Audio Processing Unit
EE 552 Project
Final Report

March 31, 2003

Duncan Campbell - duncanc@ualberta.ca
Grant Cunningham -grantc@ualberta.ca
Clint Lozinsky - lozinsky@ualberta.ca
Richard Schultz rschultz@ee.ualberta.ca

Declaration of Original Content

The design elements of this project and report are entirely the original work of the authors and have not been submitted for credit in any other course except as follows:

-Stereo Codec driver code and associated interface components supplied by [9].
-SDRAM controller driver and memory access taken from [9].
-LCD interface based on code taken from [11].
-Audio Effects Algorithms and implementation hints from [12].
-Various building blocks and support modules generated by the Xilinx Core Generator Tools.

_____	_____
Duncan Campbell	Grant Cunningham
_____	_____
Clint Lozinsky	Richard Schultz

Abstract

The Audio Processing Unit (APU) is a device that uses an FPGA to do digital signal processing of an audio signal. Developed on the Xess XSA-100 and Xstend board combination, it is capable of sampling an analog input signal, processing it digitally and storing it, and transferring the digital signal back into an analog audio signal suitable for playback. The APU uses an external LCD and pushbuttons to interact with the user for selection of the various operations associated with it. The main goal of the project is to develop an FPGA based device that could provide real-time audio effects including Chorus, Flange, Echo, Delay, and Pitch Shifting. Included in this was the goal to provide some facilities for storage of an audio signal. This report outlines the various aspects concerning the design, implementation and testing of the APU.

Table of Contents

Achievements

Throughout the development of this project, numerous achievements were made. Most notably, the successful implementation of a significant digital project in an FPGA was realized. Many valuable achievements were made in the areas of effective group and project management. In addition to this, several more specific achievements were made relating specifically to the various design components of the APU.

User Interface

The user interface was implemented with an LCD controller (lcd_top_level.vhd) module and a user interface controller (uicontroller.vhd). The LCD controller was originally implemented with each character stored in a logic block, but this was found to be very slow to implement and simulate. Due to this each message was stored in ROM that was implemented using the block RAM. The LCD controller, after configuring the display, then retrieved the appropriate message and sent it to the LCD. This worked well and the overall speed of implementation was increased drastically, chip space was also conserved as the block RAM was now being used instead of the logic blocks. The uicontroller was a module that de-bounced the inputs from the push-buttons and kept track of what state the system was in, this state was then sent to the other components via a system wide bus.

Memory/Storage

The storage unit (storage.vhd) implements an interface between the Codec interface and the SDRAM controller. The storage unit is able to store the audio data from the Codec interface into SDRAM while simultaneously playing audio out to the user. The data stored in SDRAM can then be played back at the user's request. The storage unit is able to store up to 170 seconds worth of audio to SDRAM.

Effects Engine

Numerous achievements were made in terms of the effects engine. To date, the circular buffer has been fully tested and works as expected. An echo has been added to the input signal, and this has been tested completely and functions correctly. As of this writing, the Chorus and Flange units are functioning, but need minor tweaking and debugging to produce audibly pleasing results. It is fully expected that this will be complete in time for demonstration and presentation purposes. As well, complete implementation with the storage module has been performed, and it has been verified that both modules will fit into the target device.

Fast Fourier Transform

A control unit, FFTcontroller, has been created to coordinate the downloading and uploading of data from the codec to RAM, and the use of the FFT with RAM. Another subordinate controller, sel_controller, has been created to coordinate the operation of the FFT butterfly algorithm. An arithmetic unit and lookup table IMMULT2 and WNLUT have been created to facilitate all the required arithmetic to perform FFT and inverse FFT operations. All that remains to bring the FFT module to completion is some debugging in the interaction between RAM and the arithmetic units.

Codec Interface

The codec driver supplied by Xess [9] was successfully modified and integrated into the design of the APU.

Introduction

The APU is able to accept an audio input signal, interact with a user to provide operations on the data, and output a modified audio signal. The operation of the entire unit is outlined below.

First, the APU samples audio input to it at 48kHz with 20-bit precision. This is accomplished by means of a stereo codec chip included on the Xstend board. This codec performs both digital to analog and analog to digital conversion. Once converted to a digital medium, the signal can then be processed as desired.

To determine what operation to perform, the APU has a user-friendly interface to the world. For this particular application, an LCD display is used to display messages and pushbuttons to select different options. What this allows is selection of a number of different parameters, which translate into specific operations. Each of these operations is performed by one of the three datapath components.

The first component of this datapath is the FFT module. Its basic operation is to provide a conversion from the time-domain signal to a frequency-domain signal, and allows processing of the signal in this domain. A frequency shifter follows the FFT section to provide pitch shifting of the input voice signal in real-time.

The effects engine is the next component of the data path. As opposed to the FFT module operating on the signal in the frequency domain, the effects engine operates completely in the time domain. Data incoming from the codec is placed into memory in a circular buffer. The data is then read out with different samples added together to provide a modified signal. The architecture of the effects engine allows it perform echo, chorus, flange and phaser effects on the signal.

The storage component is the final component of the datapath. Its basic purpose is to perform one of two operations. First, it stores a certain amount of input data as determined by either the user or the available memory size. Following this, it plays that stored data back. Effectively this creates an audio record and playback device, which is useful in many applications.

Using the above basic operation, the APU is a complete audio processor with options similar to many commercially available models.

