turnitin **Turnitin Originality Report**

FPGA Implementation of a Real Time Acoustic Echo Cancelling System by Louis Yang

From Spring 2011 Masters projects (Final Graduate Project Reports)

Similarity Index	Similarity by Source	
3%	Internet Sources:	2%
	Publications:	0%
	Student Papers:	2%

Processed on 17-May-2011 18:58 PDT
ID: 187774972
Word Count: 15658

sources:

1 < 1% match (student papers from 11/12/10)
Submitted to CSU, San Jose State University on 2010-11-12

2 < 1% match (Internet from 2/17/09)
http://automation.usa.siemens.com/pds/Docs/9300_Series/Technical_Notes/9300_Series_Profibus_Protocol.pdf

3 < 1% match (Internet from 2/22/11)
http://www.wolfson.co.uk/productsfr.htm

4 < 1% match (student papers from 12/06/04)
Submitted to Texas A&M University, College Station on 2004-12-6

5 < 1% match (Internet)
http://support.simware.com/ts_rumba/solution/pdf/AS400/3118.pdf

6 < 1% match (Internet from 9/9/10)
http://ysfactory.nobody.jp/bb/wep/stm_blst.html

7 < 1% match (student papers from 09/18/07)
Submitted to Concordia University on 2007-09-18

8 < 1% match (Internet from 1/5/11)
http://www.elenota.pl/en/search.php?szuk=&man=ISSI&ile=50&start=0

9 < 1% match (Internet from 4/7/09)
http://www.freepatentsonline.com/5859778.html

10 < 1% match (student papers from 12/11/09)
Submitted to CSU, San Jose State University on 2009-12-11

11 < 1% match (publications)
A Khomich. "Upgrade of the PreProcessor system for the ATLAS level-1 calorimeter trigger", Journal of Instrumentation, 12/21/2010

12 < 1% match (Internet from 12/10/06)
http://preprints.ians.uni-stuttgart.de/downloads/2004/2004-013.pdf

13 < 1% match (Internet from 4/4/10)
http://www.freepatentsonline.com/7616025.html

14 < 1% match (Internet from 1/19/11)
http://fileadmin.cs.lth.se/cs/Personal/Pierre_Nugues/memoires/erik/polyphonic_pitch_modification.pdf

15 < 1% match (publications)
Berliner, R.. "A large area position sensitive neutron detector", Nuclear Instruments and Methods In Physics Research, 19810615

16 < 1% match (student papers from 10/29/09)
Submitted to CSU, San Jose State University on 2009-10-29

17 < 1% match (student papers from 04/11/11)
Submitted to American Intercontinental University Online on 2011-04-11

18 < 1% match (Internet from 12/18/07)
http://www.fpga-faq.com/archives/48700.html

19 < 1% match (Internet from 8/26/10)
http://ecaaser2.ecaa.ntu.edu.tw/weifang/WSN/Simulating%20the%20Power%20Consumption%20of%20LargeScale%20Network%20Applications.pdf

20 < 1% match (Internet)
http://www.weixun-ic.com/data/saa7146ah.pdf

21 < 1% match (Internet from 12/15/09)
http://direct.xilinx.com/publications/magazines/dsp_01/xc_pdf/p28-31_dsp-alpha.pdf

22 < 1% match (student papers from 03/13/09)
Submitted to Clarkson University on 2009-03-13

23 < 1% match (Internet from 12/20/07)
http://omega.com/manuals/manualpdf/M3730.pdf

24 < 1% match (Internet from 7/12/10)
http://wwwpub.utdallas.edu/~cantrell/matzke.pdf

25 < 1% match (Internet from 12/26/07)
http://rtdusa.com/NEW_manuals/hardware/datamodules/PCI4520_DM7520_SDM7540_SDM8540_BDM61001000

26	< 1% match (Internet from 4/29/11) http://www.coursehero.com/file/1666316/intro-overview/
27	< 1% match (publications) Chipalkatti, J.V.. "On equations defining Coincident Root loci", Journal of Algebra, 20030901
28	< 1% match (student papers from 04/11/11) Submitted to City University of Hong Kong on 2011-04-11
29	< 1% match (student papers from 04/06/09) Submitted to University of Central Florida on 2009-04-06
30	< 1% match (student papers from 01/09/09) Submitted to Northport High School on 2009-01-09
31	< 1% match (student papers from 12/04/09) Submitted to University of Lugano on 2009-12-04
32	< 1% match (student papers from 07/13/07) Submitted to University of Newcastle on 2007-07-13
33	< 1% match (student papers from 11/11/09) Submitted to University of KwaZulu-Natal on 2009-11-11

paper text:

1**FPGA Implementation of a Real Time Acoustic Echo Cancelling System MSEE Project by Louis Yang**

Last updated May 16, 2011

1**Project Advisor: _____ Date: _____ Project Co-advisor: _____ Date: _____ Graduate Coordinator / EE** 297B **instructor: Tri Caohuu Department of Electrical Engineering, San Jose State University, San Jose, CA Louis Yang 4194 Hamilton Ave. #6, 611-18-2708 San Jose, CA 95130 408-674-8226 louis_yang2@yahoo.com Table of Contents** FPGA Implementation **of**

16**Department of Electrical Engineering, San Jose State University,**
..... 2 **San Jose,**
CA 2
Abstract

 Abstract
Wireless communication and video conferencing transmit audio data at full bandwidth and with noticeable delay, requiring an efficient acoustic echo cancellation system. This project prototypes an acoustic echo cancelling (AEC) system on a FPGA board. Two boards each with speaker and microphone simulates two telephones. In the absence of echo cancelling, the speaker will hear an echo one second after the speech. With echo cancelling, the speaker will not hear such echo. The bulk of the project explores different algorithm acceleration methods using an FPGA. Algorithm used is frequency-domain adaptive filter. The FPGA design can run the AEC algorithm at 32 kHz. Using FPGA in Embedded Systems Currently the main application for FPGA is prototyping. This project does not deal with this usage of the FPGA. Instead the objective is direct use of FPGA in embedded systems. This project specifically uses the Altera Cyclone series. In the Xilinx line up, it would be the Spartan series. FPGA direct use has the following benefits when compared with ASIC: Reduced Startup Cost and Risk – ASIC is a high volume technology, with high starting costs. Once the chip is made it cannot be changed so all bugs have to be cleared early on. FPGAs can be purchased in unit of one for price as low as $13 [1] . Getting started with FPGA therefore requires smaller investment and lower risk. It's perfect for those brave souls who want to start their own business! Reduced Testing Cost and Faster Time to Market – Since FPGA can be reprogrammed later on, not all bugs have to be cleared at design time. Many of the more costly ASIC verification techniques can be bypassed, reducing testing cost. Reduced Coding – The ability to reprogram later on in general simplifies the RTL coding. As an example, RS232 UARTs will have work at different rates for different embedded systems. But once deployed, the system will always use the same rate. In other words, a common situation is that customer #1 will always use 115200 bps while customer #2 will always use 57600 bps. An ASIC RS232 UART must support multiple baud rates, and so it needs to have some kind of "baud rate" register that specify the rate. To be standard compliant, the ASIC UART better implement all eight RS232 signal pins. An FPGA implementation can have a fixed baud rate where the baud rate is a fixed counter value. There's no need to write Verilog code for a logic to load the "baud rate" register. Usually only transmit and receive pins are needed. The FPGA implementation can usually get away with just implementing two of the eight signal pins, and implement the rest on an as needed basis. Processors versus FPGA Always use the processor to the greatest extent possible! The FPGA's great flexibility is also a great trap for those who do not understand the embedded world cost structure. A large variety of high volume processors are readily available for the embedded designer, and a comparison of these processors will with FPGA shows that processors can absolutely crush FPGAs in terms of cost. The following comparison of NIOS 2 versus the Microchip PIC32 and Atmel ARM9E should illustrate this point. Currently, low cost FPGAs do not have hardened processors. Implementing a processor on FPGA means using a softcore processor such as the Altera NIOS 2 or the Xilinx Microblaze. Since this project uses the Altera Cyclone series, the NIOS 2 is used as the benchmark example. On the Cyclone series, the NIOS 2 maximum benchmark is 195 MIPS. [2] This 195 MIPS is achieved using the fastest grade Cyclone III available and running NIOS 2 at 175 MHz. [2] Lower cost FPGAs and using custom logic in tandem with the NIOS 2 will both slow down the processor. In this project, the first implementation runs NIOS 2 at only 95 MHz. The NIOS 2 performance on Cyclone series FPGAs puts it in between 32 bit microcontrollers such as the Microchip PIC32 and low end ARM processors such as the Atmel ARM9E. [3,4] But unlike the PIC32 or ARM9E, the FPGA requires a large amount of supporting chips. FPGA designs have to be stored in off chip flash memory. Cyclone FPGA fabrication processes are digital so they don't have ADCs (analog to digital converters). Memory controllers on Altera Cyclone series are not free and will require more money spent on purchasing IPs. Any soft implementation of well known functions like USB or Ethernet MAC layer will be more expensive on FPGA than on a mass produced processor. The table below summarizes the cost and performance of an FPGA versus a processor.

