The Xtremix System

Karen Chu, Jehan deFonseka
TA: Javier Castro
6.111
May 17, 2007

Abstract: The purpose of this system is to allow users to remix music using specific hand gestures. The system takes as inputs an audio signal as well as readings from 3D accelerometers and gyroscopes. The readings from the accelerometers are translated into action codes, which are used to modify the incoming music signal before it is outputted. The system effectively distinguishes between 15 different gestures and modifies an incoming signal using a combination of loop insertion and music filters.

1 Introduction

The XtremiX system uses hand gestures as an interface to a music remixing system. The name Xtremix comes from the combination of extreme and remix, and these two words embody the main motivations behind the project. The system was designed using the Xilinx FPGA and coded in verilog.

The main features of this project are the hand gesture recognition system and the music remixing system. The gesture recognition system uses a 3D accelerometer and a gyroscope per hand, allowing the system to capture 15 different gestures, including both one-hand gestures and two-hand gestures. The audio system includes the capabilities to fully remix input audio clips by recording, layering, and filtering these clips, as well as changing the volume of playback.

In the Xtremix system, 5-bit gestures created are passed to a gesture processor. This gesture processor filters the gestures, decreasing the probability that unintended gestures are triggered. The gesture processor outputs an action codes to the audio system, which uses the code to enable or disable various audio features. See Figure 1 for a depiction of the system.

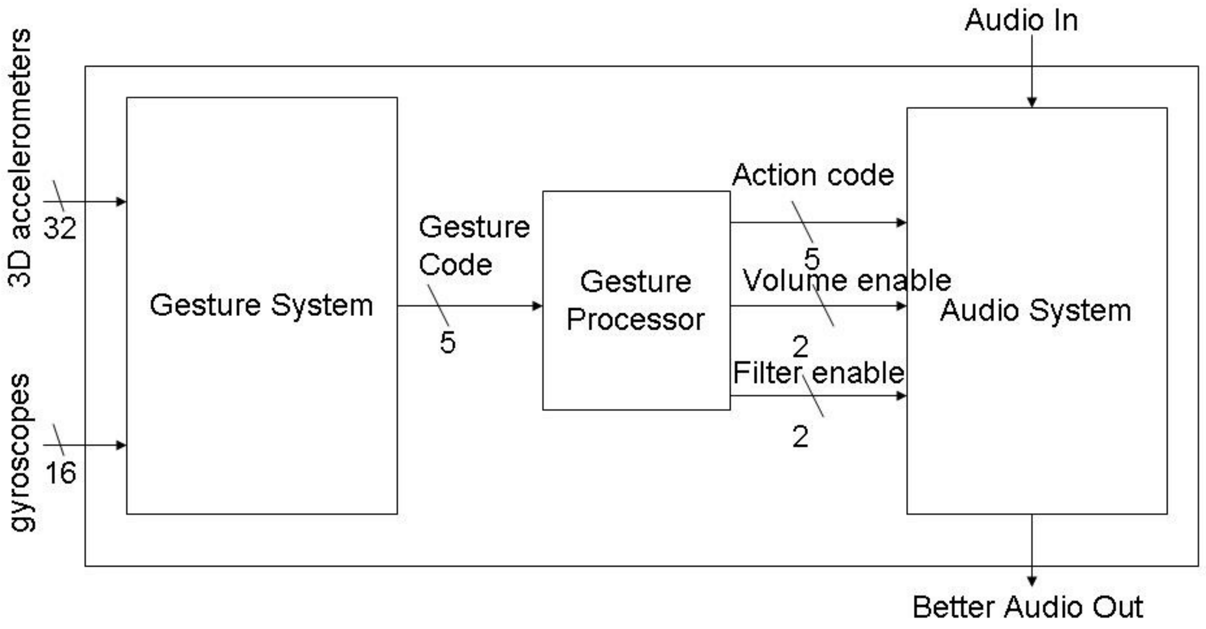

Figure 1: The Xtremix System Diagram

This project is important because people should be able to remix music easily in a fun, intuitive way. They should be able to dance with the controllers if they want to, and the result should be at the very least interesting to hear. Even a small child should be able to pick up the controllers, and, with a bit of trial error, understand what action corresponds to what change in the audio. But most of all, the project is important because it allows people to have a bit more fun.

This paper follows with description of the gesture system, followed by testing of the gesture system. The paper continues similarly with description of the audio system and testing of the gesture system.

2 The Gesture System

The gesture system comprises a gesture sensing system consisting of analog accelerometers and gyroscopes and A/D converters, and a gesture processing system implemented digitally on the labkit FPGA. The user interacts with the gesture system by holding the paddles, one in each hand with the surface of the paddles parallel to the ground. They can then proceed to move their hands independently or together from left to right and vice versa along the x-axis, up and down along the y-axis, or make twisting motions clockwise or counterclockwise in the plane parallel to the ground. The accelerometers and gyroscopes output a voltage proportional to the amount of acceleration sensed in their respective directions. These voltages are converted to 8-bit numbers by the A/D converters that are controlled by and output through the user I/O ports on the labkit. The data is then processed by modules implemented on the FPGA that infer the action made by the user and output a gesture code indicating the corresponding audio effect to the audio system.

2.1 Gesture Sensing System

The gesture sensing system consists of two accelerometers, two gyroscopes, and six A/D converters. All components used are from Analog Devices. The accelerometers used are ADXL330 3D accelerometers. The gyroscopes used are ADXRS330 single-axis accelerational gyroscopes. The sensors are mounted on two breadboard paddles, one accelerometer and one gyroscope per paddle. Each paddle has two sets of twisted wires connecting it to the labkit breadboard and power supply. One set consists of a +5.0V supply, +3.3V supply, and ground wire. The second set consists of a wire each for the accelerometer output along the x and y axes and a third wire for the gyroscope output.

Each of the six outputs from the paddles is converted to an 8-bit number by an A/D mounted on the labkit breadboard. The converters used are AD7810 8-bit parallel output chips. They are controlled by a signal indicating start of conversion and a signal indicating start read output through the labkit user ports. The busy signal and data bits output by the A/D are relayed to the gesture processing system also through the user ports.