Description of Design and Operation

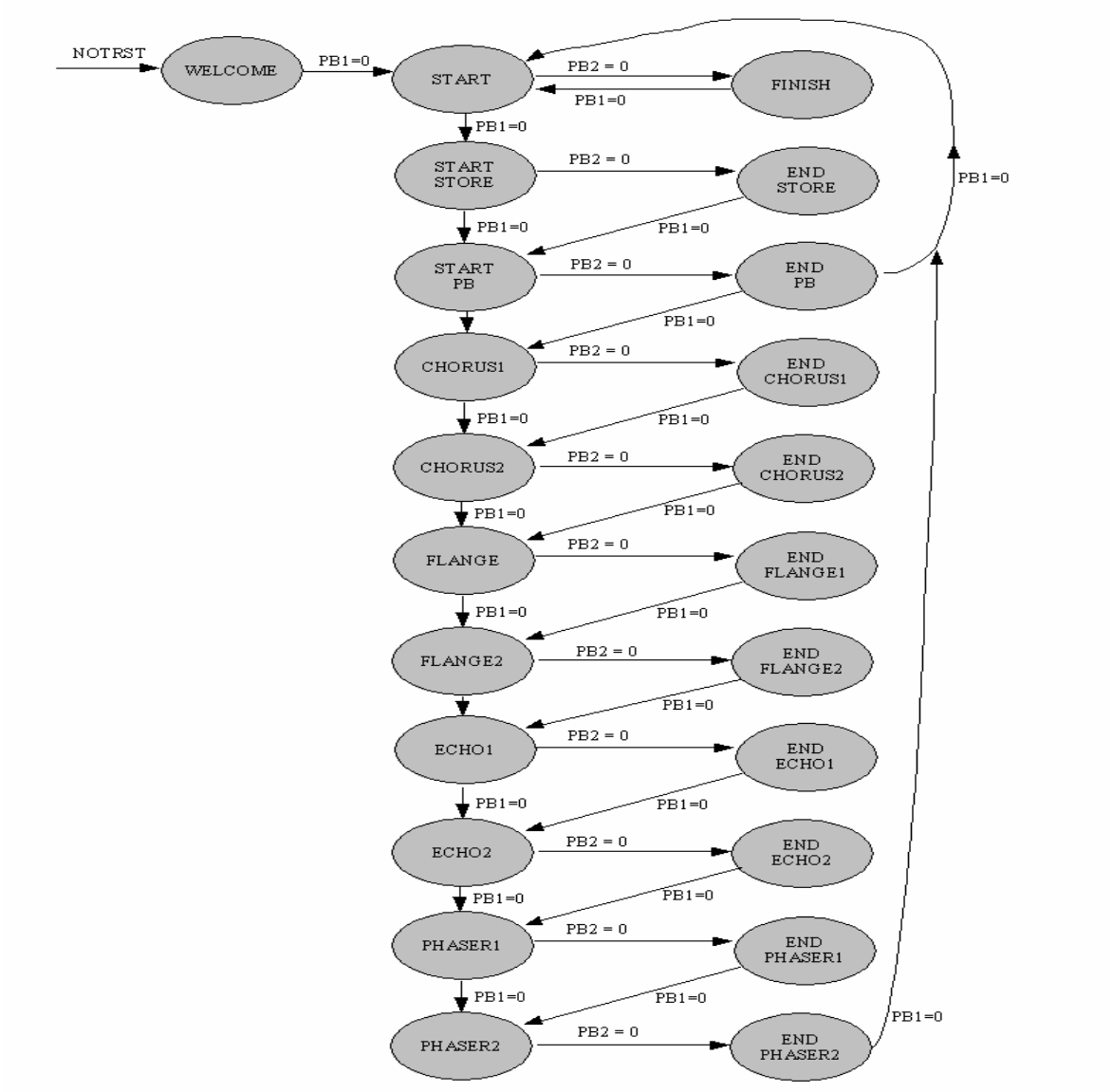
User Interface

The user interface controls what state the system is in by taking inputs from the two push-buttons and outputting the current state on the ui_bus, by using the uicontroller module. The entire system monitors this bus to ensure that there are no conflicts caused by two processes running at the same time. The uicontroller has two other inputs, pb_overrun and *full*, that also cause the state to be changed. These inputs are passed from the module that controls audio storage to the SRAM and notify the user interface if the RAM is full or if there are no more valid addresses to be read.

The LCD display is controlled by the LCD controller which consists of two main modules, lcd_decoder and lcd_driver, as well as a clock divider and top level to connect all of the internal modules. The driver module initializes the LCD upon reset and also ensures that the proper delays are obeyed between commands sent to the LCD. These delays are crucial since the driver does not check the busy flag and therefore must ensure that the LCD has had enough time to complete the current instruction before the next command is sent. The decoder monitors the ui_bus and ensures that the proper message is displayed on the LCD. If the ui_bus changes state then the decoder sends a clear signal to the driver which causes the driver to initialize the LCD and prepare to receive characters. When the driver is ready to display a character it notifies the decoder by setting the dvr_ready signal high. The decoder then reads the next ASCII character to be displayed from the ROM and sends it to the driver. Once the driver has received the signal it lowers dvr_ready and prints the character on the LCD. This process is repeated until the entire message is displayed on the screen, the cursor is then returned to the home position, by the decoder sending the home signal to the driver, and both modules then wait until the state on the ui_bus changes and a new message must be displayed.

UI Controller

The uicontroller module uses push button user inputs to control the operation of the Audio Processing Unit. The uicontroller is made up of a finite state machine. The finite state machine is controlled by the three user push buttons. Push button one (PB1) is used as a enter command, and push button two (PB2) is as a scroll command. The third push buttons acts a system wide active low reset. These user push buttons are debounced in the uicontroller. The ui_bus is used to control the operation of the entire Audio Processing Unit. The commands placed on the ui_bus tell the rest of the unit what operations to perform. The finite state machine of the uicontroller can be seen in figure 1.



Memory/Storage

The storage module provides an interface between the Codec interface and SDRAM Controller. The storage module takes in audio data from the Codec interface and stores it in SDRAM. The audio data stored in SDRAM can then be played back at the users' request. The storage unit can hold up to 170 seconds of audio. The storage unit only reads data from the left channel of SDRAM in order to save memory space, and because a mono input is inputted to the stereo codec, so both left and right channels will be identical.

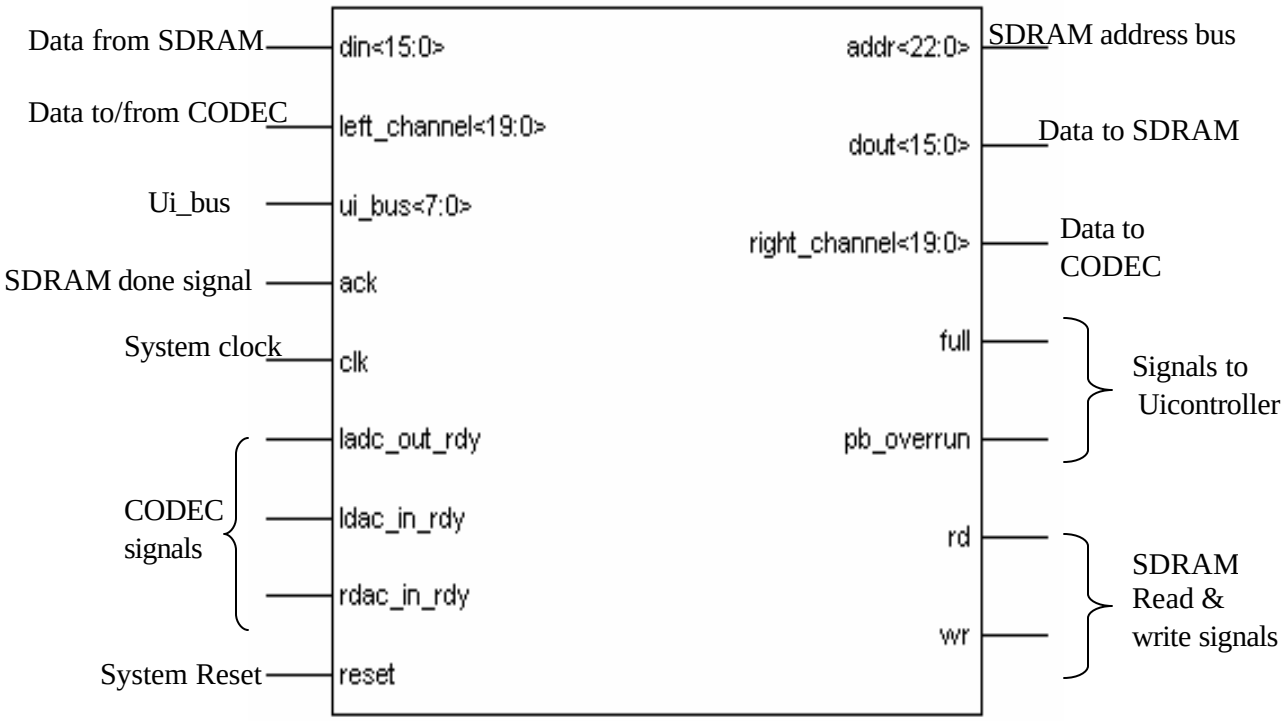


Figure 2

When the storage unit is storing data to SDRAM it waits for a rising edge on ladc_out_rdy signal from the Codec interface. At this point in time the data on left_channel from the Codec interface is valid and is stored in SDRAM by asserting the write signal (wr) high to SDRAM.

When the storage unit is reading data from SDRAM the read signal (rd) is asserted and the data from SDRAM is read into to a register. When either ldac_in_rdy or rdac_in_rdy goes high the data read into the register is placed onto the left_channel or right_channel busses. The data placed left_channel or right_channel is then played back as audio to the user by the codec_interface module.

Effects Engine

The purpose of the effects engine is to provide various time-based effects on the input signal. By employing a circular buffer arrangement of memory, using two read pointers and a single write pointer, the effects engine is able to produce an output signal that is the weighted sum of two previous inputs. It creates the different effects by carefully controlling the difference between the two pointers.