Processor Performance Features Price Altera NIOS 2 Microchip PIC 32 Atmel AT91SAM9261 195 MIPS [2] 1.11 MIPS/MHz for fastest variant. Can be significantly slower. No ADC. Must pay for any IP, including NIOS 2 and memory controllers. $12.80 ~ $29.60 for EP3C5, the smallest Cyclone 3. No volume discount listed [6] 125 MIPS [3] 1.56 MIPS/MHz On chip Flash, ADC, Ethernet, USB FS 210 MIPS [4] 1.105 MIPS/MHz On chip Flash ADC, Ethernet, USB FS/HS, DDR2 $3.90/unit to $8.62/unit for 25 $13.89/unit for 25 units, depending on units [5] peripheral [5] FPGA's Unique Role FPGA is all about low volume. Well known, high volume applications such as Ethernet, USB, H.264, and running Linux have all been addressed extensively with high volume products. FPGA is a coprocessor that accompanies a standard processor, doing things that are not popular enough to make it into ASIC. In this project, the FPGA is viewed as an algorithm accelerator, using two mechanisms: custom datapath and array processing. Custom datapath means to map a portion of the algorithm in hardware. Array processing means to apply a single operation on a large array of values. FPGA is strong at array processing for two reasons. The first is that low end standard processors do not have vector instructions. High end standard processors have too many other features not needed in embedded systems. The second reason is that additional FPGA I/O pins can be obtained for minimum cost increase. To illustrate this point, the price list of a particular Cyclone FPGA is shown in the table below. Part Number Free IO Pins Price [6] EP3C16E144C8N 84 $26.70 EP3C16U256C8N 168 $31.70 EP3C16F484C8N 346 $33.20 All three FPGAs belong to the same type (EP3C16) and the same speed grade (C8N). The IO pins can increase by roughly four times, for just a 24% increase in FPGA price. The FPGA is build to read in an array of values from multiple memories all at once, processing these values in parallel, and stuffing them back into main memory in parallel. This project uses a 32 bit wide memory. Although the processing efficiency doesn't look quite so spectacular, it's the thought that counts. Project Setup Two FPGA boards are connected by a ribbon cable. The two boards simulate the two ends of a telephone system. These boards are not the same. The board on the left is a simpler DE1 board that does not have echo canceling. It will simply pass data. Data from DE2-70 board will be passed to the speaker #1. Data from microphone #1 will be passed to the DE2-70 board. It acts like a phone that does not have echo cancelling capabilities. The DE2-70 board will pass data as well as do echo canceling. The data passing is analogous to the DE1 board. The data from DE1 board will be passed to the speaker #2. The data from the microphone #2 will be passed to the DE1 board. The echo canceling feature is illustrated below. Suppose the data from DE1 board says "hello". This data will be broadcasted by speaker #2. Microphone #2 will then pick up the "hello". In the absence of acoustic echo cancelling (AEC) the "hello" will be sent back to DE1 board. The job of the AEC algorithm is to filter out the "hello". The "hello" echo only gets noticeable if there's a lag between the speaker on the left side (the DE1 board) and the retransmission of "hello". To make the echo noticeable, the board on the right (DE2-70) will introduce a one second delay on all voice traffic going back to the DE1 board. This way, without the echo cancelling, the "hello" can be heard. Cyclone 2 The boards used in this project, the DE1 and DE2-70, are both available from Terasic Technologies (www.terasic.com.tw). The DE1 board is also available from Altera, where it's known as the Cyclone II Starter Kit. Both boards are based on the cyclone II series of FPGAs. As a technology, the Cyclone II is one generation behind the current Cyclone III/IV technologies, which is in turn one generation behind the current Spartan 6 technology. Although Altera technology is behind at the moment, the next version of cyclone would be sampling in 2012. Altera is chosen over Xilinx mainly because it seems easier to learn from. The literature section appear better organized and Altera provides free NIOS 2 tools, and a time limited NIOS 2 IP that runs for one hour. The comparison between this project's Cyclone 2 technology and current generation Altera and Xilinx technologies are summarized below. Altera Cyclone 2 [8] Altera Cyclone 3 [8] Altera Cyclone 4 [9] Xilinx Spartan 6 [10] Manufacturing Process 90 nm 65 nm 60 nm 45 nm NIOS 2 top speed [2] 140 MHz 175 MHz 165 MHz N/A High Speed Transceiver for PCI Express Yes Yes Memory Controller Yes (DDR2, 3) Efficient shift register implementation No No No Yes Peripherals Although the center piece of the project is algorithm acceleration, the relevant peripherals attached to the DE1 and DE2 boards has to be made to work for the demonstration to occur. Each peripheral is implemented on three separate levels. At the hardware level a set of Verilog files implement the basic signaling. On top of that is "C" code that allows a NIOS 2 program to use the peripheral. The highest level of implementation is a C# file

that runs on the PC for testing or configuration purposes. For example, in order for the FPGA board to send data to the PC, the NIOS 2 level C code copies the data into a FIFO. The Verilog code takes the FIFO's data and transmits the

> 23**data one bit at a time,** basically working as **a** parallel to **serial**

converter. The PC's C# code then reads the PC buffers and displays the data. The following discussion is actually an abridge version of the documentation that can be found on the CD. Even many of the high level details are left out to prevent the report from becoming too long. RS-232 UART RS-232 is implemented to allow for interaction with the PC. This is used for configuration and testing. There are many test routines in project that involves the PC sending data to the FPGA, the FPGA processing the data, and sending the results back

> 33**to the PC. As mentioned** in **the**

section "Using FPGA in Embedded Systems", the RS-232 UART used in this project is not a full RS-232. The timing information is hardcoded, and only two of the eight RS232 signal pins are implemented. This is not just for academic use, it's actually a realistic approach. The ability of the FPGA to be customized for each customer, and for it to be re- flashed later if needed, allows such quick implementation of the RS232. Files List: Verilog Files ? uart_fifo08.v - MegaWizard instantiated FIFO that is 8 bit wide. It will store in coming UART data collected by the RS-232 receiver (uart_rx.v). ? uart_fifo32.v - MegaWizard instantiated FIFO that is 32 bit wide. It will store data that will be sent by the RS-232 transmitter (uart_tx.v). ? uart_rx.v - UART receiver that stores incoming UART data into an 8 byte FIFO. This is basically a serial to parallel converter. ? uart_tx.v - UART transmitter that reads the data from the 32 bit FIFO and sends it out. ? uart_avalon.v - Top level file that contains the transmitter (uart_tx.v), receiver (uart_rx.v), and the two buffers that they use (uart_fifo08.v and uart_fifo32.v). C Files for NIOS 2 ? rs232.h and rs232.c – the key functions are "int rs232_read()" and "void rs232_write(int value);" C# Files for PC Side ? MyRS232.cs - The .Net library has a SerialPort object already. This class wraps around the SerialPort object and provides a few more useful functions. Audio Codec The audio codec peripheral interfaces with the WM8731 audio chip to enable the speaker and microphone to work. [7] The audio chip and the Verilog file interfacing to it runs at 12 MHz, while the rest of the system runs much faster. A dual clock FIFO is used to interface this 12 MHz clock domain

> 25**to the** rest **of the** system. **The** relationship **of the FIFO**

to the audio module is shown below. The WM8731 audio codec is a serial to parallel converter. The chip implements a left and right audio channel. In this application though, they have been reduced to the same number. Sending audio data to the speaker means sending it

> 7**one bit at a time** into **the** codec. Reading data **from the**

microphone means to read it one bit at a time

28**from the** codec. **The** algorithm **for the** Verilog code **is shown** pictorially **below.**

Each data read/ data write with the audio codec is a long serial to parallel conversion process that involve a long set of states. The 12MHz clock driving the WM8731 audio chip and this peripheral is the "clk". The "bit_clk" is the data clock for sending data to the speaker

7**one bit at a time** and reading data **from the**

microphone

7**one bit at a time. The**

"read_tick" and "write_tick" are control signals to that are one "clk" cycle wide. In contrast, the "dac_lrc" and "adc_lrc" are control signals to the audio chip that are one "bit_clk" wide. File List: Verilog Files: ? audio.v - This module does two things. It will read from a FIFO and output its content to the audio codec, which then becomes the speaker output. It will also read the microphone input from the audio codec and store it into another FIFO. ? audio_write_fifo16.v - The speaker output data goes into this FIFO ? audio_read_fifo16.v - The microphone input data will be placed into this FIFO. This FIFO is a show-ahead FIFO in this project, but in general does not have to be so. If it is not a show-ahead FIFO, the sample read will just be one sample older. ? audio_avalon.v - This file contains the audio.v and the two FIFO files. This is the component to be instantiated in SOPC builder. Testbench Files (ModelSim only): ? audio_avalon_tb.sv C Files for NIOS 2: ? audio.c and audio.h – functions for configuring the audio codec, reading from the microphone FIFO, and writing to the speaker FIFO. GPIO Communication A wire ribbon connects the DE1 to the DE2-70 board. The communication method is a basic one: The data is transmitted MSB first, so that the upper 8 bits are transmitted first. The time that "upper_bits_valid" remain high is a fixed value chosen so that the data transmission rate is slightly higher than 32 kHz. This enable the transmitter and receivers to run as slowly as possible, at a speed that is just fast enough to keep up with the real time needs of the audio codec. Since the data rate is slightly faster than 32 kHz, at times there will be no data. In that case, the "upper_bits_valid" will stay low. This basic communication scheme is implemented in "basic_receiver_16bit.v" and "basic_sender_16bit.v". Data is then stored in a FIFO. File List: ? ? ? ? ? ? fifo_in_avalon.v - Custom component that interface with the reading end of a FIFO. fifo_out_avalon.v - Custom component that interface with the writing end of a FIFO. fifo_16bit.v -

15**A 16-bit wide, 256** element **FIFO**

generated by MegaWizard. fifo_16bit_SA.v -

15**A 16-bit wide, 256** element **FIFO,**

with show-ahead read, generated by MegaWizard. basic_receiver_16bit.v - Combines the two bytes received on the data[7:0] line into a single 16 bit word and store it into a FIFO. basic_sender_16bit.v - Gets a 16 bit word from the FIFO and sends it as two separate bytes. ? basic_comm.v - This module contains "basic_receiver_16bit.v", "basic_sender_16bit.v", and their two FIFO modules. In the project top level

module, this file connects to the FIFO interfaces "fifo_in_avalon.v" and "fifo_out_avalon.v". ? basic_comm_tb.sv - Testbench file. ModelSim only. Flash Memory The Flash memory on the DE2-70 board is used for testing purpose. Some tests use a long sequence of data that is stored on flash memory. The algorithm for reading from Flash is shown below. [11] The flash memory is 16 bit wide, and the NIOS 2 processor is 32 bit. So each NIOS 2 read is actually mapped to two consecutive 16 bit reads. The read algorithm graph shows there are four timing parameters just for read. Actually this is a simplification already. Flash memory has many timing parameters and a full ASIC style implementation, with a loadable register for each timing parameter, would be quite time consuming. Fortunately it's FPGA, so certain simplifications are assumed, and the timing information is hardcoded. The flash memory write algorithm is illustrated below. It doesn't actually write data to flash memory. Instead a "write" is a part of a command to the flash memory. A full flash memory command consists of a series of these "write" operations. Again taking advantage of the FPGA programmability, only three of the flash commands are implemented. A table of all flash commands are shown below, and only "write to buffer", "chip erase", and "sector erase" were implemented. [11] File List: Verilog Files: ? flash_avalon.v – implements flash memory read and write signaling ? flash_timing.xlsx – a worksheet to calculate the timing parameters used in "flash_avalon.v". The input values in the spread sheet include timing values from the datasheet (in ns) and system clock frequency. The spread sheet produce timing values that should be loaded into the various couters in the "flash_avalon.v" file. C Files: ? flash.h and flash.c – NIOS 2 code that implements a few of the flash commands. There are also test routines. C# File (for PC): ? Flash.cs – Contains test routines that copy data from PC to the FPGA, or vice versa. The code running on the FPGA end (NIOS 2) is very basic and does not kind of file system. Flash memory is divided into sectors, and this information is known only to the code running on the PC. I2C The WM8731 audio codec chip is configurable through I2C. A full I2C module is actually quite complicated, but taking advantage of FPGA re-programmability, only a subset of I2C protocol is implemented. Only the operations needed by the WM8731 chip have been implemented. [7] There are no I2C reads, only I2C writes. The I2C implementation is split into a basic module that handles the I2C signaling, and a higher level module that implements the I2C write operation. The basic I2C operations implemented by the lower level module are shown below, with each operation taking four states and the I2C module hard coded for 100kHz I2C operation. A higher level module implement the I2C write. The I2C write is a parallel to serial conversion process, with the higher level module calling the basic module multiple times to send out the data one bit at a time. File List: ? i2c_basic.v - this module executes four basic I2C operations: start condition, I2C write, I2C read, and stop condition. ? i2c_avalon.v - this module uses the i2c_basic to execute writes onto the I2C bus. There is no C file – The audio codec configuration functions in "audio.c" invoke IORD and IOWR macros directly to access the I2C peripheral module. SSRAM The SSRAM device has many control pins, and its datasheet shows many different read and write modes. [12] This is probably done so that it can accommodate as many different processors as possible. Reading the SSRAM follows the following steps and timing: ? ? ? ? At time t = 0, the FPGA launches the address. At time t = 1, the SSRAM chip latches the address. At time t = 2, the SSRAM chip launches the corresponding data. At time t = 3, the FPGA latches the data. The FPGA and the SSRAM share the same databus. When writing to the SSRAM, disable the "output enable" pin of the SSRAM and have FPGA drive the bus. The output enable pin (OE') is active low on this device. For the correct timing, set OE' to high, wait a cycle, then drive the bus. The idea is to give the SSRAM time to react and disengage from driving the bus. Constant Signals Byte Enable Signals are Constant: The SSRAM is always used in 32 bit mode - when reading, every read returns all 32 bit, and when writing, every write modifies all 32 bit. Therefore the various "byte enable" signals are always false. This applies to the pins: BWE', BWa', BWb', BWc', and BWd'. Certain Address Latch Pins are Constant: There is no good description of how the SSRAM works in the datasheet. It's best to read the block diagram directly. [12] The addressing portion of the diagram is shown below. The SSRAM can internally generate the lowest two bits of the address bus (labeled "A"). If this feature is not used, then the ADV' pin can always be high, which will disable the counter that generates the lowest two bits of the address. The ADSP' pin has been set to constant low, so that the address register enable pin will depend on just the ADSC' pin. Chip Enable is Constant The SSRAM data bus is only used by the FPGA and the SSRAM. There's no other memory using