2.2 The Gesture Processing System

The gesture processing system consists of eight modules as depicted in Figure 3: a master FSM, an acceleration decoder, a gyroscope decoder, an acceleration numerical processing unit, a gyroscope processor, a 100-Hz divider, a 1-KHz divider, and a gesture processor. Every unit receives the same 27-MHz clock signal and a synchronized reset signal. The behavior of the entire gesture system is controlled by the master FSM, which receives enable pulses from the 100-Hz divider in addition to the system clock. The master FSM in turn calls on the acceleration and gyroscope decoders to provide an acceleration code and a gyroscope code, which are then processed by the master FSM to result in a gesture code. The acceleration decoder receives input signals from the acceleration numerical processor, which captures digitized acceleration data from the A/D converters and outputs the average of eight most recent acceleration measurements. The gyroscope decoder receives input from the gyroscope processor, which captures and passes through digitized rotational acceleration data from the A/D converters. The acceleration numerical unit and the gyroscope decoder both receive enable pulses from the 1-KHz divider in addition to the system clock. After the gesture code is computed by the master FSM, it is passed to the gesture processor, which then outputs an action, volume, and filter code to the audio system, indicating the desired audio effect.

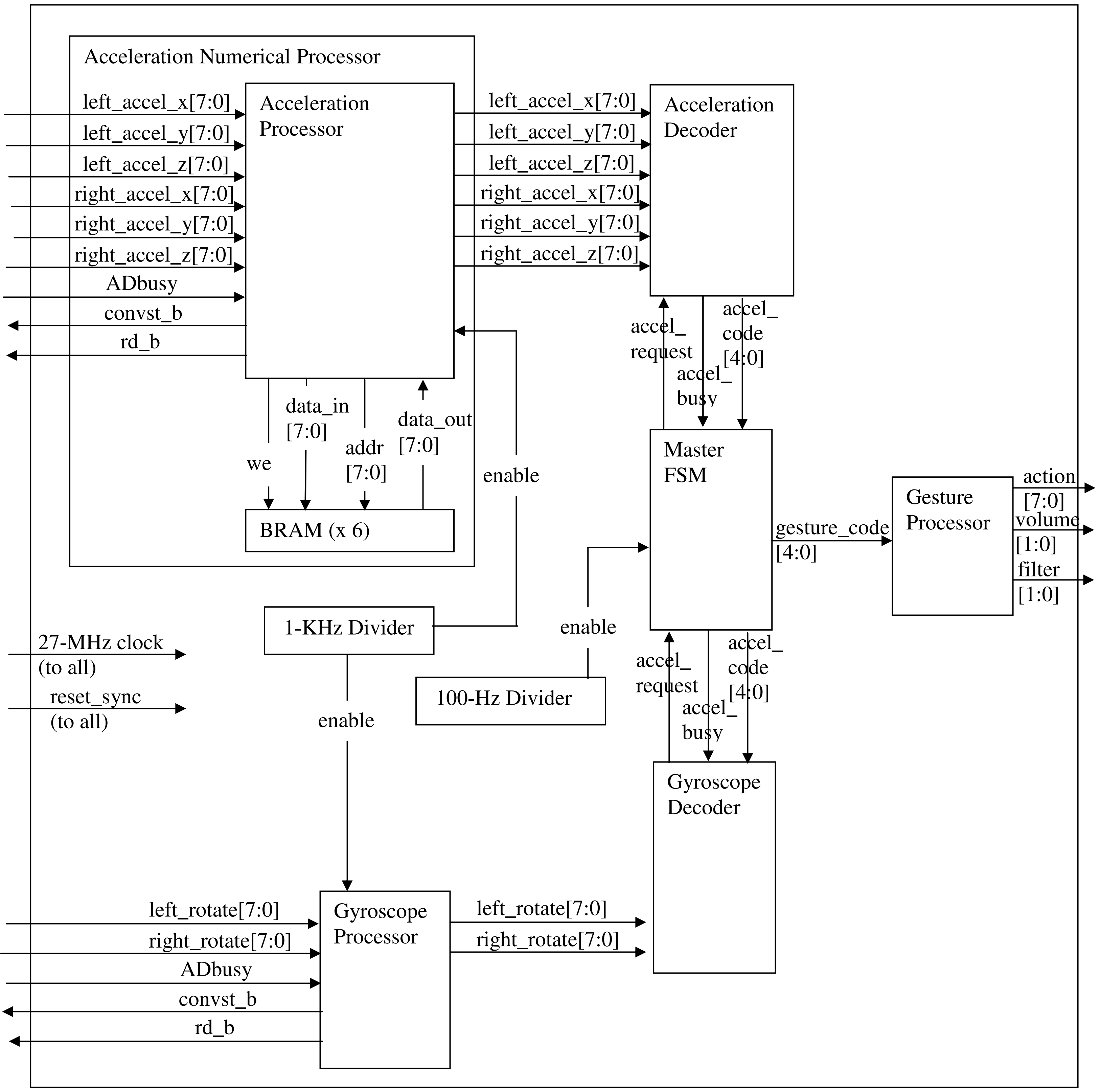

Figure 2. The gesture processing system. The system consists of seven modules coordinated by the master FSM and receiving digitized input from the accelerometers and gyroscopes.

2.2.1 Master FSM

The master FSM coordinates the interaction between other modules of the gesture system and calculates the final gesture code based on the input acceleration and gyroscope codes. In addition to the acceleration and gyroscope codes, this unit also receives a 100-Hz enable signal from a divider and busy signals from the acceleration and gyroscope decoders. To request acceleration and gyroscope codes, the master FSM pulses a request signal, activating the units responsible for calculating those codes.

The master FSM is implemented as a Mealy FSM with eight states. The FSM advances to the next state on each clock cycle. The next state is calculated as a logic combination of inputs from the 100-Hz divider, the acceleration decoder, and the gyroscope decoder. The transition diagram in Figure 3 illustrates the state changes. A particular signal is asserted if the signal is high and not asserted if the signal is low. In Figure 3, only asserted signals are listed. Unlisted signals can be assumed to be not asserted.

The initial state is the IDLE state. In this state, reset_accel and reset_gyro are asserted in order to zero the internal accel_code and gyro_code. The unit remains in the IDLE state until it receives an enable pulse from the 100-Hz divider, on which it advances to the next state, GyroRequest.