The operator controller (op_cntrl.vhd) accepts signals from the UI bus. As well, the operator controller is responsible for accepting a control signal from the audio codec when new data is ready. Once it detects a valid code on the bus, and new data is

ready, the controller begins operation by setting the different value registers for the correct values for the various modules beyond it.

The address generator (add_gen.vhd) takes as input the registers set by the operator controller. It uses these values to constantly generate new addresses depending on which effect is desired. The write pointer and primary read pointer are controlled by basic 16 bit counters, while the secondary read pointer is a slightly offset version of the primary one.. For the chorus and flange effects, this offset value changes with time, while the phaser and echo effects are time-constant offsets. For a more detailed description of each of these effects, please refer to Appendix F.

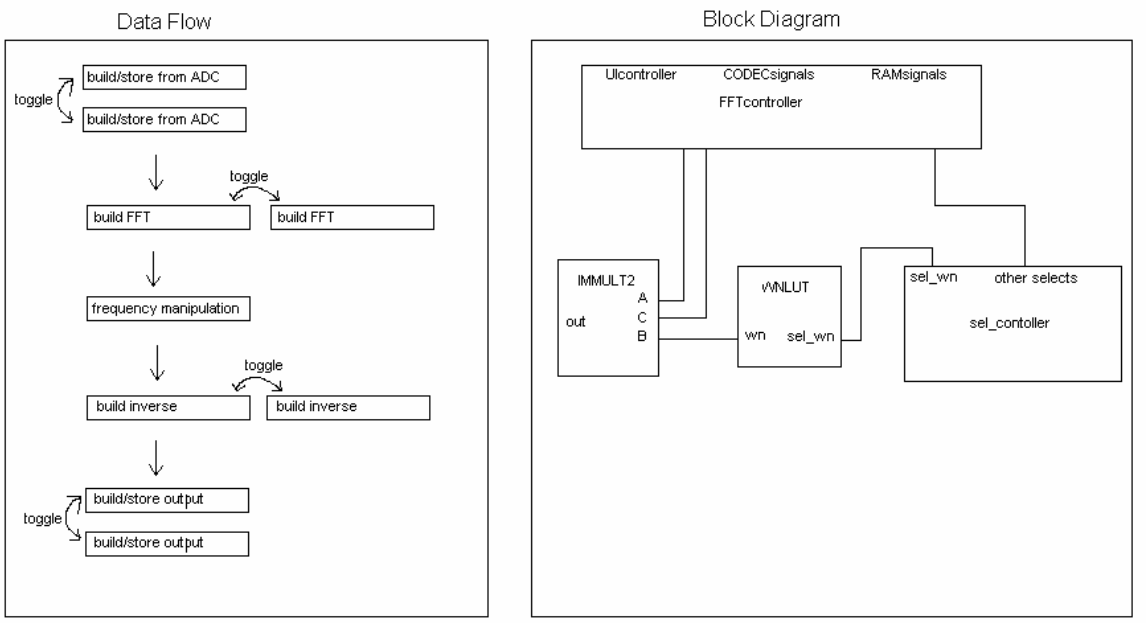
The circular controller (circ_cntrl.vhd) simply arbitrates memory reads and writes. Every time new data is available, the circular controller writes this value to memory, and reads two values. It then waits for the data clock to go low again before returning to its idle state. Once the data has been recovered from memory, it is put into registers for further processing.

Following the circular controller is the mixer (mixer.vhd). The purpose of the mixer is to accept the two values from the circular controller and output the weighted sum of them. Currently the mixer is only capable of weighting the values by values represented by 2^{-n}. It is expected that this may be improved upon before demonstration and presentations, should time permit.

Fast Fourier Transform

The FFT is implemented by a butterfly algorithm [4]. It is a 32 point FFT with 8 bit accuracy. The FFT consists of 4 modules:
-An arithmetic unit called IMMULT2 which performs a complex multiplication and addition for 3 complex inputs, and outputs a complex number. Complex numbers are represented as 2 signed numbers, one for the real component, and another for the imaginary. This is required for each point at each stage in the butterfly algorithm.
-A lookup table, WNLUT, which receives an input and outputs the corresponding Wn, where $Wn = e^{2\pi j/n}$. This contains decimal components. This is handled by outputting the number times 2^6 and then later truncating 6 bits after the multiplication.
-A state machine sel_controller, which has several output signals, selA, selB, sel_wn, counter, stage. This controller works with the FFTcontroller to pull values for to input to the complex multiplier from RAM, select the correct Wn, and output the correct result from IMMULT2 to RAM.
-An overall controller FFTcontroller interacts with RAM and stereo codec through a higher toplevel. It coordinates the building of 32 point samples to RAM from the codec driver, moving values in and out of designated sections of RAM for IMMULT2 with the aid of sel_controller, and the output of the end result to the codec driver.

Below is a diagram of the flow of DATA through sections of RAM, and a block diagram of the FFT unit. Each block in the Data Flow diagram corresponds to an array of 32, 16 bit registers, with only the upper 8 bits used, and the lower 8 bits padded with zeros.



Codec Interfacing

The codec driver (codec_intfc.vhd) has several important signals for interfacing with it that are passed through to various components in the design:
-Two 20 bit parallel input signals to the DAC .
-Two 20 bit parallel output signals from the ADC.
-ladc_out_rdy, radc_out_rdy which flag when data is ready to be read from ADC.
-ldac_out_rdy, rdac_out_rdy which flag when new data can be placed on the shift registers to be written to the DAC.

The _rdy signals must be watched by all DSP units to synchronize the processing of data with the flow of data, and tell the DSP modules when new data must be dealt with, and when processed data must be sent out. Other signals, such as the rd and wr are controlled by the top-level.

Datasheet for Chip

DATA SHEET
Audio Processing Unit

Features

- Stereo Codec
- LCD user interface
- SDRAM Controller
- Audio effects module
- Fast Fourier Transform
- Audio Storage/Playback module

Specifications	
Maximum Clock Freq	43.156 MHz
Minimum SDRAM clock Freq	25 MHz
Number of Logic Cells	1198
Number of Gates	49,174

Audio Parameters	
Codec sampling rate	48.8KHz
Codec sampling resolution	20 bits
Memory Storage resolution	16 bits
Maximum audio storage length	170 seconds

User Interface and Codec Module			
Pin #	**Pin Name**	**Description**	**I/O**
88	clk	System clock	I
93	notrst	Active low replace	I
26	pb1	Active-low push button	I
78	pb2	Active-low push button	I
77	mclk	Codec Master clk	O
59	lrck	Codec left/right clock	O
75	sclk	Codec Shift clock	O
74	sdin	Codec Serial data to DACs	O
76	sdout	Codec Serial data from ADCs	I

LCD Module			
Pin #	**Pin Name**	**Description**	**I/O**
60	data_bus_out<0>	LCD Output	O
62	data_bus_out<1>	LCD Output	O
54	data_bus_out<2>	LCD Output	O
56	data_bus_out<3>	LCD Output	O
63	data_bus_out<4>	LCD Output	O
64	data_bus_out<5>	LCD Output	O
66	data_bus_out<6>	LCD Output	O
67	data_bus_out<7>	LCD Output	O
49	enable_out	LCD control signal	O
46	read_write_out	LCD control signal	O
44	reg_sel_out	LCD control signal	O