this data bus. Therefore the SSRAM can be always enabled. There are three chip enable pins total: CE', CE2, and CE2'. File List: ssram.v - The SSRAM module is just this one file. There are no software drivers for memory access. There are two different versions of the "ssram.v". In the earlier project, the NIOS 2 processor is using the SSRAM. This one has some hacks in it to make it work with NIOS 2. In the later project, the SSRAM is used by the array processor, and no hacks are needed. First Approach: AEC Algorithm Acceleration using MAC and External FFT Block The first implementation of the acoustic echo cancelling (AEC) algorithm was done in fall 2010 semester. The associated files are in the "EE297A_AEC" folder. In this implementation, the AEC algorithm run as C code with hardware acceleration for two operations: FFT and MAC. This acceleration managed to produce quick results. But the resource usage and performance is poor. FFT Acceleration The AEC algorithm uses FFT numerous times. Altera has an FFT IP that can be instantiated via the MegaWizard. Code is then written to allow NIOS 2 to interface with this FFT wizard. Using this approach, it was not necessary to know how to do FFT. It was only necessary to know how to pass data into the Altera FFT block and retrieve the result. File List: Verilog Files ? fft.v - This is the FFT core generated by the MegaWizard ? fft_scaling.v - This is a scaling and saturation unit that is applied to the output of "fft.v". The "fft.v" output is 42 bits wide. When doing standard FFT, this 42 bit output needs to be saturated down to 32 bit. When doing inverse FFT, the Altera MegaWizard generated "fft.v" will not scale down by N. So if "inverse" signal is true, this module will scale down by N, as well as apply saturation. ? fft_avalon.v - This is the top module of the component. It contains the fft.v and fft_scaling.v modules. SystemVerilog Test Bench Files: The following test bench files are present in the ModelSim project folder only. ? ? fft_scaling_tb.sv - This is the test bench for "fft_scaling.v". fft_avalon_tb.sv - This is the test bench for "fft_avalon.v". NIOS 2 C Files ? "fpga_fft256.h" and "fpga_fft256.c"– The code to interact with the FFT module. There are two FFT and two inverse FFT calls. These four function alls are all that's needed by the AEC algorithm at "aec_proc.c". The four function calls are: ? fpga_fft256_real_input – FFT where the input is real value only. The output includes both real and imaginary values. ? fpga_fft256_128pt_real_input – FFT where the input is 128 points of real values only. The component then pads another 128 zeros. The output includes both real and imaginary values. ? fpga_ifft256_real_output – Inverse FFT where the input is 256 points with both real and imaginary components. Only 256 real points of the output will be read. ? fpga_ifft256 – General inverse FFT where both the input and output include both real and imaginary components. It's certainly possible to utilize the Altera FFT module with just a single FFT function call. The reason to have multiple functions is to speed up the process of moving data to the FFT and reading from it. For example, in the case where the FFT input values are real only, the underlying hardware will automatically supply a zero for the imaginary component. Usage Model The FFT module has four registers at offset addresses 0 through 3. The function of the registers are listed in the following table. Address Upper 16 bits Lower 16 bits 00 Real FIFO read/ write 01 Imaginary FIFO read/write bit[1] = soft reset bit 10 bit[0] = read returns 1 if FFT engine output is valid, 0 if output is not ready FFT Loading Modes bit[5] = 1 for inverse bit[4:3] = read mode bits 00 = read real values only 01 = read both real and imaginary values; reading from imaginary values will advance the FFT output 10 = ignore the remaining values from the FFT bit[2] = start a new write flag bit[1:0] = write mode bits 11 bit[19:18] = FFT source error 00 = load 256 real values, imaginary value = 0, then start FFT 01 = load 256 values in real, imaginary pairs, then start FFT 10 = load real values, imaginary value = 0 don't start FFT yet; the number of values to load is most likely 128, but nothing happens automatically at the end of the loading 11 - load 128 values, zero for both real and imaginary, then start FFT. In this mode bit[2] should be 1 in the beginning and go to 0 when the 128 zeros have been loaded into the FPGA. To do an FFT or inverse FFT, first set up the FFT's operation mode by writing to address 3. Then write the real input data to address 0, and write the imaginary input data to address 1. This step is done repeatedly until all data is written in. Poll address 2 to wait for the FFT engine to finish. Then read the output of the FFT engine by reading address 0 and address 1. MAC Acceleration The MAC acceleration is done by a specialized hardware that provides a 64 bit multiply accumulator, as well as barrel shifting capability. This hardware is implemented as a custom instruction with four sub instructions, N being 0 through 3. Internal Block Diagram Programming Information N Operation 00 Load zero into accumulator 01 10 11 Accumulation mode: load adder_out into accumulator, saturating the output as

necessary. Shift accumulator up (to the left) based on dataa[5:0] dataa[8] sets the rounding mode Shift accumulator down (to the right) based on dataa[5:0] dataa[8] sets the rounding mode The output "result" is always the lower 32 bits of the accumulator. File List ? my_64bit_shifter_part1.v and my_64bit_shifter_part2.v - two modules together forms a 64 bit shifter. The circuit is completely combination. ? ci_mac.v - MAC unit Operational Details The custom instruction uses a clock enable signal. So nothing happens or gets moved if the clock enable (clk_en) signal is low. acc register – 64 bit accumulator ? acc gets 0 if N = 00 ? acc gets adder_out, with saturation, if N = 01 ? acc gets shift_out, if N = 10 or 11. mult_out register – 64 bit register holding multiplier output ? "round" is a flag that gets set when N equals binary 10 or 11. ? mult_out gets the quantity (dataa * datab) if "round" flag = 0 ? mult_out gets the quantity (dataa * datab / 212) if "round" flag = 1. In this mode, the accumulator will take much longer to saturate. Usage Example int temp; ALT_CI_MAC64_INST(2, 0x100, 0); //truncating mode on temp = ALT_CI_MAC64_INST(0, 1000, 4000*4096); temp = ALT_CI_MAC64_INST(1, 2000, 3000*4096); temp = ALT_CI_MAC64_INST(1, 6000, 3000*4096); // temp = 4 million temp = ALT_CI_MAC64_INST(1, 0, 0); // temp = 10 million temp = ALT_CI_MAC64_INST(2, 256+4, 0); //temp = 28 million temp = ALT_CI_MAC64_INST(3, 256+6, 0); //temp = 448 million temp = ALT_CI_MAC64_INST(1, 0, 0); //temp = 7 million temp = ALT_CI_MAC64_INST(1, 0, 0); //temp = 7 million temp = ALT_CI_MAC64_INST(1, 0, 0); //temp = 7 million The arguments in the macro are (N, dataa, datab). The round bit is set while N equals binary 10 or 11. So the line "temp = ALT_CI_MAC64_INST(3, 256+6, 0);" will shift down by 6 (divide by 26) and set the round bit to 1. Note that there is a pipeline latency effect. So while accumulating values, it takes 2 instruction cycles for the input to affect the output. While shifting, it takes 1 instruction cycle to change the output. The test bench file (only in the ModelSim project folder) runs through the above sequence of operations as well. Control GUI The DE2-70 board in the telephone simulation is controlled from the PC using a program written in C#. This program is in the "AEC_PC2" folder. The program has various test commands: ? Command menu ? AEC from flash – This command instructs the FPGA board to run the AEC algorithm using input data stored in the flash memory. The FPGA will the compare the output

29**generated by the algorithm is** with **the** expected **output** stored on **the**

flash memory. The result of the comparison is reported back to the PC and will be displayed. ? The Flash tab – This tab has commands to load data onto the FPGA flash memory. The AEC algorithm acts of hundreds of thousands of data points, so testing is best done by loading the test vector onto the FPGA. ? The Audio tab – The "Play File" button tells the FPGA to play the voice data stored on the flash memory. The "Phone Test Start" button tells the FPGA to start acting like a phone. Data seen on the GPIO connector, which came from the other FPGA, will be passed to the speaker. Data seen on the microphone will be passed to the other FPGA via the GPIO connector. The "Phone Mode" can have the "AEC" algorithm on or off. Performance and Resource Utilization With the FFT and MAC accelerators, the AEC algorithm can operate on 16kHz data rate, using 79.4% of the processing time. The resource utilization is quite high, as shown in the table below. The top resource consuming entities are listed as well. Logic Cells Dedicated Logic Registers M4Ks (memory blocks) Multipliers 9x9 Multipliers 18x18 Total 14,822 10,021 146 16 70 FFT (Full) 8,600 6,991 30 16 64 FFT (Altera Core) 8,288 6,810 30 16 64 NIOS 2 (CPU) 2,950 1,771 58 2 On Chip Memory (Code and some data) 3 1 48 MAC 1,117 129 4 The main benefit to this algorithm acceleration approach has already been mentioned before – the quick implementation time. Regarding the FFT, it was not necessary to know how to do FFT. Instead, it was sufficient to just know how to move data into and out of the Altera FFT block. The MAC block is standard real number MAC, not a complex number MAC. So it's design was simple as well. Problems with this approach Problem #1: Inability to scale up to 32kHz data rate. The goal of the spring 2011 semester is to scale up to 32 kHz data rate. At this data rate, most operations will involve twice as much data and will take twice as long. The FFT operation will take more than twice as long. With this approach, the CPU is far above 50% loading. It's possible to increase the