In the GyroRequest state, reset_accel and reset_gyro remain asserted to keep the internal code values at zero until they can be set to the proper value. In addition, gyro_request is asserted, activating the gyroscope decoder. The unit then proceeds directly to the GyroPause state, in which set_gyro is asserted, setting the internal gyro_code to the value received from the gyroscope decoder. reset_accel is still asserted to keep the internal accel_code at zero. The unit then moves to the GyroPause2 state, which asserts reset_accel again, and waits for the busy signal from the gyroscope decoder to go low.

The master FSM is designed such that any motion that triggers the gyroscopes will cause the acceleration input to be ignored. This is because rotational motion is used to insert overlay clips, which is an action that is more likely to be frequently used than the translational motion to adjust volume or apply filters. Thus, at this point, the unit checks to see if the internal gyro_code is now something nonzero, indicating a movement sensed by the gyroscopes. If so, the FSM advances to the Decode state. In this state, the gesture code is calculated as combinational logic depending on the internal accel_code and gyro_code. There are fifteen total different audio effects implemented in the current version of XtremiX. Vertical motion of either hand individually raises and lowers volume. Movement along the x-axis of either hand individually applies and removes a low-pass filter and high-pass filter. Rotational motion in the plane of the paddle neutral position overlays short pre-recorded sound clips on top of the bas track. A complete listing of the gesture codes and corresponding accel_code, gyro_code, and functional description can be found in the appendix. Additionally, reset_accel and reset_gyro are both asserted again to set the internal codes back to zero.

If the internal gyro_code remained zero after GyroPause2, indicating insignificant rotational motion, the FSM advances to the AccelRequest state. Analogous to the GyroRequest state, in the AccelRequest state accel_request is asserted, activating the acceleration decoder. Additionally, reset_accel and reset_gyro are asserted to keep the internal codes at zero. The FSM then enters the AccelPause state, in which set_accel is asserted, setting the internal accel_code to the value received from the acceleration decoder. reset_gyro is also asserted to keep the internal gyro_code at zero. The FSM then moves on to the AccelPause2 state in which reset_gyro is still asserted. When the busy signal from the acceleration decoder goes low again, indicating completion of calculations, the master FSM advances to the Decode state. From the Decode state, the FSM returns to the IDLE state. Upon receiving a reset signal, the FSM returns to the IDLE state regardless of its current state.

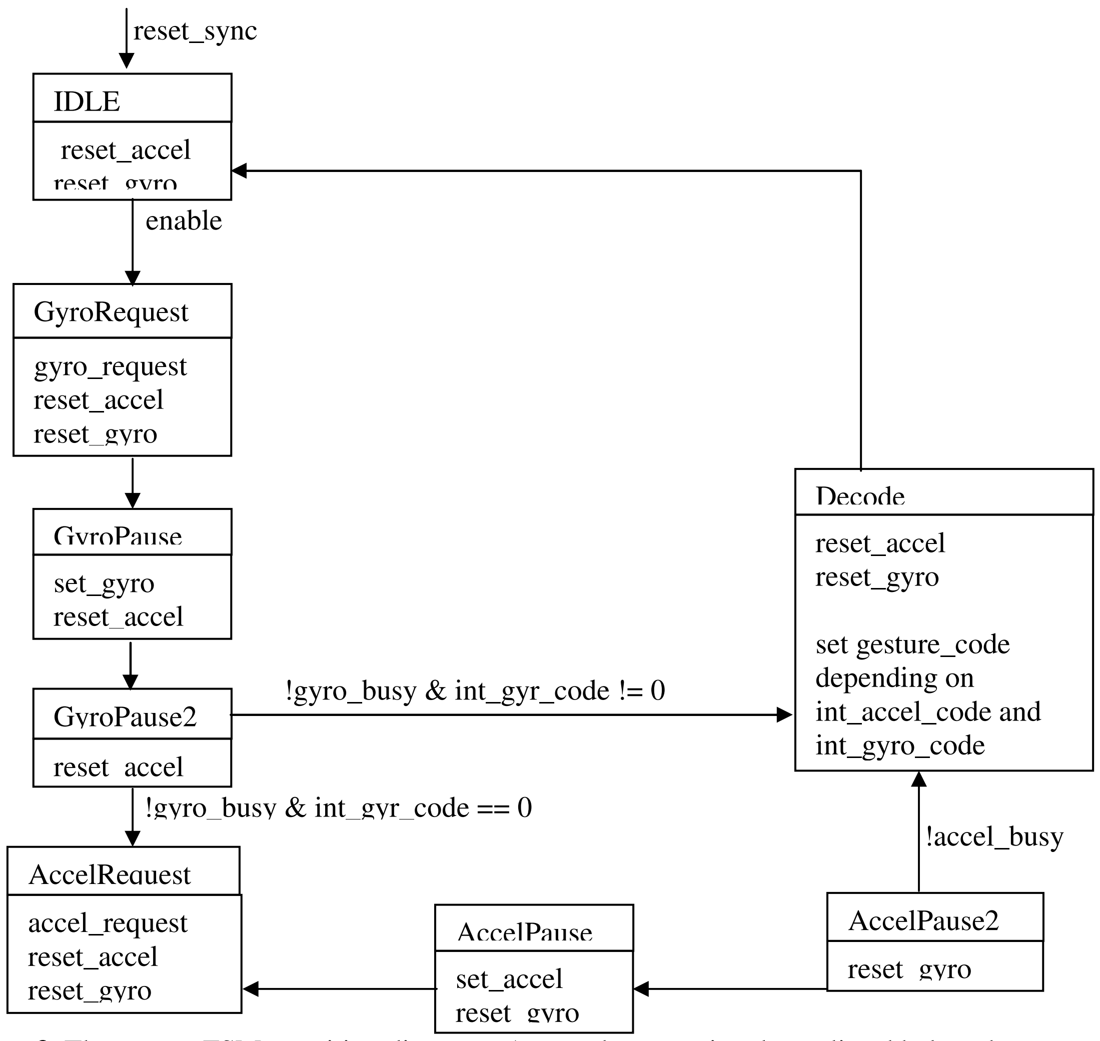

Figure 3. The master FSM transition diagram. Asserted output signals are listed below the state name. All states return to IDLE upon receiving reset_sync.

