fun in the process. We got to try something new and take risk in trying to implement a complicated system. Sometimes life's greatest lessons come from the small setbacks we encounter along the way. Although we didn't finish the full implementation of the hand controlled digital audio synthesizer we were aiming for, we are now more knowledgeable about digital systems and are fully armed and confident to take on even more challenging and complicated problems head first.

4. References

1. Norvig, Peter; Stuart, Russel. Artificial Intelligence: A Modern Approach. Second Edition. 2003. Pages 736-748.

2. Higgins, Anna-Maria. Pluck…bow…strike…blow….
http://www.clubi.ie/amhiggins/adsr.html. Visited on November 6, 2006.

3. Scavone, Gary. Envelopes.
http://www.music.mcgill.ca/~gary/307/week2/envelopes.html. Visited on November 8, 2006.

4. Wikipedia.org. http://en.wikipedia.org/wiki/Subtractive_synthesis. Visited on November 7, 2006.

A. Appendix

A.1 Test_guitar2.m

```
a = .03;
d = .08;
r = .89;%.1197;%.125;
rate = 256;
decay_percent = .2;

% Do attack portion
N = round(a * rate);
p = N^2;
t = 0:N;
y = (t .^2) / p;

%N = round(a * rate);
%y = [0:N-1] / (N-1);

% Do decay portion
N = round(d * rate);
decay_target = decay_percent*y(length(y));
n = (y(length(y)) - decay_target) / N;
cpp = (y(length(y))-n):-n:decay_target;
y = [y cpp];

% Do release portion
%N = round(d*rate) + round(a*rate);
%N = round(a*rate);
%y4 = [N-1:-1:0] / N;
%p = (N - rate)^2 / (4*y(length(y)));
%t = N+1:rate;
%y4 = ((t - rate).^2) / (4*p);

N2 = round(a*rate) + round(d*rate);
N = rate - 100 - N2;
T = 1/N;
tau = 0.2;

yz = [y(length(y)) 0 ];     % y1 and y2 values
t = [N2+1 rate-100];    % t1 and t2 values
tinc = T;
aa = exp(-T/tau);

% Do iteratively.

tvals = t(1):t(2);
yvals = zeros( size(tvals) );
yvals(1) = yz(1);
tn = t(1);

for n = 2:length(tvals),
  yvals(n) = aa*yvals(n-1) + (1-aa)*yz(2);
end

y = [y yvals];

yf = zeros(1,99);
y = [y yf];

%ph = log(y(length(y))) + length(y);
%y5 = exp(t + ph);
%y = [y y4];

%N = r * rate;
%y4 = y(length(y)) * [N-1:-1:0] / N;
%y = [y y4];

highest = 2^8;
```

```
y = round(highest/2 .* y);

%%%%

depth = 128;
width = 8;

fid = fopen('guitarADSR.coe','w');

fprintf(fid,'\tcase(address[6:0])\n');


for i = 0:255
    fprintf(fid,'\t\t%1.0f: out[7:0] =  %3.0f;\n',i,y(i+1));
end

fprintf(fid,'\t\tdefault: out[7:0] = 0;\n');
fprintf(fid,'\tendcase\n');


status = fclose(fid);
```

A.2 test_violin_r2.m

```
a = .5;
s = .1;
r= .4;
%0.1239
rate = 256;

% Do attack portion
N = round(a * rate);
p = N^2;
t = 0:N-1;
y = 1 + (((t - N).^2) / -p);

% No decay portion

% Do sustain portion
N = round(s * rate);
y3 = ones(1,N);
%y3 = decay_target .* y3;
y = [y y3];

% Do release portion
N = r * rate;
N = round(s*rate) + round(a*rate);
p = (rate - N)^2;
t = N+1:rate;
y4 = 1 + (((t - N).^2) / -p);

y = [y y4];

zpwm = wavread('Subsynth-wavemix.wav');
zpwm = zpwm';

z1 = squareGen(2*pi*440,1,2*pi*44000);
z2 = squareGen(2*pi*493.88,1,2*pi*44000);
z3 = squareGen(2*pi*523.25,1,2*pi*44000);
z4 = squareGen(2*pi*587.33,1,2*pi*44000);

z = [z1 z2 z3 z4];

yy = [y y y y];



highest = 2^8;
```

```
y = round(((highest/2) -1) .* y);

depth = 256;
width = 8;

fid = fopen('violinADSR.coe','w');

fprintf(fid,'\tcase(address[7:0])\n');


for i = 0:255
    fprintf(fid,'\t\t%1.0f: out[7:0] =  %3.0f;\n',i,y(i+1));
end

fprintf(fid,'\t\tdefault: out[7:0] = 0;\n');
fprintf(fid,'\tendcase\n');


status = fclose(fid);
```