MAC efficiency further by using a complex MAC – a MAC that is designed to work with complex numbers. But due to the time spent loading data into the MAC, this MAC will not be two times as fast before. A complex multiply operation ideally needs to take in four values at a time – two real numbers and two imaginary numbers. The NIOS 2 custom instruction interface is only 64 bit wide, so it cannot keep up with the complex MAC by feeding in a new pair of complex values every cycle. Even with ideal pipeline conditions, NIOS 2 will need two cycles to load a pair of complex numbers into the complex MAC, and the complex MAC will spend just one cycle multiplying those two complex numbers. Without a way to double the processing speed, operation on 32kHz voice data is impossible. Problem #2: High FFT resource use and low FFT cycle use. The FFT single handedly takes up most of the multipliers, and large quantity of the memory, but only a small fraction of time is spent on the FFT module. The Altera FFT is a radix-4 FFT module that can do a 256 point FFT in 256 clock cycles. The AEC algorithm does 35 FFT per frame of voice data. So the total number of cycles actually spent doing FFT is 35 * 256 = 8,960 cycles. There are 125 data frames per second, and the system runs on a 95 MHz clock. The total number of cycles available per data frame is then 95e6 / 125 = 760,000. FFT operation therefore is taking up just 1.2% of the total number of system clock cycles. Moving data in and out of the FFT module was actually taking much more time than the FFT operation itself. Problem #3: High onchip memory utilization. The onchip memory use is already 73kB. This probably won't double for the 32kHz scenario though because most of the data memory is in SSRAM already. The real problem is that the implementation is loading CPU around 80% of the time. Moving data to off chip memory will slow down the CPU further, making even 16kHz operation marginal. So many things are placed on the onchip memory, just for the sake of speed. Second Approach: AEC Algorithm Implementation using ASIP Speed versus Area, and the ASIP approach In digital design, the main trade off is the speed versus area trade off. In this embedded system, the voice data rate establishes a limit on the speed. The cost optimal design is something that can meet just the 32 kHz data rate. It should not run too much faster than 32 kHz, because that will mean unnecessary area. The smallest area design approach also means reusing resources to the greatest extent possible. The correct design strategy to facilitate design reuse is determined by the clock per data metric. On the DE2-70 board's Cyclone 2, past experience indicate that once operations are pipelined, a clock frequency of 100MHz is possible. On newer Cyclone FPGAs, this frequency can be slightly higher. On more expensive FPGAs, the operation frequency can be significantly higher. Altera's own benchmark for NIOS 2 records the following potential frequencies: FPGA / Platform Cyclone II Cyclone III Arria II GX Stratix IV HardCopy IV NIOS 2/f Frequency 140 MHz 175 MHz 240 MHz 290 MHz 305 MHz From the chart, it can be seen that an assumption of 100 MHz clock frequency is actually quit conservative, especially if ASIC conversion is a future possibility. Even at 100 MHz clock, the clock per data is 100 MHz clock / 32 kHz data, leading to a result of 3125 clock per data. In addition, the AEC algorithm in this case is a block based algorithm. Meaning the data is computed on a per block basis, so the more correct metric is actually 3125 * 256 clock per 256 data, leading to a metric of 800,000 clock per block. A circuit that produce one data every few hundred cycle maybe driven by a state machine, although that state machine might be quite complicated. Whether it's 3125 clock per data, or 800,000 clock per block, the solution to run the AEC algorithm should be processor based. Yet on an FPGA, the NIOS 2 processor cannot run this algorithm. Accelerating a processor to be able to run a certain algorithm is the classic goal of ASIP, which stands for Application Specific Instruction-set Processor. Indeed, the ASIP company Tensilica just announced an ASIP for HD audio processing three months ago, specifically to run: "complex algorithms, including Wolfson's class-leading transmit noise cancellation (Tx), acoustic echo cancellation (AEC) - which both deliver up to 32dB of noise cancellation capability - and beam-forming solutions for VoIP applications, dramatically enhance the audio call experience." [13] The strategy used to build an ASIP on the Cyclone 2 FPGA is to start with the NIOS 2 processor. As a 32-bit RISC processor, the NIOS 2 is abysmal at: ? Operations wider than 32 bit ? Operations that has more than one output. For example, the FFT butterfly is a two input, two output operation. ? Operation involving arrays and circular buffer addressing due to the lack of special purpose addressing registers. The ASIP is created by building a processor with opposite characteristics to the NIOS 2 processor, and merging the two. First the AEC algorithm is ported to C# and re-examined to derive the array processor design requirements. This step also yields a software model for

the array processor, which is vital for later testing. Next the array processor is coded and tested in ModelSim. Then the array processor is extensively tested by issuing instructions to both the FPGA version of the array processor and the software C# model. Finally the AEC algorithm is run using the array processor. Algorithm Prototyping in C# This project is mainly concerned with the implementation of the AEC algorithm. But certain basic level of algorithm level testing and support needs to be met. The goal of the algorithm prototyping stage is to provide an environment for testing the algorithm, as well as to identify the accelerations needed by the array processor. The AEC algorithm is restructured with array operations separated into a separate object. This separate object, the "ArrayProc", later becomes the software model of the array processor. The "ArrayProc" object defines operations that the array processor needs to support in order to run the AEC algorithm. This object is also

32**used to test the** FPGA implementation **of the** array processor. **The**

C# files associated with all this are in the "AEC_Algo" folder. AEC Algorithm Testing Environment The AEC_Algo C# project provides a bare minimum testing environment for the AEC algorithm with the following features: WAV file support, random vector support, bit width surveys, graphical attenuation feedback, and file comparison utility. WAV File Support: Prior to the work in this stage, there seems to be only one input test vector, which is in Korean. This input vector is a simple binary file, which is a problem in the long run since binary files don't contain information about the bit width or data rate. A wav file should be the minimum standard for storing test vectors. The AEC Algo project supports wav files through the class "MyWavFile", in the file "MyWavFile.cs". Only a bare minimum set of features is supported. The code is based on the information from the website "WAVE PCM soundfile format". [14] On a PC running Windows XP, the Sound Recorder software that is bundled with Windows can be used to produce wav files from voice recordings. On Vista, the wav format is no longer supported by the Sound Recorder software. In that case, the free software Audacity located at http://audacity.sourceforge.net/ can be used to record wav files. Random Vector Support: The "Data" tab of the software has features to support the generation of random test vectors. Having a large number of test vectors is important for thorough testing. x[n] creation method – x[n] is the far end speaker. The software supports the creation of random waveforms at different ranges. x[n] basis file for d[n] creation – The d[n] is the near end speaker, plus the echo of the far end speaker. The "basis file" refers to the source of the echo. d[n] creation method – This combo box determines the method with which to generate the echo. In a triple echo, the data from the "basis file" is attenuated and delayed to produce three mini echoes, and then combined to produce the triple echo. d[n] interference - The d[n] is the near end speaker, plus the echo of the far end speaker. The "interference file" refers to the speech of the near end speaker. The final d[n] file is the sum o the "interference file" plus the echo created using the data found in the "basis file". d[n] file name – The name of the d[n] file that will be created when the "Make d[n]" button is pressed. Bit Width Surveys: The original AEC algorithm is in double precision floating point. The implementation will be in fixed point. The question is how many bits are necessary for a functional algorithm. The bit width question is a dynamic range question. The dynamic range of any particular value can be obtained by taking it's logarithm. The project has a "Histogram" object in the "Histogram.cs" file that collects the logarithm information. The following is the key trigger code in the use of the histogram. An example histogram output is shown below. In this particular case a variable called "Wei" is collected and its logarithms are tabulated. This is a tab delimited output that is Excel friendly. <-32 -31 -29 -27 -25 -23 -21 -19 -17 -15 -13 -11 -9 -7 -5 -3 -1 1 "Wei" Count 700000 600000 500000 400000 300000 Count 200000 100000 0 In this case, the chart is implying that 18 bit precision maybe all that is needed. But ultimately, my algorithm understanding is deemed insufficient and I chose a conservative precision plan: In this plan, scalar data gets floating point precision. The array data will be stored in fixed point Q31 format. While in the array processor, the array data will be stored in Q7.40 format. Large resource savings are possible if a more aggressive precision plan can be adopted. But this is beyond the scope of this project. Graphical Attenuation

Feedback: The "Graph" tab has a chart that displays the attenuation between output[n] and d[n]. File Conversion and Comparison Utility: The original floating point, double precision, AEC algorithm is ported in a format suitable for array processor execution through multiple stages. The file comparison utility is used to ensure that this porting process does not generate errors. The final version meant for FPGA implementation produce the same output as the original version, save for small differences due to precision issues. The "Util" tab also has a "PCM to Wav" button that converts a binary (raw) audio file to the Wav format. These are all just bare minimum features: In no way does the AEC_Algo project constitute any sort of satisfactory algorithm level testing platform. It's merely to gather the minimum knowledge to implement the algorithm. Without wav file support there's no easy way to play the binary files. Without random vector support there's no test vector for long term testing. Without bit width survey there's no clue to what kind of fixed point system to use. Without graphical plot of the attenuation there's no way to know how well the algorithm is working. Without a file comparison utility there's no way to know much inaccuracy the fixed point representation created. If there's time, all of these features need further expansion. Algorithm Reformatting The AEC algorithm is provided as a typical C program, which needs to be reformatted into a syntax that's more suitable for an array processor execution. This reformatting should be done on the desktop, since debugging on the desktop is easier than debugging on the FPGA board. This algorithm transformation occurs in several stages. Stage 1: AEC32_FPGA_port1.cs This port focus on re-organizing the code order to better suit FPGA execution. The idea is to group NIOS 2 instructions separately from the array processor instructions, with the goal of avoiding situations where one processor has to wait for another. Pictorially this concept is shown below: There are other improvements as well. The Xb and Eb memories are no longer being shifted around. A circular buffer scheme is used instead. This is shown below: The stage 1 port is still using double precision, so its output should match the original algorithm's output exactly. Stage 2: AEC32_FPGA_port2.cs This is part 2 of the FPGA port. This port focus on changing the double precision into "int" and "float". The intermediate calculations often use double, for example during the FFT, but the storage values are in "int" and "float". The output of this port will differ from port1 output slightly. Stages 3 and 4: AEC32_FPGA_port3.cs and AEC32_FPGA_port4.cs These stages focus on moving the array processor portion of the code to a separate source file called the "ArrayProc.cs". The AEC32_FPGA_port4.cs code is intended to run on the NIOS 2 processor, while the ArrayProc.cs code is intended to run on the array processor. An example of this code separation is shown below: Note that the memory needs to be manually managed. In this example, the Xc array is spread over the buffers B0 and B1. The Xb two dimensional array needs to be mapped to specific addresses in the SSRAM. The separation of code into the "ArrayProc" module also allowed the identification of design requirements for the array processor. Eventually, a modified ArrayProc object will play a key role in the array processor testing as well. Basic Array Processor Structure and Key Attributes A basic array processor is presented below to illustrate the overall structure and key attributes. On the left side there is a list of four memory buffers. All buffers have the same read address (rdaddress) applied to them. A multiplexer with a selection line called "sel" chooses the output that will go into the ALU. In this example, the ALU just invert all the bits and produces an output "Y". The same Y and the same write address is fed to multiple memories. Each memory has its own write enable pin, which determines whether the value on Y will be written into the memory. Timing The array processor datapath is pipelined for high frequency operation. The timing of the simple array processor presented above is shown below: The read address launched at time t = 0. For high speed operation, memories have registered read address input and data output. It takes one cycle to latch the incoming read address, and a second cycle to place the data on the output. So the memory data output is launched at time t = 2. The multiplexer registers its output as well, so the multiplexer launch the selected memory's data at time t=3. The ALU contributes another cycle of delay. Taken altogether, the read address launched at time t=0 produces an output at time t=4. Due to this time delay, the write address that corresponds to the read address has to be launched at time t=4 as well. The launch of the write enable signal likewise has to be delayed for t=4 cycles. An example of this write address and write enable delay is shown below: Attributes ? One Address Feeding Multiple Memories – Each address is generated by an independent counter. To conserve resources, as few addresses should be used as possible. In the AEC array processor, the most complex operation is the FFT.