2.2.2 Acceleration Decoder

This unit receives as input six values indicating the average of the eight most recent acceleration measurements along the x, y, and z axes for each hand and calculates an acceleration code depending on the motion indicated by the acceleration values. In this implementation of XtremiX, information regarding movement along the z-axis is ignored. However, inference of motion along that axis is analogous to motion along the other two axes. The unit's operation is controlled by a request signal from the master FSM indicating when a calculation should be made.

The acceleration decoder is implemented as a two-state Mealy FSM. The FSM is initialized in the IDLE state and returns to the state after ever calculation and when reset_sync is received. In the IDLE state, no signals are asserted. When the decoder receives a request signal from the master FSM, it advances to the Decode state in which the acceleration along each axis is compared to the threshold acceleration values set in the zero_bias and min_accel parameters. For example, if the acceleration along the x-axis is larger than the sum of the zero_bias and min_accel, then the user is inferred as having moved their hand to the right. The threshold values are meant to filter out spurious local maxima caused by slight tremors or contamination from other motions. Depending on the movement inferred, the unit outputs an acceleration code corresponding to that movement. The unit currently infers 17 combinations of motion from the two paddles. A complete listing of the acceleration codes and corresponding inferred motions can be found in the Appendix. In addition to outputting a new acceleration code, the unit also asserts a busy signal during the Decode state to indicate to the master FSMs that the calculation process has not yet been completed.

2.2.3 Gyroscope Decoder

This unit receives as input two values indicating the most recent rotational acceleration measurements around the vertical axis for each hand and calculates a gyroscope code depending on the motion indicated by the acceleration values. This unit is controlled by a request signal from the master FSM indicating when a new calculation is needed.

The acceleration decoder is implemented as a two-state Mealy FSM. The FSM is initialized in the IDLE state and returns to the state after every calculation and when reset_sync is received. In the IDLE state, no signals are asserted. When the decoder receives a request signal from the master FSM, it advances to the Decode state in which the rotational acceleration is compared to the threshold acceleration values set in the zero_bias and min_rotate parameters. For example, if the acceleration is larger than the sum of the zero_bias and min_rotate, then the user is inferred as having rotated their hand in a clockwise direction. Alternately, an acceleration less than the zero_bias with min_rotate subtracted indicates that the user rotated their hand in a counterclockwise direction. The threshold values are meant to filter out spurious local maxima caused by slight tremors or contamination from other motions. Depending on the movement inferred, the unit outputs a gyroscope code corresponding to that movement. The unit currently infers 9 combinations of motion from the two paddles. A complete listing of the the gyroscope codes and corresponding inferred motions can be found in the Appendix.

2.2.4 Acceleration Numerical Processor

This unit receives as input six values corresponding to the raw digitized data from the accelerometers indicating acceleration along the x, y, and z axes. It then calculates a moving average of the eight most recent measurements along each axis for each hand and outputs the results to the acceleration decoder.

The acceleration numerical processor consists of seven subunits: six block RAMs, one for each axis of each hand, storing the eight most recent acceleration measurements, and an acceleration processor, which interfaces with the A/D to read the acceleration value, controls the BRAMs, and calculates the moving average.

2.2.4.1 BRAMs

The BRAMs are six instantiations of the accel_mem core. These memory blocks are set in read-before-write mode, so they output the old value in a given memory slot when that slot is being written to. This is useful, because the summing scheme used requires being able to subtract the old value in a memory slot and add the new value being written. This mode allows the acceleration processor to access the previous value in the slot without performing an explicit read.

2.2.4.2 Acceleration Processor

This unit performs several vital functions: interfaces with the A/D to read data, controls the BRAMs, and calculates the moving average. To interface with the A/D, the unit must send an active low signal indicating the start of a conversion. Once a conversion has begun, the A/D outputs a high busy signal until the conversion is finished, at which point the data can be read. The unit then sends an active low read signal to the A/D in order to read the converted data. Because of the timing requirements on the A/D, there must be sufficient time between the previous read signal and the next conversion process.

Thus, there needs to be several clock cycles of waiting. In this waiting time, this unit writes the newest accelerations to the BRAMs and calculates the average.

To control the BRAMs, the acceleration processor contains a pointer field that increments every new A/D conversion cycle. Because only the eight most recent values are desired, the pointer is a 3-bit number which, allowing it to roll over automatically when it has cycled through all used addresses in the BRAMs. The moving pointer indicates the address the data should be written to, and the data is the most recent data from the A/D. After writing the new data to the BRAMs, the acceleration processor must read the values of the data previously stored in that memory slot, and then calculates a new set of sums. Finally the acceleration processor calculates the values to be output to the acceleration decoder.

In order to coordinate the interface with the A/D and the additional processes performed during the wait time required by the A/D, the acceleration processor is implemented as a Mealy FSM with ten states. On each clock cycle, the FSM advances to the next state, which is calculated as a logic combination of inputs from the 1-KHz divider, reset signal, and a counter used in the Reset state. The transition diagram in Figure 4 illustrates the state changes. A particular signal is asserted if the signal is high and not asserted if the signal is low. In Figure 4, only asserted signals are listed. Unlisted signals can be assumed to be not asserted.

The FSM initializes to the IDLE state, in which the convst_b and rd_b signals are asserted because the conversion start and read signals to the A/D are active low. It advances to the ADActivate state when it receives an enable pulse from the 1-KHz divider. In this state, the convst_b signal is unasserted, indicating to the A/D that a new conversion process should be started. The rd_b signal remains asserted. The FSM then proceeds directly to the Pause state in which both convst_b and rd_b are asserted. It remains in this state until the busy signal from the A/D is low, indicating the completion of conversion. The FSM then enters the ADRead state in which the rd_b signal is unasserted, indicating a request for data from the A/D. The convst_b signal remains asserted. The next state is the ADRead1 state in which rd_b remains unasserted. Additionally, the cap_val indicator is asserted, which captures the A/D output bits to an internally stored value. The FSM then advances to the ADRead2 state in which convst_b and rd_b are both asserted, and the set_data and set_addr indicators are asserted, setting up the data to be written to the BRAMs and the address to which to write the data. The FSM proceeds then to the MemActivate state in which the A/D control signals remain asserted, and the we signal is asserted indicating a write enable on the BRAMs. The next state is the Math state, in which the A/D control signals remain asserted. Additionally, the cap_read indicator is asserted, capturing the values output by the BRAMs to internally stored values in the acceleration processor. The FSM then proceeds to the Output state in which the set_sum and set_out indicators are asserted. Thus, the new moving sums are calculated by adding the values captured from the A/D to the current sums and subtracting the values captured from the BRAM outputs. The output is then calculated by right-shifting the sums by three, effectively dividing the sums by eight. The A/D control signals remain asserted in this state as well. After performing these calculations, the FSM then proceeds back to the IDLE state.