SDRAM Module			
Pin #	**Pin Name**	**Description**	**I/O**
141	saddr<0>	SDRAM address bus	O
4	saddr<1>	SDRAM address bus	O
6	saddr<2>	SDRAM address bus	O
10	saddr<3>	SDRAM address bus	O
11	saddr<4>	SDRAM address bus	O
7	saddr<5>	SDRAM address bus	O
5	saddr<6>	SDRAM address bus	O
3	saddr<7>	SDRAM address bus	O
140	saddr<8>	SDRAM address bus	O
138	saddr<9>	SDRAM address bus	O
139	saddr<10>	SDRAM address bus	O
136	saddr<11>	SDRAM address bus	O
95	sdata<0>	SDRAM data bus	O
99	sdata<1>	SDRAM data bus	I/O
101	sdata<2>	SDRAM data bus	I/O
103	sdata<3>	SDRAM data bus	I/O
113	sdata<4>	SDRAM data bus	I/O
115	sdata<5>	SDRAM data bus	I/O
117	sdata<6>	SDRAM data bus	I/O
120	sdata<7>	SDRAM data bus	I/O
121	sdata<8>	SDRAM data bus	I/O
116	sdata<10>	SDRAM data bus	I/O
114	sdata<11>	SDRAM data bus	I/O
112	sdata<12>	SDRAM data bus	I/O
102	sdata<13>	SDRAM data bus	I/O
100	sdata<14>	SDRAM data bus	I/O
96	sdata<15>	SDRAM data bus	I/O
130	ras_n	SDRAM ras	I/O
131	cke	SDRAM clock-enable	O
123	we_n	SDRAM write-enable	O
134	ba<0>	SDRAM bank select	O
137	ba<1>	SDRAM bank select	O
126	cas_n	SDRAM cas	O
41	ce_n	Flash Ram chip enable	O
132	cs_n	SDRAM chip select	O
124	dqmh	SDRAM DQMH	O
122	dqml	SDRAM DQML	O
91	sclkfb	SDRAM feedback clock	O
129	sclk_ram	SDRAM clock	I

FPGA Resource Usage

The completed components of the project exceeded the device resources. For this reason, the project was divided into two parts for implementation. Because of the modular nature and three distinct datapath components, the implemented parts work independently of each other. The specific resource usage is documented in the table below.

Total Implementation Size

	Storage and Effects Engine (Slices)	***FFT Module***	***Both Together***
Number of Slices:	1187/1200 98%	463/1200 38%	122%
Total Number Slice Registers:	866/2400 36%	165/2400 6%	42%
Total Number 4 input LUTs:	1892/2400 78%	878 /2400 36%	98%
Number of bonded IOBs:	58/92 63%	58/92 63%	58/92 63%
Number of Block RAMs	1/4 25%	1/4 25%	1/4 25%
Number of GCLKs:	1/4 25%	1/4 25%	1/4 25%
Number of GCLKIOBs:	2/4 50%	2/4 50%	2/4 50%
Number of DLLs:	2/4 50%	2/4 50%	2/4 50%
Total equivalent gate count	50512	31742	75422

Codec Interface and Effects-engine

Number of Slices: 625 out of 1,200 52%
Total Number Slice Registers: 599 out of 2,400 24%
Total Number 4 input LUTs: 898 out of 2,400 37%
Number of bonded IOBs: 74 out of 92 80%

Total equivalent gate count for design: 11,011
Additional JTAG gate count for IOBs: 3,600

LCD Controller

Number of Slices: 362 out of 1,200 30%
Number of Slice Flip Flops: 151 out of 2,400 6%
Total Number 4 input LUTs: 625 out of 2,400 26%
Number of bonded IOBs: 20 out of 92 21%
Number of Block RAMs: 1 out of 10 10%

Total equivalent gate count for design: 21,651
Additional JTAG gate count for IOBs: 1,008

U.I. Controller

Number of Slices:	53 out of 1,200	4%
Number of Slice Flip Flops:	46 out of 2,400	1%
Total Number 4 input LUTs:	80 out of 2,400	3%
Number of bonded IOBs:	13 out of 92	14%

Total equivalent gate count for design: 899
Additional JTAG gate count for IOBs: 672

FFT

Number of Slices:	463 out of 1,200	38%
Number of Slice Flip Flops:	165 out of 2,400	6%
Total Number 4 input LUTs:	878 out of 2,400	36%
Number of bonded IOBs:	106 out of 92	115%

Total equivalent gate count for design: 8,698
Additional JTAG gate count for IOBs: 5,136

Results of Experiments and Characterization

Throughout the course of the project, numerous experiments were completed to determine the best way to do things. Some of these experiments are documented below.

Modifying Synthesis Options

The Xilinx ISE synthesis tools have options for different synthesis settings. These include optimizing for speed or area, and changing the effort used by the synthesizer from normal to high. During each stage of development, different options were tried to determine the best settings.

Optimization Goal	***Optimization Effort***	***Maximum Clock Frequency***	***Number of SLICEs used***	***Synthesis/Implementation Time***
Speed	Normal	43.156MHz	1198/1200 (99%)	02:14:00
Speed	High	43.482MHz	1187/1200 (98%)	02:30:00
Area	Normal	28.206MHz	1146/1200 (95%)	01:57:00
Area	High	30.348MHz	1140/1200 (95%)	02:15:00

In all cases, the project containing the storage and effects engine module was compiled. The synthesis time is the total time taken to generate a programming file. As shown above, changing these settings has a great effect on the performance of the finished program. It is noted that changing the optimization settings has very little effect on total synthesis time. When set to high there is a marginal improvement in

both speed and area for each optimization goal. However, the differences are so marginal that there appears to be very little benefit.

In addition to the given settings, tests were done to attempt to use incremental synthesis. In this style of synthesis, only the changed components are re-synthesized each time, saving time generating new programming files each time only one or two files are changed. Unfortunately all efforts to make this work resulted in bit streams that didn't function properly. For this reason and due to time constraints, further experiments were abandoned.

External FFT Core

At the outset of this project, it was expected that the APU would contain a fast fourier transform module supplied by an external source [7],[10]. It was learned through trial and error that none of the available FFT cores would fit into the target device. The FFT core generated using the Core Generator system [10] used 927 out of 1200 (77%) total SLICEs. This was using a 32 Point transform with 8 bits of precision. The core downloaded from OpenCores.org [7], with the same specifications used 16 of 10 (160%) possible Block RAM of the available Block RAM, and did not include customizations to modify this. As a result of these cores being far too large to fit into the rest of the project, a separate FFT core using other memory was developed, as described previously.

Designed FFT module

Experiments were conducted to discern the relation ship between multiplier input width, and size. The following date was obtained for multipliers with the given input widths, and outputs of twice that input width.

Input width

4	Number of Slices:	9 out of 1,200 1%
	Maximum combinational path delay: 15.285n	
8	Number of Slices:	36 out of 1,200 3%
	Maximum combinational path delay: 20.112ns	
16	Number of Slices:	140 out of 1,200 11%
	Maximum combinational path delay: 25.519ns	

It can be seen that input width can be is a function of the square of the input width, and that the max combinational delay increases 5 ns with the doubling of the input width. Since the complex arithmetic unit IMMULT2 requires 4 such multipliers, 8 bits is the only feasible size for the input widths, and it has a max operating frequency of just under 50MHz.

Another experiment was done to yield the size of a 32 to 1 multiplexer.

Number of Slices:	18 out of 1,200 1%
Maximum combinational path delay: 13.987ns	

With the previous choice for design, 8 * 4 = 32 such multiplexers would have to be uses, taking up 48% of the available slices. It was attempted to minimize this by

implementing shift registers, to reduce the number of multiplexers required by a factor of 8(the input width).

In an attempt to minimize the size of the original FFT design using multiplexers and registers, the registers were replaced with shift registers, and where there had been 8 MUXes in parallel, there was one MUX moving data in serial form to an output shift register. The control path was modified. The end result was an FFT that took up 94% of the chip, not usable. It could be used with some debugging on a larger FPGA. RAM will be used instead.

External VHDL Components

As a requirement for this project, certain external HDL modules were used. In addition to fulfilling the requirements, it was also found that using external components drastically reduces overall design time and gave more time to concentrate on other areas of design. As a result, it improved the quality of the project and made extra progress possible. A description of each of the externally supplied modules is given below.