This operation requires one read address, and two write addresses. ? Selection and Write Enable Pins – In the array processor, a smaller number of addresses are fed to a larger number of memories. On the ALU input side, multiplexer selection pins chooses the input that will be used. On the ALU output side, write enable pins choose the memory or memories that will actually record the ALU output. ? Non-looping, CISC, and Pipelining – The array processor is the opposite of the NIOS 2 processor. It is expected to execute instructions in sequence and does not loop. The instructions are expected to be algorithm specific. For certain simple instructions, the pipeline through the ALU is expected to be just one cycle. But for other algorithm specific instructions, the pipelining through the CPU is expected to much longer so to achieve maximum throughput. The control scheme must be ready to deal with a wide variety of pipeline delays. AEC Array Processor Multiplier Design: Two types of multipliers are used in the array processor, a 48 bit times 48 bit multiplier and a 32 bit times 48 bit multiplier. The building block multiplier on the Altera Cyclone 2 FPGA is the 18 bit times 18 bit multiplier. The breakdown into the basic multiplier occurs as shown in the following diagrams: The multipliers are pipelined into three stages. The first stage computes the various products. The second and third stage is an adder tree that sums the products into the final answer. The math used in the ALU is Q7.40. In this system, the multiplier "cz" is not contributing much to the final result and so is not instantiated. This decreases the resource requirement of the 48x48 multiplier. Testing: The multiplier is tested by reading a binary file containing random numbers to be multiplied, computing the multiplication, and then comparing the results to another computer generated binary file. These binary files are generated in C#. Although the bit-width is 48 bit, Microsoft's library has a "BigInteger" class that can handle extremely large numbers. The results are computed using this "BigInteger" class and stored as a binary file. Files: Design Files for the multipliers are in the "MyMult" folder of the ModelSim project. ? ? ? ? MyMult.v – contains several multiplier designs, some pipelined, some are not. This file includes designs that are not used in this project. MyMult_tb.sv – Testbench for the multipliers declared in "MyMult.v". MyMult.do – ModelSim script file that sets up the waveform window and starts the test bench. MyMult32P_in.bin, MyMult32P_out.bin, MyMult40_in.bin, MyMult40_out.bin, MyMult48_in.bin, MyMult48_out.bin – Various input and output binary files, for use with the testbench. The "_out.bin" files are generated using Microsoft .Net library's "BigInteger" class. ALU A diagram of the ALU is shown below: The ALU has two outputs, Y0 and Y1. The simpler one cycle operations are placed into the "by pass logic" block in the lower left hand corner. The chart is about the ALU's longer and more complex operations. The precision is shown for each stage of these longer operations. The vertical bars denote the pipeline stages. ALU Operations mode[3:0] Effect on output Y0 0 Zero: Y0 = 0 and Y1 = 0 FFT: 1 Y0 = X0 + X1*T ; Y1 = X0 - X1*T The "FFT_shift" determines what bit shift will take place 2 Y0 = accumulator (acc) Y0 = rb = mu * T * X1 * (1 << C0) 3 The value of C0[] shift value is 8 to 16, all left shifts E Copy and round to Q31: Y0 = X0 rounded and saturated to Q31 F Y0 = X0 (for bit reverse instruction) mode[3:0] 0 Zero: Y0 = 0 and Y1 = 0 1 Effect on output Y1 FFT: Y0 = X0 + X1*T ; Y1 = X0 - X1*T The "FFT_shift" determines what bit shift will take place 2 Simple copy: Y1 = X0 3 Copy real component only: Y1 = {X0.real, 0} 4 Copy and swap real and imaginary component: Y1 = {X0.imag, X0.real} 5 Copy and conjugate: Y1 = {X0.real, -1* X0.imag} 6 7 E F FFT Shift: Copy and round to Q31: Y1 = X0 rounded and saturated to Q31 This was added as an instruction first - but later operation E is added - so that operation 6 is now redundant. But the C# test bench already expect operation 6, so operation 6 remains for now. Y1 = X0 << 16. There's no saturation check. This is for aligning 16 bit voice data to 32 bit. Copy and round to Q31: Y1 = X1 rounded and saturated to Q31 Y1 = X1 (for bit reverse instruction) During the FFT operation, a shift to the right or left can be done. This feature is added because in the AEC algorithm provided, each FFT butterfly is accompanied by a one bit shift to the right as a mean of preventing overflow. When the algorithm is doing inverse FFT, the algorithm does not do any division by the array length. fft_shift[1:0] Description 00 No shift: Y0 = X0 + X1*T ; Y1 = X0 - X1*T 01 11 Shift to the left by 1 bit:

27**Y0 = 2 * (X0 + X1*** T) ; **Y1 = 2 * (X0 - X1***

T) Shift to the right by 1 bit Y0 = (X0 + X1*T) / 2 ; Y1 = (X0 - X1*T) / 2 "P_int_k" Function: In the ALU there's a "P_int_k" block that is between the accumulator register (acc) and the p_int_k register (p_int_k). The name "p_int_k" comes from the code in the AEC algorithm. This function is related to logarithm and has the input / output table shown below. The "?" denotes a don't care condition on a particular bit. Other ALU Control Lines: input[9:2] meaning output[4:0] 1???_???? >= 512 5'd8 01??_???? >= 256, < 512 5'd10 ?? 001? _???? >= 128, < 256 5'd10 0001_???? >= 64, < 128 5'd11 0000_1??? >= 32, < 64 5'd12 0000_01?? >= 16, < 32 5'd13 0000_001? >= 8, <16 5'd14 0000_0001 >= 4, < 8 5'd15 0000_0000 < 4 5'd16 Control Lines mu_load c0_wren acc_clear Description When this signal is high, copy the output content of the instruction FIFO into the "mu" register When this signal is high, copy the output of the "Pi_int_k" shift block into the C0 memory When this signal is high, the accumulator (acc) clears to zero. When this signal is low, the accumulator will add the complex value (p_r, p_i) to its current value. Possible precision issue: The "shift by C0" step shifts the value "rb_pre_shift" to the left by 8 to 16 bits. Full accuracy requires a Q7.56 in "rb_pre_shift", so that after shifting 16 bits, the decimal point still has 40 digits. The problem here is that the 56 bits in "rb_pre_shift" are not all accurate, because it's the product of Q14.40 and a Q31 value. Saturation check for 1000...00 Only the saturation module feeding the y0_rb checks for the 1000...00 type number. At that particular location, such number is quite possible due to shifting in 8 to 16 zeros. This check is not done in any of the other saturation units in order to save resources. Testing: In the ModelSim project, the testing of the ALU is done by a directed test bench on just a few cases for each ALU mode. The test bench has assertions in it so that it's a self-checking test. Files: The ALU files are in the ALU folder of the ModelSim project. ? ? ? APU_ALU.v – This file has the APU_ALU module, plus a shift module and a saturation module that the ALU instantiates. APU_ALU_tb.sv – This file is the test bench of the ALU. APU_ALU.do – A ModelSim script that sets up the waveform window and runs the test bench. Datapath The AEC array processor works on same principles as the simple array processor presented in another section. Inputs to the ALU: The "X1" and "X2" are must choose from

30**A0, A1, B0, and B1 in order to**

do the FFT. The complex multiplier inside the ALU is hard wired to "X1" and "T". The inputs feeding

31**"T" is determined by the** needs **of the** algorithm. **The**

"T_q" is the twiddle factor for the FFT. The "ONE" is a way to disable the complex multiplier, so to add two arrays via X1 + X0 using the FFT datapath. The "A0_q" is to do multiply accumulate where the sum of products A0 * A1 is to be computed. The "A0_q conjugate" is to do power computation, where the power is A0 * (A0 conjugate). Address Feeding the Memories A0, A1, B0, B1: There is one read address and two write addresses serving the memories. This is required by the FFT operation. Memory A0 rdaddr wraddr0 A1 rdaddr wraddr1 B0 rdaddr wraddr0 B1 rdaddr wraddr1 Input to the Memories