On receiving a reset_sync signal, the FSM enters the Reset state in which reset_data, reset_sum, and we are asserted. This causes a default zero-bias value to be written to every slot in the BRAMs, and resets the sums to a default value corresponding to all eight slots being the default value. The A/D control signals are asserted in this state as well. Rather than use the pointer as the address, a separate counter used only in the Reset state is used. When the counter has counted past eight, the reset is complete and the FSM advances to the IDLE state.

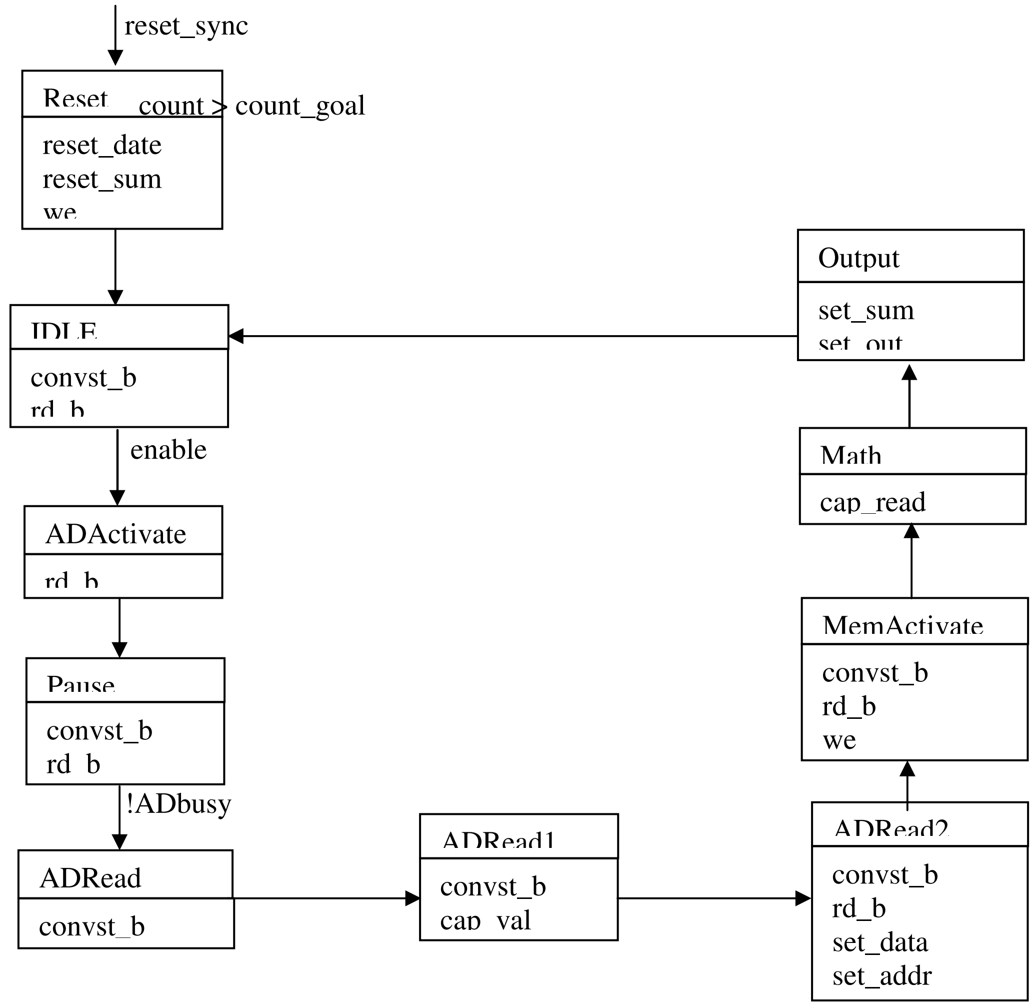

Figure 4. The acceleration processor transition diagram. The FSM exits the IDLE state when an enable pulse is received and proceeds straight through the other states.

2.2.5 Gyroscope Processor

This unit receives as input two values corresponding to the digitized data from the gyroscopes indicating acceleration around the vertical axis. It then passes this value to the gyroscope decoder. In order to read data from the A/D, the gyroscope processor must interface with the A/D as described for the acceleration processor.

The gyroscope processor is implemented as a Mealy FSM with eight states, most of which are wait states residual from a previous version in which numerical integration was performed to find the absolute position of the gyroscopes. On each clock cycle, the FSM advances to the next state, which is calculated as a logic combination of inputs from the 1-KHz divider, and reset signal. The transitions are exactly as shown for the acceleration processor in Figure 4, with two residual wait states, Convert and Add, in lieu of the ADRead2, MemActivate, and Math states.

The FSM is initialized in the IDLE state, and proceeds through the IDLE, ADActivate, Pause, ADRead, and ADRead1 states as described for the acceleration processor. It then proceeds through the residual Convert and Add states. The FSM then advances to the Output state in which set_out is asserted, setting the output of the gyroscope processor to the internally stored value read from the A/D. The FSM then returns to the IDLE state. On reset_sync, the FSM returns to the IDLE state regardless of the current state.

2.2.6 Dividers

There are two dividers, a 100-Hz divider and a 1-KHz divider used in the gesture processing system. The two are identical except for the value up to which the divider counts before emitting an enable pulse.

The divider modules take the 27-MHz system clock and divides it to output a signal that remains high for one clock cycle at rates of either 100-Hz or 1-KHz. The modules keep track of a current count, which is compared to the goal, in this case 270000 and 27000, respectively, every clock cycle. If the goal has been reached, the modules output a high signal and reset the count to zero. If the goal has not been reached, the modules output a low signal and increment the count. Thus, the high output is reset to low on the next clock, so the output consists of high signals lasting only one clock cycle. If the modules receive a reset signal, they output a low signal and set the current count to zero.

2.2.7 Gesture Processor

This unit takes as input the gesture code from the master FSM and calculates an action code, as well as a volume and filter code based on the gesture code received. These codes correspond to appropriate inputs to the audio system indicating the audio effect related to the gesture performed by the user.