Stereo Codec Driver
As the Xstend V1.3.2 board includes a stereo codec interface chip, it was necessary to implement a driver module to control the operation of the codec. It was discovered early on in the project that Xess had provided such functionality through their examples [9]. Only minor changes needed to be made to the supplied codec_intfc.vhd to get it to function and provide basic audio loop-back functions. Once this was accomplished, the digital signal provided by the codec could be routed to any particular component and modified accordingly.

SDRAM Controller
Much like the Xstend V1.3.2 board contained the codec chip, the XSA-100 Board contains 16MB of SDRAM onboard. Using sdramcntrl.vhd as provided [9], it was possible to interface directly with the SDRAM as though it were SRAM. The provided module controlled all necessary timing and refresh operations and made design overall much simpler.

Xilinx Core Generator Modules
Included in the Xilinx ISE software package was the Xilinx Core Generator System. This was used throughout the project as a way of simplifying design of certain elements by generating cores to do specific tasks and implementing these cores structurally rather than behaviorally. As a general rule, structural designs are much easier to debug and design than behavioral designs, so this further reduced design time. Each of the following cores were created by the Xilinx tools and used in the design of the APU:
-lcd_rom.xco – Block RAM configured for read-only operation to hold the characters used for displaying messages on the LCD. Instantiated by lcd_decoder.vhd
-count16.xco – 16-bit counter used to control address pointers for the circular buffer. Instantiated by add_gen.vhd.
-count8.xco – 8 bit counter used in the delay generator to provide an input to the sine-LUT. Instantiated by delay_gen.vhd.

-sine_gen.xco – Sine/Cosine Lookup-Table (LUT) to provide a sine-varying offset to the secondary address read pointer for chorus/flange effects. Instantiated by delay_gen.vhd.
- add.xco – Adder/Subtracter to provide a constant offset value to the secondary address read pointer for echo effects. Instantiated by delay_gen.vhd.
- subtract.xco – Adder/Subtracter to subtract the offset values from the primary read pointer. Instantiated by delay_gen.vhd.

References

[1] Smith, Steven W. *The Scientist and Engineer's Guide to Digital Signal Processing, Second Edition.* San Diego, CA: California Technical Publishing, 1999

[2] DeFatta, David J., Lucas, Joseph G., and HodgeKiss, William S. *Digital Signal Processing: A System Design Approach.* Toronto, ON: John Wiley & Sons, 1988

[3] Lim, Jae S. *Speech Enhancement.* New Jersey: Prentice-Hall, 1983

[4] Lathi, B.P. *Signal Processing and Linear Systems.* Carmichael, CA: Berkeley Cambridge Press: 1998

[5] Antoniou, Andreas *Digital Filters Analysis, Design, and Applications, Second Edition*. Toronto, ON: Primis Custom Publishing, 2000

[6] Smith, Douglas J. *HDL Chip Design,* Madison, AL: Doone Publications, 1996

[7] *OpenCores.org,* URL: http://www.opencores.org, 2003

[8] Swartzlander, Earl E. Jr. *VLSI Signal Processing Systems.* Hingham, MA, 1982

[9] Xess Corporation, *Example Designs, Tutorials, Application Notes.* URL: http://www.xess.com/ho03000.html#Examples, 2003

[10] Xilinx Corporation, *Xilinx IP Center.* URL: http://www.xilinx.com/ipcenter/index.htm, 2003

[11] Smith, Jessamyn *Wired CDMA Network – Application Note*, URL: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/2000f/interfacing/lcd/, 2000.

[12] Micea , Mihai V., Stratulat, Mircea, Ardelean, Dan, and Aioanei, Daniel *Implementing Professional Audio Effects with DSPs*. University of Timisoara, Romania, 2001
URL: http://dsplabs.utt.ro/~micha/publications/pdfs/Audio%20Processor.pdf

Appendix A
VHDL Code

Index to VHDL Code

Entity Name	Description	Status
Toplevel.vhd	Top level entity for all elements	**Compiled**
Codec_interface.vhd	Entity combining all datapath components to control data movement	**Compiled**
Clock_gen.vhd	Generates required clock signals to operate codec	**Compiled**
Codec_intfc.vhd	Brings together interface channels to codec driver	**Compiled**
Channel.vhd	Uses signals from clkgen to move data	**Compiled**
Clkgen.vhd	Generates required clock signals to operate controller	**Compiled**
Storage.vhd	Entity controlling storage and playback functions	**Compiled**
Fx_engine_toplevel.vhd	Combines all effects engine components	**Compiled**
Add_gen.vhd	Address generator to control address pointers for effects engine	**Compiled**
Delay_gen.vhd	Delay generator to generated an offset memory address for circular buffer	**Compiled**
Modulus.vhd	Clock divider for delay generator	**Compiled**
Scaler.vhd	Ensures address falls within range	**Compiled**
Shifter.vhd	Shift register to divide offset value to a controllable value	**Compiled**
Circ_cntrl.vhd	Circular buffer controller	**Compiled**
Mixer.vhd	Mixes two memory output values to produce an output value to hear	**Compiled**
Op_cntrl.vhd	Controls effects engine operation	**Compiled**
Lcd_top_level.vhd	Overall control of the LCD interface.	**Compiled**
Clock_divider.vhd	Divides clock to provide LCD slow clock	**Compiled**
Lcd_decoder.vhd	Determines which message to display and sends proper characters to lcd_driver.	**Compiled**
Lcd_driver.vhd	Initializes LCD and outputs characters.	**Compiled**
Sdramcnt.vhd	SDRAM controller module to provide timing and refresh operations	**Compiled**
UIcontroller.vhd	User Interface that takes in user inputs and tells the other components what to do.	**Compiled**
Storage2.vhd	FFT controller module to provide FFT operations	**Compiled**
Immult2.vhd	Provides complex number multiplication	**Compiled**
Selectcontroller.vhd	Provides memory selection for FFT	**Compiled**
WNLUT.vhd	Look-up table for FFT twiddle factors	**Compiled**
Interface.vhd	Package for LCD interface	**Compiled**
Level2.vhd	Package for overall integration	**Compiled**
Store.vhd	Package for storage functions	**Compiled**
Codec.vhd	Package for codec operations	**Compiled**
Fx_pkg.vhd	Package for overall effects engine operation	**Compiled**
Delay_pkg.vhd	Package for delay generator functions	**Compiled**
Common.vhd	Package for added LCD operations	**Compiled**
Codec_top.vhd	Package for codec top level.	**Compiled**
UIPackage.vhd	Package for user interface functions	**Compiled**
Counter4b.vhd	4 bit counter for storage functions	**Compiled**