24**A0, A1, B0, B1:** The **A0, A1, B0,** and **B1**

are all complex numbers, divided into real and imaginary components. Each memory can be written with data from Y0 and Y1. Furthermore, the AEC algorithm involves SSRAM storage, so all four memories can get data from the SSRAM as well. The SSRAM is 32 bit wide, so each copy from the SSRAM produce only the real part or the imaginary part. The "instr_fifo" stands for instruction FIFO. The NIOS 2 passes data to the AEC through this FIFO. Due to the NIOS 2 being a 32 bit processor, this interface is also 32 bit wide. The data from the 32 bit SSRAM and NIOS 2 FIFO are always in Q31 format. The data in the memory

buffers A0, A1, B0, and B1 are always in Q7.40 format. SSRAM Input: The SSRAM input comes from the ALU Y1 output. The Y1 is a complex number, so the ssram_sel chooses whether the real or imaginary component will be presented to the SSRAM interface's "data" line. Result FIFO Input: Data to be passed back to the NIOS 2 are placed into the "Result_FIFO" structure. This FIFO takes data from the ALU Y1 output. On the other end of the FIFO is NIOS 2. Similar to the SSRAM situation, there's a "result_sel" selection line to choose whether the real or imaginary component of the ALU Y1 output will go into the FIFO. Files: The datapath design files are in the "Datapath" folder of the NIOS 2 project. ? ? ? ? Datapath.v – Contains the "Datapath" module that instantiates the ALU, the various memories, the SSRAM interface module, and connects them together. ssram.v – The SSRAM interface module. ssram_tb.sv – A test bench for the SSRAM interface module. The real external SSRAM is 2 MByte. In this test bench, the external SSRAM is modeled as a 2048x32 bit memory in the "ssram_mod" module. ssram.do – A ModelSim testbench that sets up the waveform window and runs the SSRAM test bench. Overall Control The control uses a two layer scheme. The upper layer control module, called "AP_Control", generates control signals that are constant throughout the array processing operation. A middle layer will generate the variable signals during the array operation. The main variable signals are the read and write addresses that are feeding the memories. A graphical representation of this scheme is shown below. Run Signals The datapath is pipelined so that different signals have to arrive at different times. For example, during the FFT operation, the memory read address happens nine clock cycles ahead of the corresponding memory write address. To implement this, the read and write address generators both have their own run signals. Then during the FFT operation, the write address generator run signal will go high nine cycles after the read address generator run signal. The various run signals are red in the block diagram. An example of how the run signals relate to the pipeline timing is shown below. SSRAM Access The array processor control block (AP_Control) configures the SSRAM address generator (SSRAM_AG). The address generator provides the address, as well as the "ssram_sel" pin, which selects from either Y1.real or Y1.imag. The AP_Control block sets the SSRAM control pins. ssram_read ssram_write Reading Mode 1 0 SSRAM is driving the databus Writing Mode 0 1 FPGA SSRAM interface module is driving the databus SSRAM Disabled Neither FPGA SSRAM module nor SSRAM is 0 0 Mode driving databus When going from the reading mode to writing mode, or vice versa, go to the "disabled mode" as an intermediate mode. The idea is that under reading mode, the SSRAM chip is driving the data bus. If the interface goes to writing mode immediately, there would be a period of time when both the SSRAM chip and the FPGA is driving the data bus. Going to the disabled mode gives time for the SSRAM to disengage from driving the data bus. This is to comply with SSRAM device timing. Memory Write The block diagram above only shows control of memory A0r. The control of other memories are highly similar, including memories A0i, A1r, A1i, B0r, B0i, B1r, and B1i. wraddress0 – This is generated by write address generator #0 (Write_AG0). It's the write address for the memories A0i, A0r, B0i, and B0r. wraddress1 – This signal is not shown in the block is generated by write address generator #1 (Write_AG1). It's the write address for the memories A1i, A1r, B1i, and B1r. A0_sel[0] – These signals select the data that feeds the A0 memory's writedata lines. For many operations, this signal will be constant, so it will take on the value provided by AP_Control, which is the "A0_sel0_ctrl" in the block diagram. In FFT mode, this signal have to change depending on the FFT stage, so it will be under the address generator's control. B0_sel[0], A1_sel[0], and B1_sel[0] – Similar to A0_sel[0] in that these signals during FFT operation these signals will be under the control of the address generator, but they will follow the value set by AP_Control at other times. A0r_wren and the Wren_toggle block - The write enable (wren) signals are constant for most, but needs to toggle for SSRAM read operations. The SSRAM is 32 bit and each SSRAM read produces either the real or imaginary component of a complex number. So half of the time the SSRAM read feeds the A0r memory, and the other half of the time the read feeds the A0i memory. The "Wren_toggle" blocks are also blocks that remember which memory will have write enable go high, while the AP_Control block determines when to do it. During the instruction decode phase, the "Wren_toggle" block is set up. When run signal (write_ag_run) goes high, only the right "Wren_toggle" block will exert the write enable signal. For example, if the ALU output is meant for the memory A0i only, then when the run signal (write_ag_run) goes high, only the write enable signal for A0i should go high. This is illustrated below. Memory Read rdaddress – A single read

address is applied to all memory blocks. In the block diagram only A0r is shown, but the read address also applies to A0i, A1r, A1i, B0r, B0i, B1r, and B0i. x0_sel[0] and x1_sel[0] – These signals are constant for most array operations, but they vary during the FFT operations. When x0_sel[0] and x1_sel[0] are constant, they will take on values provided by AP_Control. When running FFT, these signals are under address generator control. State Machine The state machine of AP_Control executes in numerical order. A summary table of the states is shown below. IDLE (0) DECODE0 (1), DECODE1(2), DECODE2, DECODE3, … START0 (10), START1(11), START2, START3, … ARRAYING (30) … FLUSH1 (41), FLUSH2 (42), FLUSH3, FLUSH4, … Idle, the array processor is not running any instruction. Control signals are reset to their defaults. The array processor is decoding instructions. This is where the address generators are set up. The array processor is starting the array operation. The read address generator "run" signal will go high at the START0 state. Depending on the pipeline delay needed, the write address generator "run" signal will go high at a later START state. The array processor is operating in steady state. The signals from AP_Control are constant. The array processor remain in this state until the "length_counter" variable overflows. At the last clock cycle in the ARRAYING state, the read address generator "run" signal will go low. The datapath pipeline flushes. The write address generator "run" signal will go low during one of the FLUSH states. END The end of operating an instruction. Control signals are reset to their defaults. An example of array processing, relative to the states, is shown below. Files: The APU and control files are in the APU folder of the ModelSim project directory. ? ? AP_Control.v – This file has the AP_Control module, which is the top layer control module. APU.v – This file represents the APU. It mainly instantiates the AP_Control module and the Datapath module, then connect the two. FFT Addressing The radix-2 FFT uses a butterfly structure that takes two inputs and creates two output. The data needs to be positioned such that the butterfly takes inputs from two memory elements, and sends outputs to two memory elements. There must not be a situation where the butterfly wants two inputs from the same memory, or has to write two outputs to the same memory. 8 Point FFT Example: Figuring out a way to line data up for the 8-point FFT is sufficient to illustrate a general addressing algorithm. Note that with this algorithm, the data processing is not done in place. So there will be one set of memory holding the data being read, and a separate set of memory holding the data being written. FFT Addressing when Reading The address value when reading from the memory is sequential counting, from 0 to 7 in this case. The butterfly will alternate where it get its input. This alternating depends on the FFT stage. FFT Addressing when Writing The Mem0 address is sequential. There is alternation of where to put the output. The alternating pattern is related to the FFT stage. The Mem1 address inverts a bit of the sequential address. For example, in the second FFT stage, the Mem1 address is 2, 3, 0, 1. This is the sequential address 0, 1, 2, 3, with the second LSB inverted, as illustrated below: Address Generators Features Linear Bit Inversed 0 000 010 2 1 001 011 3 2 010 000 0 3 011 001 1 Not all address generator can generate all sequences. The feature present on each address generator is determined by the algorithm necessity. This is an attempt to save resources by having only the necessary features. In hindsight, the amount of resource saved is probably not worth the effort spent in a detailed classification of the address generators. Future attempt at building an array processor should define several classes of complexity for the address generators, and then just use the class that has the necessary address modes. Read_AG Write_AG SSRAM_AG C0_AG T_AG Variable starting value X XX X Decrement X Wrap around X Variable skip X Half speed X X 56 FFT read FFT write FFT twiddle Bit reverse X X X X Feature Descriptions Starting values - The address generator starting address is a variable. Decrement - The address sequence can decrement. Wrap around - The address sequence can wrap around once it reach a variable value. This is for implementing circular buffers. Skip - The address sequence can increment / decrement by a variable amount, instead of one. Half speed - Increment the address once every two clock cycles to work with the SSRAM. The first cycle is for the real value, the second cycle is for the imaginary value. FFT read - The address generator can produce a sequence used for FFT reading. FFT write - The address generator can produce a sequence used for FFT writing. The two FFT input memories, memory 0 and memory 1, have different write address patterns. So there are two write address generators, Write_AG0 and Write_AG1. Address Generator Timing The address generator timing scheme is shown below. The initial conditions are in red. The AP_Control state machine will set the read address generator

"run" signal to low when the "length_counter" overflows to 512. The actual address will overshoot to 106 – this is expected. The write will be applied only up to address 105 though. Note the "length_counter" initial value is "512 – length", where length is 6 in the case shown in the diagram above. Twiddle_AG Twiddle_AG is responsible for generating the twiddle factor address. Read_AG FFT Stage Twiddle Address 0 0 1 0, 256, 0, 256, ... 2 0, 128, 256, ... 3 0, 64, 128, ... 4 0, 32, 64, 96, ... 5 0, 16, 32, 48, 64,... 6 0, 8, 16, 24, 32 7 0, 4, 8, 12, ... 8 0, 2, 4, 6, 8, ... 9 0, 1, 2, 3, ... The read address generator is mainly sequential only. It has a "load" input that can be used to initialize its starting value. The outputs "x0_sel0" and "x1_sel0" control the ALU input multiplexer. They choose which memory will feed into the ALU. During FFT these lines are under the control of the address generator Read_AG. During non FFT operations they follow the value provided by AP_Control. Furthermore the outputs "x0_sel0" and "x1_sel0" needs to be delayed relative to the read address. The reason for this delay is illustrated in a previous section titled "Basic Array Processor Structure and Key Attributes". The behavior of the "x0_sel0" and "x1_sel0" are summarized below. FFT Stage 0 1 2 x0_sel0_delay x0_sel0_delay <= x0_sel0_ctrl; x1_sel0_delay <= x1_sel0_ctrl; x0_sel0_delay <= rdaddress [0]; x1_sel0_delay <= ~rdaddress[0]; x0_sel0_delay <= rdaddress[1]; x1_sel0_delay <= ~rdaddress[1]; 3 4 5 6 7 8 9 default: x0_sel0_delay <= rdaddress[2]; x1_sel0_delay <= ~rdaddress[2]; x0_sel0_delay <= rdaddress[3]; x1_sel0_delay <= ~rdaddress[3]; x0_sel0_delay <= rdaddress[4]; x1_sel0_delay <= ~rdaddress[4]; x0_sel0_delay <= rdaddress[5]; x1_sel0_delay <= ~rdaddress[5]; x0_sel0_delay <= rdaddress[6]; x1_sel0_delay <= ~rdaddress[6]; x0_sel0_delay <= rdaddress[7]; x1_sel0_delay <= ~rdaddress[7]; x0_sel0_delay <= rdaddress[8]; x1_sel0_delay <= ~rdaddress[8]; x0_sel0_delay <= x0_sel0_ctrl; x1_sel0_delay <= x1_sel0_ctrl; Write_AG0 and Write_AG1 There are two write address generators, Write_AG0 and Write_AG1, due to the difference in write address generation for the FFT operation. The FFT butterfly takes two values at a time, so the number of bits involved in the address is actually half of the total FFT length. For example, a 32 point FFT would store just 16 values in each buffer. The address involved in this case is 4 bit (0 to 15) rather than 5 bit (0 to 31). The "bit_reverse[2:0]" input determines kind of address bit reverse that will occur. bit_reverse[2:0] Description 1 Reverse wraddress[3:0] - for 32 value FFT 2 Reverse wraddress[4:0] - for 64 value FFT 3 Reverse wraddress[5:0] - for 128 value FFT 4 Reverse wraddress[6:0] - for 256 value FFT 5 Reverse wraddress[7:0] - for 512 value FFT 6 Reverse wraddress[8:0] - for 1024 value FFT The FFT values are spread across two memory arrays. Data arrangement prior to FFT and after FFT:

6**Address 0 1 2 3 4 5 6** 78 9 **A B C D E F**

Mem0 0 1 2 3 4 56

4**7 8 9 A B C D E F** Mem1 **10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F**

Being in Mem0 adds a prefix of zero to the full address. Being in Mem1 adds a prefix of one to the full address. The location of "3" is 0x03. The location of "15" is 0x15. Data format after bit reverse stage:

6**Address 0 1 2 3 4 5 6**

5**7 8 9 A B C** DE **F** Mem0 **0 8 4 C 2 A 6 E 1 9 5 D 3 B 7 F**

Mem1

9**10 18 14 1C 12 1A 16 1E 11 19 15 1D 13 1B 17 1F**

Being in Mem0 adds a suffix of zero to the full address. Being in Mem1 adds a suffix of one to the full address. The location of "4" is binary "00100" and the location of "14" is binary "00101". The "fft_stage[3:0]" input effects the following: ? ? Mem0_sel0 for Write_AG0. The "wraddress" of set 0 memory just increments linearly during FFT. wraddress and Mem1_sel0 for Write_AG1 For the write address, the last FFT stage should be 0xF - meaning the last FFT stage should not use special encoding. In summary: FFT Length FFT Stage 32 4 0xF 0x4 128 4 0x4 0x4 The write address generator during the final pass is a special case that uses linearly address that is incrementing by 1. The source of the writedata line (the Mem0_sel0 and Mem1_sel0 control lines) should be constant throughout the final FFT stage. Note that the read side still needs the usual "fft_stage" value. So the control path and the instruction encoding has to make provision for separate "fft_stage" numbers to the read and write sides. Wren_Toggle The Wren_Toggle module stores a mode value that determines how the write enable (wren) pin will act once the AP_Control tells the write address generator to "run". The mode values are listed below.

20**Mode[2:0] 0 1 2 3 4** Description **OFF**

wren_r and wren_i will both be zero even if "run" is 1 Real Only wren_r will be 1 when "run" is 1 Imaginary Only wren_i will be 1 when "run" is 1 Both Real and Imaginary wren_i and wren_r will both be 1 when "run" is 1 Toggle The first output will be wren_r=1, wren_i=0. Second output will be wren_r=0, wren_i=1. So effectively: ? ? ? mode[2] = toggle flag mode[0] = real on mode[1] = imag on SSRAM_AG The SSRAM address generator (SSRAM_AG) supports circular buffer addressing, so it has a number of internal registers. Register loading occurs with "load" = 1. load_mode[1:0] Action Performed rdaddress <= address_in; skip <= 19'd1; 0 highest_addr <= 19'H7_FFFF; block_size <= 19'H7_FFFF; 1 skip <= address_in; 2 highest_addr <= address_in; 3 block_size <= address_in; The counting code shows how the registers are used: assign next_addr = {1'b0, rdaddress} + {skip[N-1], skip}; if(next_addr > {1'b0, highest_addr}) rdaddress <= rdaddress - block_size; else rdaddress <= next_addr[N-1:0]; The SSRAM_AG also has a "complex" input pin that should be toggled when storing a complex number into the SSRAM. complex ssram_sel 0 ssram_sel = rdaddress[0] 1 ssram_sel = ssram_sel_ctrl This "rdaddress" is an internal counter to SSRAM_AG. It is not the read address generator output. Files: The address generator design files are in the "AG" folder of the ModelSim project. ? ? ? AG.v – Contains various address generators used in the AEC array processor project. AG_tb.sv – Address generator test benches. The correctness checking is meant to be done visually. AG.do – ModelSim script to set up the waveform window and start the AG test benches. Array Processor Interface with NIOS 2 The NIOS 2 interface with the "Instr_Filter" through a custom instruction interface. The "Instr_Filter" will pass instructions to the array processor unit (APU) as appropriate. The NIOS 2 passes instruction into the instruction FIFO (Instr_FIFO), which the APU reads. The APU return data to NIOS 2 via the result FIFO (Result_FIFO). Instructions that NIOS 2 send to the Instr_Filter fall into two groups. One group of instructions are about the status of the array processor and the status of the interface. These instructions are handled locally by the Instr_Filter block. Another group of instructions are passed into the array processor via the Instr_FIFO. The APU and NIOS 2 runs in separate clock domains, as shown. This arrangement should enable the two processors to run at their own maximum speeds. Also, as clock gating is not recommended for FPGAs, the preferred way to manage power is to shutdown the clock domain when the APU is not in use. Files: In the ModelSim project, the NIOS 2 APU interface is the top level entity that contains the APU as well. The design files are in the "NIOS2_APU" folder. ? ? Instr_Filter.v – Instr_Filter module NIOS2_APU.v – NIOS2_APU module, that contains the Instr_Filter module, various FIFOs, and the APU. ? NIOS2_APU_tb.sv – Test bench of the NIOS2_APU

module. Various instructions are tested. ? NIOS2_APU.do – ModelSim script that sets up the waveform window and runs the NIOS2_APU_tb test benches. ? fft32_data.xlsx, mac_data.xlsx, rb_data.xlsx – Excel files that contain in depth information about the input and expected results of the computation oriented instruction test benches. These files are used with the test benches that invoke the 32 point FFT, the MAC, and the computation of "rb". In the Quartus project, the interface is just a custom instruction bridge that does not contain the APU. The files are in the "\src\apu_ci" folder. Twiddle Factor Memory The various memories used by the array processor are instantiated in the "Mem" folder in the ModelSim project. One memory that does deserve special mention is the twiddle factor memory, which is read only. The memory is initiated by the hex files "twiddle_r.hex" and "twiddle_i.hex". These hex files are created using the function "make_twiddle_hex_file" in the "form1.cs" file of the "APU_Tester" project. The "Hex_File.cs" file contains support for the HEX format in general. It's coded according to recommendations from Wikipedia. The key diagram describing the HEX format is shown below. [15] The Altera Quartus seems to complain whenever each line in the HEX contains more than one word. Despite the warning, the memory does seem to be instantiated correctly. ModelSim does not complain about having more than one word per line. To avoid the Altera Quartus warning, the twiddle factor HEX files are created with one word per line. Instruction Set Instruction Formats Instructions are variable length. Each instruction word is 32 bit wide. The format of the first instruction word: Op code = {Mode, Code} Mode = ALU

26**mode [31:28] [27:24] [23:20] [19:**

16] [15:0] Mode Code Source Dest Arg Code = An additional code to differentiate operations that use the same ALU mode. In the "AP_control.v" file, the "Mode" is called "alu_mode", and the "Code" is called "op_code". Source is the value applied to the pins x1_sel and x0_sel. ? ? x1_sel <= bits[23:22] x0_sel <= bits[21:20] This means: "source" code X1 Selection X0 Selection 0 A0 A0

21**1 A0 A1 2 A0** B0 **3 A0** B1 **4** A1 **A0**

E B1 B0 Dest is the destination memory. This group of bits translate into which write enable pin will be toggled on. This field is decode according to the following table. "dest" code Write Enabled 0 A0r and A0i 1 A1r and A1i 2 B0r and B0i 3 B1r and B1i 4 A0r only 5 A1r only 6 B0r only 7 B1r only 8 A0i only 9 A1i only A B0i only B B1i only C A0r, A0i, A1r, A1i - FFT case D B0r, B0i, B1r, B1i - FFT case All: A0r, A0i, A1r, A1i E B0r, B0i, B1r, and B1i. None - none of the write F enables will be turned on. Arg = Instruction Argument. This is usually the (512 - length) value. 512 - length Various array length values are encoded as 512 - length. So to do an operation for 6 numbers, don't pass a "6" to the array processor. Instead, pass a "512-6", which is "506", to the array processor. This value is loaded into a counter, and the overflow of the MSB, which is the decimal value of 512, indicating the end of the array processor operation. Onchip Memory Instructions Op Code {Mode, Code} 02 21 20 30 40 50 60 70 E0 Description send_data(DEST dest, int[] i_array, int length) NIOS 2 will send data via the fifo_in word[0] dest = destination memory word[0] arg = 512 - length of the data word[1] = the first data value word[2] = the second data value, etc. get_data(SOURCE source, int real, int start_index, int length) Copy data to the "fifo_out" for NIOS 2 to read word[0] source used to choose the memory to read from word[0] dest must be "F" - no write to memory will occur word[0] arg = 512 - length word[1] MSB = 0 for getting the real component; 1 for imaginary word[1][8:0] = start index copy_general(ALU_COPY_MODE alu_mode, SOURCE src, DEST dest, int length, int src_start, int dest_start) This operation copy data from "source" to "dest". The source of the copy is source[source_start] to source[source_start + length - 1]. The destination of the copy is dest[dest_start] to dest[dest_start + length - 1]. The exact operation depends on the ALU mode used. word[0] source, destination used word[0] arg = 512 - length word[1] upper 16 bits = source_start word[1] lower 16 bits = dest_start dual_round(SOURCE

src, DEST dest, int length, int src_start, int dest_start) This operation is created after running the FFT module shows that rounding is needed if the final digital (LSB) is to be preserved. This operation is similar to the "0x60" operation - but the rounding is applied to both outputs. The 0x60 operation only has rounding on Y1. The "0xE0" operation pass X0 to Y0, and X1 to Y1, rounding both outputs. word[0] source, destination used word[0] arg = 512 - length word[1] upper 16 bits = source_start word[1] lower 16 bits = dest_start reset_memory(DEST dest, int start_index, int length) 00 This function loads zeros into the specified memory. word[0] destination used word[0] length = 512 - length word[1] lower 16 bits = destination start_index FFT Instructions Op Code {Mode, Code} F0 Description bit_reverse_copy(SOURCE source, DEST dest, int length, int bit_reverse_type) The bit reverse length is actually the half length of the FFT. For example, a length 32 FFT would have 16 numbers in each array. The bit reverse would copy the first 16 values to memory 0 and the second 16 values to memory 1. This needs to be specified manually. The APU also does not infer the type of bit reverse necessary from the "length" code. The bit reverse type needs to be manually specified. The data is copied from "source" to "dest". The starting index is 0. It's like copy, with starting index 0, and the destination address having a bit reverse applied to it. word[0] source, destination used word[0] arg = 512 - length word[1][2:0] = bit reverse type FFT(SOURCE source, int read_stage_number, int write_stage_number, int fft_shift, int length) 10 The length is actually the half length. For example, a 32 point FFT would use a length of 16. The destination is assumed based on the source. If the source is memory A, then the destination is memory B. If the source is memory B, then the destination is memory A. To make the control design easier, both the source and destination should be encoded in the instruction so that the control circuit does not have to assume. word[0] source and destination used word[0] arg = 512 - length word[1][29:28] = fft_shift (code directly applied to ALU input) word[1][7:4] = FFT stage number for read address generator word[1][3:0] = FFT stage number for write address generator Math Instructions Op Code {Mode, Code} Description mac(SOURCE src, DEST dest, int length, int src_start, int dest_index) Carries out for example B0[dest_index] =