2.3 Testing and Debugging

Preliminary testing and debugging was performed using ModelSim behavioral simulations. However, the limitations of behavioral simulations became apparent in debugging the acceleration numerical processing unit. Thus, further testing was performed on the acceleration and gyroscope processor units as well as on the system as a whole using post-place and route simulations in ModelSim. Although these simulations take longer than the behavioral simulations, the time saved from having to generate a new programming file for each change is still significant. Additionally, because the simulations are controlled directly through test bench code, the input signals were more convenient to control and vary. Once the debugging was moved to the FPGA and labkit, the logic analyzer became the main tool.

For each of the modules, a test bench was created with stimulus meant to evoke interesting behaviors. The master FSM and the decoder FSMs were tested together in an fsm testing module to ensure correct behavior of all the vital control signals. The acceleration numerical processor and the gyroscope processor were tested individually using both behavioral and post-place and route simulations, as well as actual test units implemented on the FPGA and debugged with the logic analyzer before being connected with the master FSM and decoders, thus ensuring their operation. The functionality of the entire system was then simulated in a system test module using post-place and route before being implemented on the FPGA for final testing. To facilitate observation of output gesture codes in FPGA implementation, the LEDs on the labkit were programmed to indicate the gesture code being output by the system. This allowed direct observation of the current gesture code.

3 Audio System Overview

The audio system has the functionality to record multiple clips of audio to the ZBT ram, to play back those audio clips, possibly simultaneously, and to filter the outgoing audio. The system also has the ability to record a sequence of user actions and repeat them on command using ‘macro mode’ commands. In addition, a playing macro can be itself recorded and added to, enabling convenient user multi-tracking.

The system uses 18-bit per channel audio sampled at 24000 Hz. The system uses 2 channels, for a total of 36-bits of signal per sample. Digital sound signals flow through the system as follows. Digital audio signals from the input source flow into the AC97 controller and into a signal combiner, which adds the input source to whatever is playing from each master ZBTRAM controller. The signal combiner supports adding four simultaneous playbacks from each master ZBTRAM controller, giving the user the ability to play 9 clips simultaneously (4 from each master ZBTRAM controller, and 1 from the input source). In addition, each ZBTRAM can store 4 different audio clips within 4 recording slots: two can hold a clip with maximum duration of 2.5 seconds, one can hold a clip with maximum duration of 5 seconds, and one can hold a clip with maximum duration of 10 seconds. Recording slots can be selected for record or playback using the switches and buttons on the labkit.

Once the signal combiner is complete, the data is passed into a 4-point high pass filter, which calculates a z-transform of the data. If the filter is enabled, it outputs the result of the z-transform. If not, the filter just passes its input as its output through to a low pass filter. The low pass filter behaves similarly, and outputs the final audio to both the AC97 controller as well as the master ZBTRAM controller. The AC97 controller outputs the audio as headphone output. If recording, the ZBTRAM will record the audio into the predefined recording slot.

The audio system uses a master controller to take inputs from the labkit buttons and switches as well as certain outputs from the AC97. This master controller sends enable signals or action commands to intermediate 'sync' modules. These 'sync' modules take as input both gesture processor commands and master controller commands, and output appropriate action or enable signals to the other modules. This allows a user to choose his or her interface. A diagram of the Audio System can be seen in Figure 5.