Appendix B
Simulation and Test Cases

Index to Test Cases and Simulations

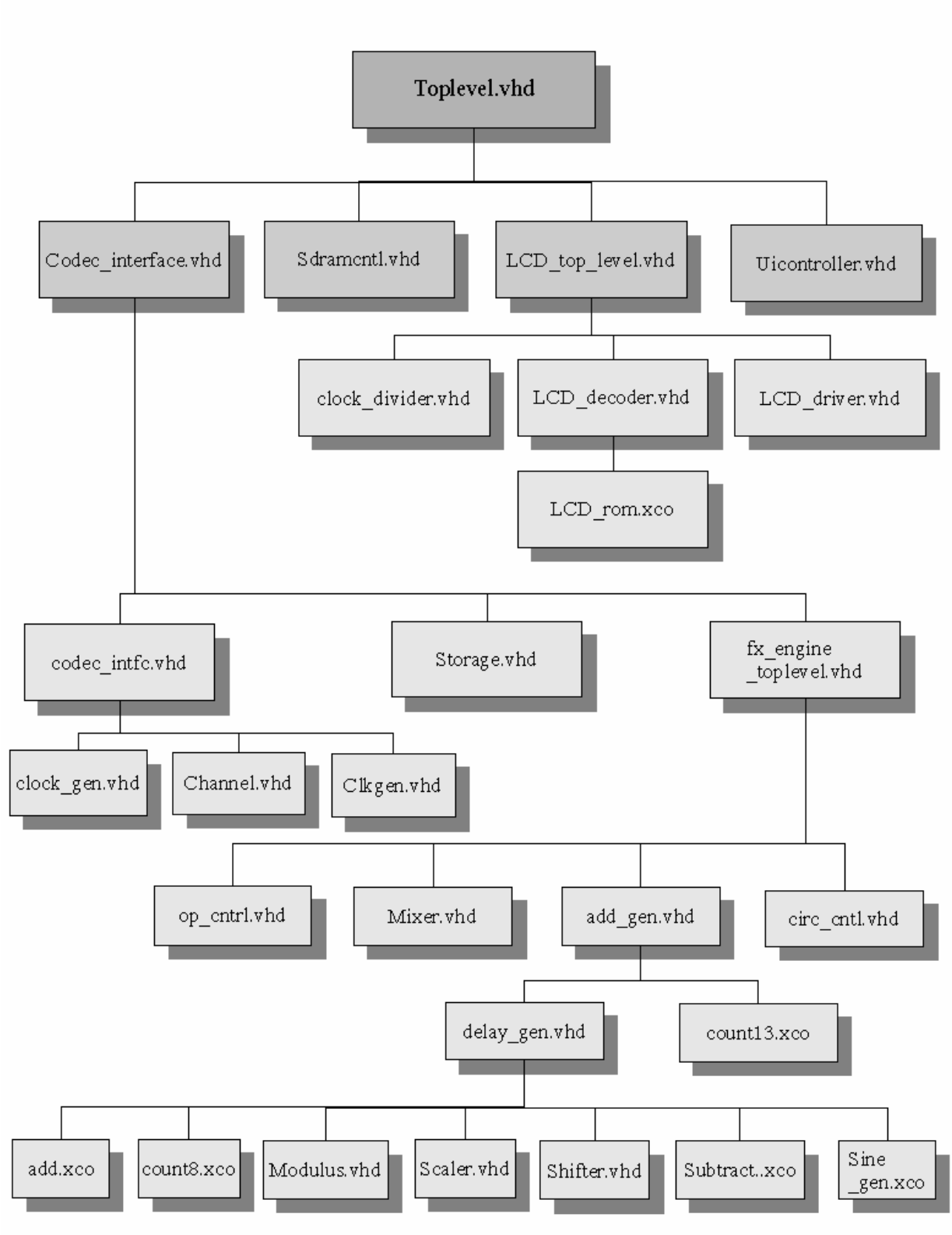
Name	Description	Status
UI Controller	Simulation waveforms for Storage Unit	Simulated
FFT modules	Simulation waveforms for FFT operation	Simulated
LCD Operation	Simulation waveforms for LCD unit	Simulated
Storage Unit	Simulation Waveforms for storage unit and memory operations	Simulated
Effects engine	Simulation Waveforms for Effects Engine	Simulated

Appendix C
Test benches

Test Bench Index

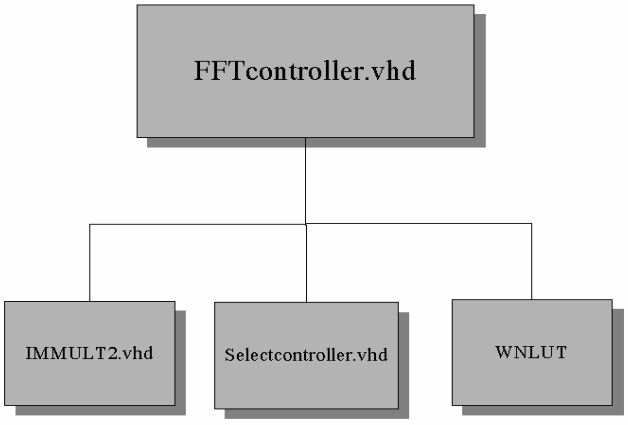
Test Bench Name	Description	Status
lcd_test.vhd	Testbench for LCD interface. Simulates all of the valid inputs.	Simulated – no known bugs
Test.vhw	UI controller testbench (generated)	Simulated – no known bugs
Top_level.vhw	LCD driver testbench (generated)	Simulated – no known bugs

Appendix D
Diagrams of Design Hierarchy



Implemented Design without FFT

FFT Design Hierarchy



Note: As this is the final report, all elements listed above have been Compiled/Simulated and have no known bugs or are having minor bugs currently being worked out. As well, two separate design have been drawn to reflect the way that the design was implemented. FFT controller can be considered to be connected to Codec Interface, in much the same way storage and fx_engine_toplevel are.

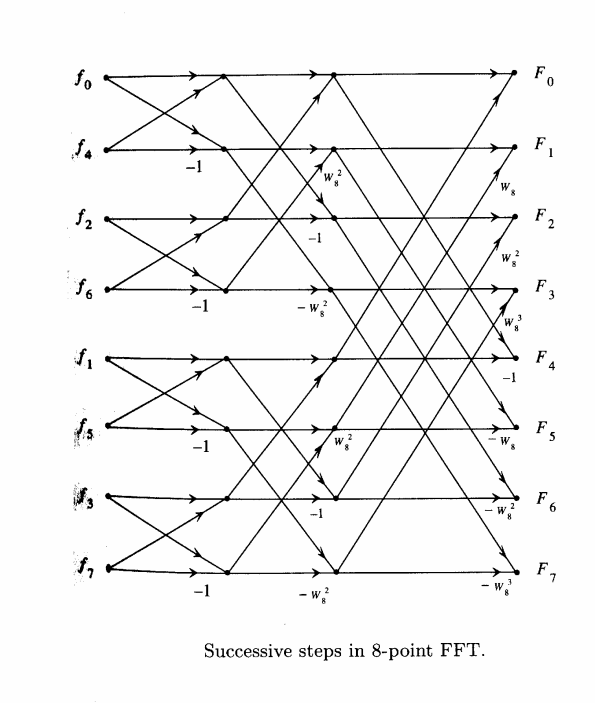
Appendix E
FFT Operation

Description of FFT Implementation

The inputs are 2 arrays of 32, 8 bit signals, for the imaginary and real components of the 32 points to be transformed. Most have been deleted to save space. Also shown are the state, next_state, calc, reset and counter signal.

In the FFT, 1 complex multipliers is multiplexed to perform all 32 multiplications for each stage(5 in all).

Description of FFT



Successive steps in 8-point FFT.

Figure 1: Successive Steps in an 8-point FFT taken from "Signal Processing & Linear Systems" by B.P. Lathi. [4]

index	stage0			stage1			stage2		
	Fh	Fl	Wn	Fh	Fl	Wn	Fh	Fl	Wn
000	000	100	000	000	010	000	000	100	000
001	000	100	100	001	011	010	001	101	001
010	010	110	000	000	010	100	010	110	010
011	010	110	100	001	011	110	011	111	011
100	001	101	000	100	110	000	000	100	100
101	001	101	100	101	111	010	001	101	101
110	011	111	000	100	110	100	010	110	110
111	011	111	100	101	111	110	011	111	111

Table A1: summary of indexes in Binary

Figure one shows the butterfly algorithm for an 8 point FFT. Table A1 summarizes the terms that must be multiplied together to come up with intermediate stages within the

FFT. This algorithm can be implemented with 2 arrays of 8 complex numbers, a lookup table for complex Wn and a device that computes Fh+Fl*Wn. One array would have to be an input array, and the other array would be the output array. Table A1, shows for an index(in binary) for the answer of Fh+F1*Wn to be written to the answer array for each stage. The index under Fh, F1 indicate the index from which these values come from in the input array. The index for Wn indicate which Wn must be looked up. At the end of a stage, load the output array to the input array, and then compute again, only using the appropriate logic to control the indexes. Use a counter to increment through the output index. From the table you can see that:
In stage0
Fl = '0'& index(1) & index(2)
Fh = '1' & index(1) & index(2)
Wn = 'index(0) & "00"

In stage1
Fl = index(2) & '0' & index(0)
Fh = index(2) & '1' & index(0)
Wn = index(1) & index(0) & '0'

In stage2
Fl = '0' & index(1) & index(0)
Fh = '1' & index(1) & index(0)
Wn = index

From the above logic for 8 point FFT, the logic for 32 bit FFT can be inferred to be as in the controller in FFT.vhd. For more stage1 and above, the column of '0' and '1' simply shift to more and more significant bits. Below is a block diagram of FFT.vhd architecture. Note, the math is in 2's compliment form.