12**A0[0]*A1[0] + A0[1]*A1[1] + ... + A0[** source_length **-1]*A1[** source_length **-1]**

24 One input the the multiplier comes from X1, and this input can draw from the four memory buffers A0, A1, B0, and B1. The other multiplier input comes from T, and this one can be: ONE (t_sel=01), A0 (t_sel=10), or A0_conjugate (t_sel=11). The "source2" is the selection of the T input. word[0] source and dest used. word [0][21:20] as x0_sel has no effect in this case. This field should also be used to drive t_sel, only for this instruction. word[0] arg = 512 - length word[1][31:16] = source_start word[1][15:0] = dest_index conjugate_mirror(SOURCE src, DEST dest, int length, int src_start, int dest_start) In the algorithm, the destination should not be the same as the source. For example, if the source is A0, then the destination should not be A0 as well. 51 Executes an operation where the conjugate of one array is copied to another. If used with source A0 and destination A1, then A1[dest_start - k] = Complex.Conjugate(A0[src_start + k]). The "k" runs from 0 to length-1. word[0] source and dest used word[0] arg = 512 - length word[1][31:16] = source_start word[1][15:0] = dest_start - in this case, the write address counter is running backward. compute_power(int length, int src_start, int dest_start) 0E Compute the power by adding the sum of square of the real and imaginary parts, for the values stored in buffer A0. This is implemented as A0 times its conjugate. Next find the log2 of this power (but it's not quite log2 - see ALU section for a table showing the exact mapping). Then take 16 minus the log2 result and store it at C0[dest_index]. word[0] arg = 512 - length word[0] source needs to be encoded - the source is always A0 word[0] dest must be "F" - no write to memory will occur word[1][31:16] = source_start word[1][15:0] = dest_index - in this case, the dest_index is for the "C0" memory inside the ALU, not for the write address generator. compute_Rb(int mu_k, DEST dest, int length, int src_start, int dest_start) 31 Computes: B0[k] = mu_k * Complex.Conjugate(A0[k]) * A1[k] * (double)(1 << Pi_int[k]); The k will run from 0 to length-1 word[0] arg = 512 - length word[0][23:22] src should be A0 or A1. The conjugate array source should always be A0, because only A0 is conjugated and feeding

the T input. word[0] dest should be "2" or "3" - the target is always memory B0 or B1. word[1][31:16] = src_start word[1][15:0] = dest_start word[2] = mu_k add_array(SOURCE src1, SOURCE src2, int start_index, int length, DEST dest) Adds two arrays "src1" and "src2", and puts them in the "dest" array. The indexes used will be "start_index" through "start_index + length - 1". 11 This is implemented as FFT operation with twiddle factor of 1. word[0] src contains the selection for x0_sel and x1_sel, so it contains both source 1 and 2. word[0] dest =dest word[0] arg = 512 - length word[1][31:16] = source start_index word [1][15:0] = destination start_index SSRAM Instructions Op Code Description {Mode, Code} ssram_reset(int low_addr, int length) Sets a range of "ssram" as zero. 01 Word[0] dest must be "F" - no write to memory will occur Word[0] argument = 512 - length Word[1] = low_addr, the SSRAM starting address 22 ssram_write (SOURCE source, int source_start, int ssram_base_addr, int length) Stores buffer memory (A0, A1, B0, or B1) into SSRAM. The whole complex number is stored, with the real part stored at the even address and the imaginary part stored at the odd address. Store a complex array into SSRAM - modeled in C# as an integer array. Only first "length" elements complex array is actually stored. Word[0] source used Word[0] dest must be "F" - no write to memory will occur Word[0] argument = 512 - length Word[1] = SSRAM base_addr 5/4: "length" issue currently not resolved - The length_counter overflow of 512 means the maximum length of array operation is 512. The SSRAM bus width is 32 bit, so reading or writing a complex number require two cycles. An array length of 512 limit the SSRAM transfer size to 256 complex values, which is sub-optimal. ssram_read(DEST dest, int base_addr, int length, int skip) Retrieve a Complex[] from an integer array called "ssram" This is actually a variant of the more complicated function below. ssram_read(DEST dest, int base_addr, int length, int skip, int highest_addr, int block_size) 0F Retrieve a Complex[] from an integer array called "ssram". This version has a wrap around capability for circular buffer addressing. When the address computed goes over highest_addr, the address used for ssram access will be the computed address minus "block_size". Word[0] dest used Word[0] arg = 512 - length; Word[1] = SSRAM base_addr Word[2] = skip Word[3] = highest_addr Word[4] = block_size Overall Correctness Testing The AEC_Algo C# program has a version of the original AEC algorithm (AEC32_double.cs), as well as an FPGA version (AEC32_FPGA_portX.cs) that uses the ArrayProc object. If these two versions generate the same output, then it can be said that the FPGA port and the ArrayProc object being used to simulate the array processor are correct. A separate C# program, called "APU_Tester.cs", applies the same instructions to the FPGA board and the software ArrayProc object. It's capable of comparing the entire buffer memory in the software ArrayProc object with the buffer memory of the array processor running on the FPGA. If the two sets of buffer memories match for all instructions over a long testing time, then it can be said that the FPGA board implements the ArrayProc software object correctly. Then by transitivity, the FPGA board implements the AEC algorithm correctly. The role of the AEC_Tester program is pictured below. The use of random parameters in step #2 is the key to proving the correctness of the FPGA implementation. A screen shot of the AEC_Tester program running is shown below. Synthesis Resource Requirements Logic Cells M4K Memories Multipliers 9x9 Multipliers 18x18 Total 12560 86 1 49 Array Processor 6877 64 44 Array Processor. Datapath 5804 60 44 Array Processor. Datapath.ALU 4752 1 44 NIOS 2 CPU 2696 20 2 Floating Point Custom Instruction for NIOS 2 CPU 1385 1 3 The array processor is the largest resource user, with most of that resource used in the datapath and the ALU. The datapath is a 48 bit datapath. If smaller precision will suffice, then the resource usage can be decreased. In the datapath and the ALU, going from 48 bit width to 32 bit width should result in 33% saving in terms of logic cells and 50% saving in terms of multipliers. Conclusion In this project, an array processor has been added to the NIOS 2 processor to form an ASIP. The various instructions of this array processor have been tested to match a C# model. The same C# model also correctly runs an AEC (acoustic echo cancelling) algorithm. Unfortunately, once the AEC algorithm is implemented, the FPGA does not produce the correct results. The system level C# test bench that checks instructions individually proved to be insufficient. The instructions need to be tested as a group as well. Running Time: It takes a maximum of 553,202 clock cycles, at 100 MHz, to process 1 frame (256 data points). This imply a processing rate of 46.3 kHz. The NIOS 2 processor just makes a few simple calculations and decisions. It is lightly loaded and can undertake other tasks while it oversees the AEC algorithm. Correctness: All individual array processor instructions have been tested using random input as

the instruction parameter. After the input is applied, the entire array processor internal buffer is read back to the host PC to compare with the C# software model. Unfortunately there maybe errors when the instructions are used collectively. Further testing using a collection of instructions is needed. Cost Effectiveness: The resource in the synthesis implies that the design can be done on a Cyclone 3 part number EP3C16. The lowest cost of which is currently available for $26.70 on Digikey. No price break are shown for higher quantity. [6] The resource utilization can be significantly reduced if the algorithm is allowed to run on a lower precision, which will allow a lower cost FPGA to be used. The project is however, disturbingly time consuming. Implementation of the algorithm itself on a DSP should be much easier. A floating point 200 MHz DSP, part number TMS320C6713 cost $27.72 in quantity of 100. [6] A fixed point 200 MHz DSP, part number TMS320C6204 currently cost $16.83 in quantity of 100. The Future: It's important to use the FPGA correctly to avoid a head on competition with processors, where the FPGA is likely to lose on a variety of metrics. Processors are cheap and power efficient. They come with large software libraries and optimized peripherals. But processors are usually optimize for a particular application, so that the engineers will have to relearn many things if they need to solve different problems. The FPGA has an advantage in interfacing, inventory management, and being able to quickly adapt to new situations. It has enormous computation potential, and unlocking this potential is an area of on-going research. There are many books and projects online regarding how to unlock the FPGA potential, but it's far from obvious. References #1. Digikey, price of EP3C5E144C8N FPGA is $12.8 for volume of 1, as of April 10, 2011, www.digikey.com #2. Altera, "Nios II Performance Benchmarks",

10**http://www.altera.com/literature/** ds/ds **_nios2_** perf **.pdf? GSA_pos=1&WT.oss_r=1&WT.oss=**

nios 2 benchmark #3. Microchip, PIC32 Performance, http://www.microchip.com/en_US/family/pic32/ #4. Atmel, AT91SAM9261 Preliminary Summary,

19**http://www.atmel.com/dyn/resources/prod_documents/** 6062s **.pdf #5.**

Mouser, PIC32 and AT91SAM

17**prices as of April 10, 2011,**

www.mouser.com #6. Digikey, EP3C5 and EP3C16

17**prices as of April 10, 2011,**

www.digikey.com #7. Wolfson Microelectronics, WM8731 datasheet, http://www.wolfsonmicro.com/products/codecs/WM8731/ #8. Altera, Differences Between Cyclone II and Cyclone III

13**FPGAs, http://www.altera.com/products/devices/cyclone3/overview/cy3-** differences **.html** #9. **Altera,**

About

18**Altera's Cyclone** FPGA Series,
http://www.altera.com/products/devices/cyclone- about **/cyc-**

about

22**.html?GSA_pos=** 8 **&WT.oss_r=1&WT.oss=**

cyclone iii 60nm #10. Xilinx,

11**Spartan-6 Family Overview,**
http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf

#11. Spansion, S29GL-N MirrorBit® Flash Family, http://www.spansion.com/Support/Datasheets/S29GL-N_01_12_e.pdf #12. ISSI,

8**256K x 72, 512K x 36, 1024K x 18, 18Mb SYNCHRONOUS PIPELINED, SINGLE CYCLE DESELECT STATIC RAM,** http://www **.issi.**

com/pdf/61VPS_LPS-51236A_102418A.pdf #13. Wolfson Microelectronics February 10, 2011 press release:

3**Wolfson's DSP and software suite provides world-leading HD Audio solution and significantly improves battery performance,**

http://www.wolfsonmicro.com/media_centre/item/Wolfsons

3**_DSP_and_software_suite_provides_world- leading_HD_Audio_solution/**

#14. Stanford, Music 422 –

14**"WAVE PCM soundfile format"–**
https://ccrma.stanford.edu/courses/422/projects/WaveFormat/

#15. Wikipedia, Intel Hex, http://en.wikipedia.org/wiki/Intel

2**_HEX 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27**

28 29 30 31 32

33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74