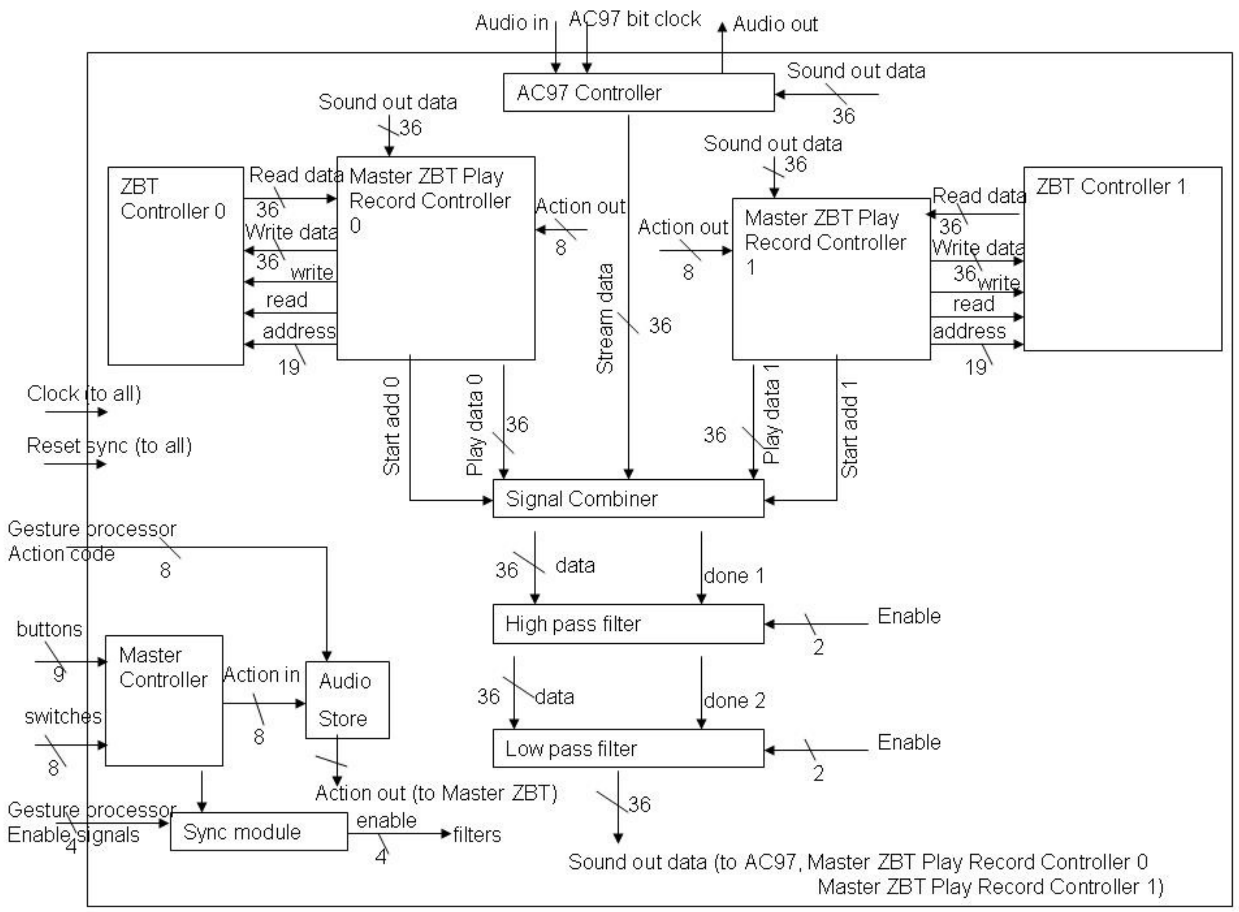

Figure 5: Audio System Block Diagram

3.1 Interface

As described in the Audio System Overview section, the audio modules take as user input both the buttons and switches of the labkit as well as action codes from the gesture processor. Each action code is associated with a particular combination of switches and a button action.

The buttons are as follows. The up and down buttons are used to record and stop recording macros, respectively. The right and left buttons are used to playback and stop playback of macros. The enter button is used for three purposes. If switch 1 is high, pressing it increases the volume of the AC97 controller. If switch 0 is high, pressing it similarly decreases the volume. If both switch 0 and 1 are low, if switch 2 is high, the high pass filter is toggled. If all three are low, the low pass filter is toggled. Button 0 acts as a global reset. Button 1 is a play button, and uses in conjunction switch 7, 6, and 5 to select the ZBTRam record slot to play and switch 4 and 3 to select the playback slot to play from. Button 2 causes the system to start recording, using switches 7, 6, and 5 similarly to select a record slot. Button 3 stops the recording.

3.2 Modules Description

The modules, seen in Figure 5 operate as follows. All modules are run off a clock synchronized with the 27Mhz onboard FPGA clock as well as the clocks to the ZBTRAMs. The ramclock module, created by Nathan Ickes, uses a phase lock loop to synchronize the ZBT clocks with the FPGA logic clock. Thus, from now onwards, this synchronized clock with be known solely as clock. This clock goes to all modules. In addition, every module accepts a reset assertion to restore initial conditions.

3.2.1 AC97 controller

The AC97 controller, designed in part by Nathan Ickes but heavily modified, uses the clock to hold a reset signal for a second to initialize the AC97. Once initialized, the AC97 outputs a bit 12Mhz bit clock, which is used to control all subsequent. Each AC97 audio sample, occurring at 48000 Hz, consists of 256 bits, 36 of which are relevant audio bits. The controller issues a sync pulse to the AC97 to sync the AC97 bit number with the internal module bit number. It then issues a series of commands to the AC97 to enable microphone recording, set the master volume, unmute the headphones, and set the PCM volume. The module then pipes a 36-bit registered digital audio input to the AC97 serially, while simultaneously latching the AC97's 36-bit input, which is streamed to the module serially as well.

The module also checks for an increase volume or decrease volume assertion once a sample. It then increases or decreases the master volume accordingly. The master volume has 5-bit precision, providing us with 32 steps for volume.

The module also issues a sound done pulse after is has recorded a new 36-bit sample. This pulse is used to start all of the other FSMs in the system. It issues this pulse every 2 samples, effectively down-sampling the AC97 to 24000Mhz.

3.2.2 Master Controller

The master controller takes inputs from the user and the AC97 and the user, and issues actions commands to the various modules. It receives a 'sound done' pulse from the AC97 and synchronizes it with the clock and registers. It addition, it receives play, record, and record stop requests from the user, with it also registers. When the master controller receives a transmit signal, indicating that the Master ZBT Play Record controllers are idle, it pulses the requests to them, and then issues the sound done pulse. The sound done pulse acts as an enable signal for the controllers. Since there are two Master ZBT Play Record controllers, one is enabled first, and then the other is enabled twenty cycles later.

3.2.3 ZBT controller

The ZBT takes as input read and write requests, an address, read data, and write data. The ZBT controller holds in its idle state until it receives a read or write request. If it receives a writes request, it outputs a busy signal and then issues write enable and the input address to the ZBTRAM. Two cycles later, it issues the write data to the ZBT. It then returns to the idle state and stops issuing a busy signal.

On a read request, the controller makes use of the pipelined aspect of the ZBT. It issues the input address to the RAM and three additional addresses on the following three clock cycles. Two cycles after issuing the first address, it latches and outputs the first of four data points, and continues latching data for the next 3 cycles. It then returns to the idle state and stops issuing a busy signal.

3.2.4 Master ZBT Play Record Controller

The master ZBT play-record controller accepts record requests, record stop requests, and play requests. Every enable pulse, the ZBT checks to see if it has a play request. If it does, it outputs the appropriate start play address to the ZBT controller depending on which record slot is selected. It then waits three cycles and then reads the data from that address. The first address of a record slot contains the final address of the record slot, further described below. The final address is registered, the playback slot which the clip will play back from is set to 'playing', and the system moves into play mode.

The module issues a read request to the ZBT, outputting and incrementing the current address tied to each playback slot in sequence. The address of each 'playing' playback slot is simultaneously checked to see if it is past the registered final address of the record slot. If it is, the playing register for that playback slot is set to '0'. The module then captures read data from the ZBT controller, and passes it onto the signal combiner for each 'playing' playback slot. It issues a 'start add' pulse to the signal combiner so that the signal combiner knows when to start adding its inputs.

The module then enters record mode. It waits for a 'start record' pulse, and then it checks to see if there is a new record request. If there is, it sets the record address to one higher that the starting address of the record slot, and issues a write request to the ZBT controller along with the record address and the record data. Each enable pulse to this module while recording, the record address is incremented and written to. If there is a record stop request, or the slot has reached its own boundary, the current record address is written to the first address of the record slot. See a depiction of the state transitions in Figure 6.