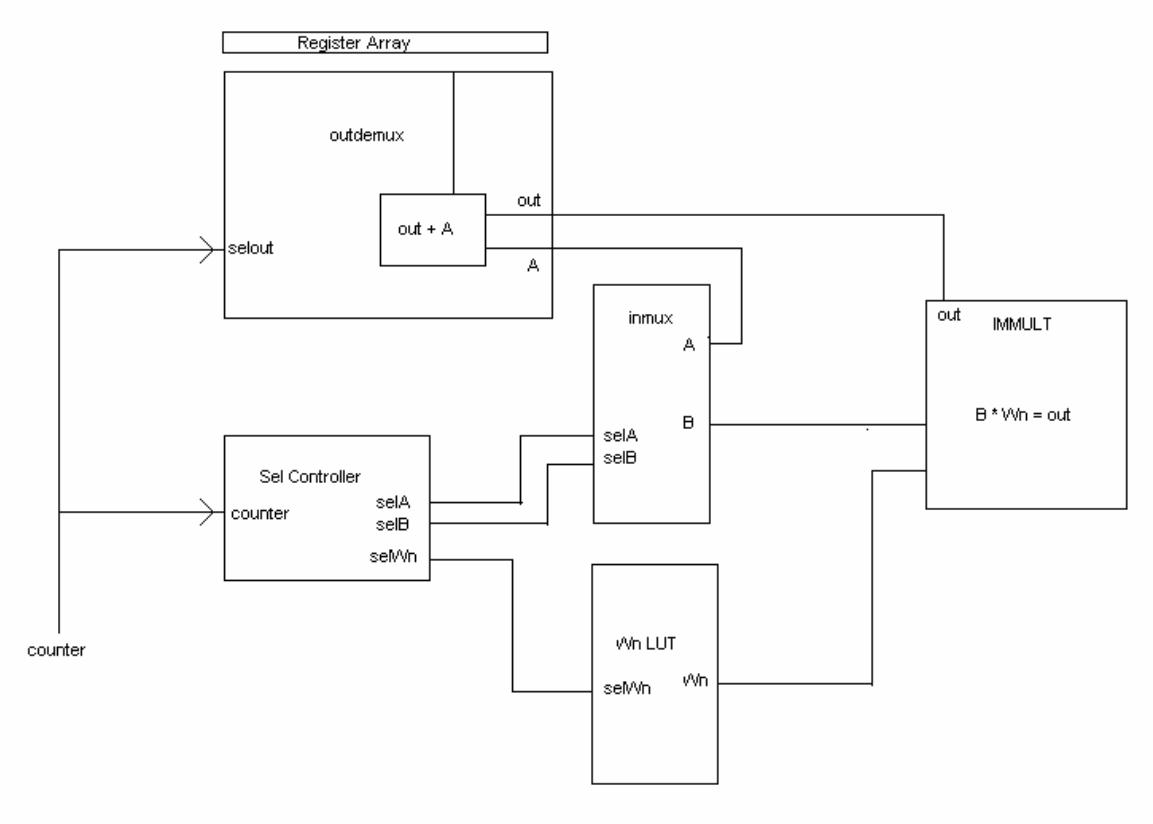


Figure A1: FFT Architecture

Appendix F
FPGA Implementation of Audio Effects

FPGA Implementation of Audio Effects - An EE 552 Student Application Note

by Richard Schultz

Audio Processing Unit
Duncan Campbell - Grant Cunningham - Clint Lozinsky - Richard Schultz

INTRODUCTION

Digital signal processing (DSP) is a very exciting market these days. FPGA's and ASIC's make up such a large part of this market that FPGA manufacturers are predicting their products will soon completely take over standard DSP microprocessors. While this prediction might be a little over ambitious, digital signal processing in FPGAs is gaining momentum. And while audio processing only makes up a fraction of the DSP market, it is both interesting and useful to understand how certain audio algorithms work.

The effects described here have all been implemented in the time domain. Frequency domain processing is possible for certain effects, but time domain processing is much easier. Because audio sampling is done in the time domain, it is inherently easier to process in this domain as it does not require hardware to transform the signal. Using a few basic elements outlined below one can easily and effectively implement a digital effects processor.

THE BUILDING BLOCKS

The Circular Buffer

The first critical element in effects implementation is a circular buffer. Circular buffers are a crucial component to any digital signal processing application. They permit data to be continually updated, and overwrite the oldest data in memory. Looking at figure 1, data is originally written to the very first memory location. As new data comes in, it is written to the next available memory address. Once the address pointer

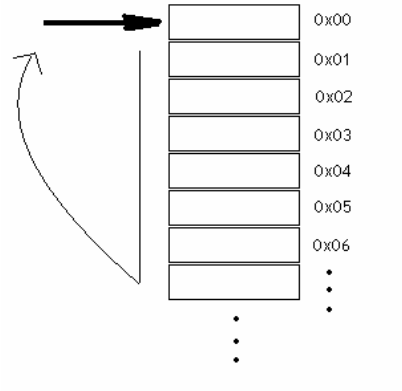


FIGURE 1 – A CIRCULAR BUFFER

reaches the end of the circular buffer, it immediately wraps around and starts writing to the first memory address again. Storing data in this manner, 2^N samples are always available in the buffer, and all that is necessary is controlling the pointers. This can be most easily done with a simple *N*-bit counter.

For the implementation of the effects described here, one write pointer and two read pointers are used as shown in figure 2. Using these three pointers, one is able to obtain an input, $x[n]$ and two outputs, $y_1[n]$ and $y_2[n]$. Both $y_1[n]$ and $y_2[n]$ are time delayed versions of the input, and can be written as:

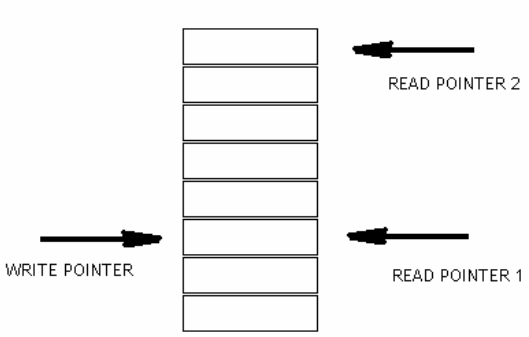


FIGURE 2 – A CIRCULAR BUFFER FOR AUDIO EFFECTS

$$y_1[n] = x[n - t_{d1}]$$
$$y_2[n] = x[n - t_{d2}]$$

where t_{d1} and t_{d2} are constant delay factors. t_{d1} is the difference between the first (current) read pointer and the write pointer. Generally this could be as little as one clock cycle (giving adequate time to store a value before immediately reading it back) but for purposes of this document it is assumed to be zero. The human ear does not notice time delays less than about 50ms, so this is a reasonable assumption. Using this, we can simplify our outputs to:

$$y_1[n] = x[n] \qquad \textbf{(1)}$$
$$y_2[n] = x[n - t] \qquad \textbf{(2)}$$

With t being a value corresponding to the time difference of the two samples. This difference is the product of the sample rate of the digital audio stream, and the number of samples. Also note that while n corresponds to an index in the circular buffer and is thus constrained to *N* (total number of samples stored), the circular buffer treats the data as continuous. The only precaution that must be taken is to keep $t < N$.