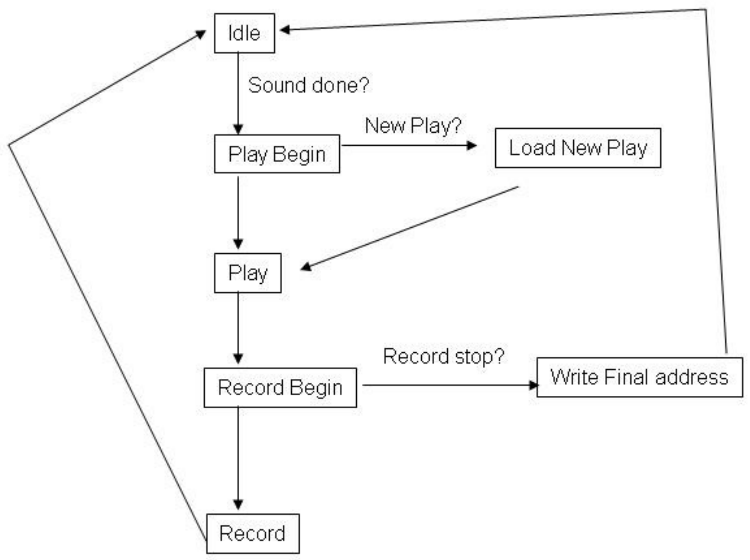

Figure 6: State Transition Diagram for Master ZBT Play Record Controller

3.2.5 The Signal Combiner

The signal combiner waits for a sound done pulse, at which point it latches the incoming stream data from the AC97 controller. It then waits for a 'start add' pulse from each of the master ZBT Play Record controllers, at which point it accumulates the data points, separately for left and right channels. It outputs its data when completed, and it issues a combiner done pulse.

3.2.6 Filter

Each filter is a four point z-transform filter, done independently for left and right channels. The coefficients are passed into the filter as parameters. The coefficients are multiplied by 2^12 and then rounded to the nearest integer. This is so floating point calculations can be avoided. The result of the z-transform is then shifted 12 bits right. In the module, upon receiving an advance pulse, it calculates the numerator, multiplying numerator coefficients with current and previous input data. It then calculates the denominator, multiplying denominator coefficients with current and previous result data.

Since the multiply operation is pipelined, the module accumulates resultant multiply data from the numerator operations and then subtracts resultant multiply data from the denominator operations as the data is ready. The result is then shifted 12 bits and then the low 18 bits of the result are outputted for each channel, along with a filter done pulse, which is either used to activate the next filter, or, for the last filter, to tell the Master ZBT Play-Record modules to start recording output data. The input and output data registers are shifted, and the module returns to an idle state.

3.2.7 Macro listener

The macro listener module handles recording and playing back of macros. It holds state of the actions requested by the macro controller as well as the gesture processor. If that state changes, the change is passed on to the appropriate modules. If recording that change, the state, as well as time since the last change in hundredths of seconds, is recorded onto the recording block ram and the record address for the recording block ram is incremented. On a record stop action, the address is filled with ones. Then, the recording block ram is copied over to the playing block ram one address at a time until the stop code (all ones) is reached.. On a play macro request, the first line of the playing block ram is read. The module then counts the required number of cycles before issuing the corresponding action to the other modules. The next addess is then loaded until the stop code is loaded. See Figure 7 for a state transition diagram of the macro listener.

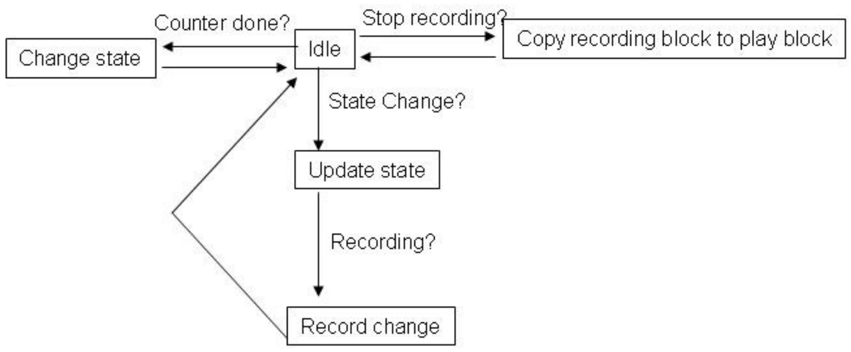

Figure 7: State Transition Diagram for Macro listener

3.2.8 Volume Sync

The volume sync module accepts volume enable pulses from both the master controller as well as the gesture processor. It registers the pulse, and waits for a sound done pulse from the master controller.

When received, the module outputs the volume enable pulse, and holds it until next sound done pulse. This occurs to avoid synchronization issues between the AC97 bit clock and the FPGA clock.

3.3 Testing and Debugging

The first module designed was the AC97 controller. It started with just having analog loopback functionality. Then, digital loopback functionality was added. Upon testing, it was pretty easy to tell that the output was shifted by one bit, because the input bits were being latched one bit too early. This bug was easy to fix. The controller was then updated to pass its input out and accept a 36-bit signal to output.

The next module designed was the ZBT controller. After a simple counter test to see if the controller was working, the module was working most of the time, but the output was glitching. Since the ZBT was off chip, timing delay due to the distance between the chip and the RAM was causing synchronization issues. Luckily, Nathan Ickes had already developed a ramclock module, which used a 2-inch segment of PCB to simulate the distance to the RAM to create a phase lock loop, so that the FPGA clock could be synchronized with the RAM clock.

The next module designed was the master ZBT controller. The main issue with this module was that it was taking an input from the AC97 controller in order to start its sequence of actions, but was not synchronizing the input. The input from the AC97 was then passed through the master controller and synchronized before being sent to the master ZBT controller, alleviating the issue.

The master controller was next created next to accept user inputs. As its only purpose was as an intermediary for existing signals, it was not a problem to design.

The next module created was the filter module. Coefficients were calculated in matlab, and the structure of the module turned out to be quite easy to create. The only filter issue was that the output was being stored before being bit shifted, leading to way too much gain. This issue was remedied easily.

The macro listener module proved interesting to design to allow users to record playing macros. Using two RAMs is a fairly simple solution, and the block rams are fast enough that the user will not notice a performance lag as the recording block is copied over to the playing block, even if the block ram is full.

4 Conclusion

The creation of the XtremiX system was an instructional and highly fulfilling experience. The process of following through the implementation of a complex system from design to completion required the application of all the design, testing, and debugging skills accumulated throughout the semester. Additionally, many new things were learned in the process of making this project, including additional design practices and debugging tricks. The final result is a fully functional remixing system that is easy to interact with, and fun to play with. In addition it offers many paths for expansion and improvement in the future.