The Mixer

The next element required to implement a few basic effects is a mixer. This is a very simple element that in effect takes in two signals and outputs the weighted sum of them. The two inputs to the mixer are the values from the circular buffer, so the output of the mixer is given by the equation:

$$y_{out}[n] = ay_1[n] + by_2[n]$$

Which, from Equations 1 and 2, can be simplified to yield:

$$y_{out}[n] = ax[n] + bx[n-t] \qquad \textbf{(3)}$$

Where a and b are the weighting coefficients.

With Equation 3, one can easily implement a large range of effects by changing only the values α, β and τ.

THE EFFECTS

Echo

The echo effect is the easiest of the effects to implement. This effect is created by adding the current sample to a previous sample. Using Equation 3, it is easy to see that this effect is created by keeping T constant. Therefore an echo effect is simply described by the equation:

$$y_{echo}[n] = ax[n] + bx[n - t_{echo}] \qquad \textbf{(4)}$$

Where t_{echo} is the echo length.

Obviously, this is accomplished using only the mixer and circular buffer outlined in equation 3 above. The only thing that is necessary to accomplish this is to subtract a value from the current read pointer and use it as the second read pointer.

Chorus

The chorus effect is only slightly more complicated than the echo. In a similar manner to an echo, the chorus is produced when you add the current sample to a previous sample, only the amount of delay is varied sinusoidally. By varying the delay from 40ms to 60ms continuously at a rate of 0.25 Hz, you have a standard chorus effect. The equation that describes this is:

$$y_{chorus}[n] = ax[n] + bx[n - t(n)] \qquad \textbf{(5)}$$

Where $t(n) = A\sin(2pf)$, A = Constant multiplier and f = frequency of variation

To implement this effect, a sine look up table can be generated, and this can be subtracted from the current read pointer. Numerous HDL design libraries include sine-cosine lookup tables, and their usage is rather simple.

Flange

The flange effect is very similar is structure to the chorus effect. In fact, the only thing that changes is the amount of varying delay, and the rate at which it occurs. For a standard flange effect the delay generally varies from 0ms to 10ms at a rate of 0.5Hz. Obviously the equation for a flange effect is the same as that for a chorus effect:

$$y_{flange}[n] = ax[n] + bx[n - t(n)] \qquad \textbf{(6)}$$

Where, once again $t(n) = A\sin(2pf)$, A = Constant multiplier and f = frequency of variation. Cleary this is the same as the chorus effect in equation 5. The only difference between implementation is the specification of the A and f values.

Phaser

The phaser effect is the result of two identical, yet out of phase, signals being added together. This produces various notches in the phase response and has a canceling effect which is audible to the human ear. In essence, this is basically the same effect as the flange and chorus, only with different parameters once again. The equation for a phaser is:

$$y_{phaser}[n] = ax[n] - bx[n - t(n)] \qquad \textbf{(7)}$$

In this case, $t(n)$ can be anything from a sinusoid to a saw-tooth or even a constant value. The only major difference is the sign of B, which results in the phase canceling effect. The varying delay isn't completely necessary for this effect, but it does have improved tonal qualities when it is changed by some small factor.

Once again, this can be implemented in the same manner as the flange and chorus effects. The only necessary precaution is to ensure that the mixer is capable of signed arithmetic. If using VHDL, the std_logic_vector array type has signed arithmetic capabilities built in, so one does not need to generally worry about this too much.

OVERALL IMPLEMENTATION

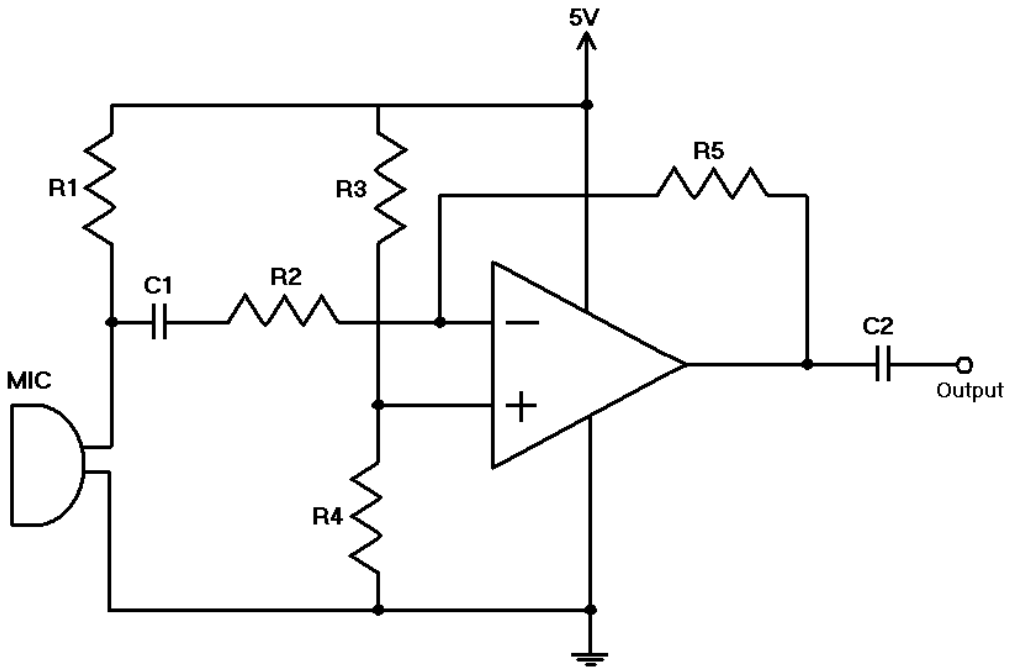
Certain other factors need to be taken into consideration when designing these effects into an FPGA project. As is clearly evident, the circular buffer needs to be large enough to be able to produce a noticeable echo effect. This requires the usage of memory, and there are numerous types of memory available to the FPGA designer. The Block RAM found on Xilinx boards is generally of sufficient size for this, and is easy to use. For this reason, as well as its lack of external components, it is the obvious solution. SDRAM (Synchronous Dynamic RAM) is another option, although this requires the usage of a separate SDRAM controller to handle refresh cycles and other control aspects. If the project is not already using SDRAM this is perhaps not feasible in all instances. However, if SDRAM is available, its larger sizes make it preferable for total flexibility of the effects. SRAM (Static RAM) or Flash memory is yet another option, as it does not have the stringent control requirements SDRAM has. However, one may find that the board they have to work with doesn't allow them to use the SRAM, or its resources may be used by other FPGA elements. SRAM is also more expensive than SDRAM, so this may make it less feasible in certain instances.

Aside from that, implementation is rather straightforward. As the sampling rate of the audio is generally much lower than the clock rate for an FPGA project, all memory read and write operations, as well as any mixing and other post-processing operations can generally be done with extra clock cycles to spare. Even at a standard sampling rate of 44-48 kHz and a conservative clock of 25MHz, there are approximately 500 clock cycles to carry out the required operations.

CONCLUSIONS

As described, implementing standard audio effects in an FPGA is not as complicated as one might assume. It is also very rewarding, as the results of the implementation are noticeable by anyone. These few basic effects are used widely in the music industry to process vocals and instruments, so their application gets heard by literally millions of ears every day. As well, the way the methods used to implement the effects are applicable to various other fields of digital signal processing. Based on the ease of implementation in an FPGA, it’s no wonder that FPGA's occupy a large share of the DSP market.

Appendix G
Microphone Preamp Schematic



Component	Value
R1*	4K7
R2	1K
R3	100K
R4	100K
R5**	100K- 1M
C1	0.1uF
C2	0.1uF

Group Self-Evaluation

After discussion with the group, it was decided that marks should be distributed evenly among group members.