# Designing a Modular DSP Core For Real-Time Audio Performance

A Senior Project presented to
the Faculty of the Electrical and Computer Engineering Department
California Polytechnic State University, San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science in Computer Engineering

Kevin Richard Brewer

June, 2014

# Table of Contents

## Significant Figures:

# Abstract

This project provides an overview for building a Digital Signal Processing (DSP) core on a Digilent Nexys2 FPGA board. The DSP core is designed to give Cal Poly students interested in DSP and its applications to audio engineering a usable platform to perform signal processing and analytics. The processes of the DSP core are modular, allowing students to design their own implementations of various adder and multiplier functions. Infinite impulse response (IIR) filters and finite impulse response (FIR) filters using both cascade and parallel implementations are the primary processing tools in the core, and all output can be visually and aurally presented using an oscilloscope and acoustic speakers, respectively.

The completed project is intended to act as a platform and guide for students to design their own filters and I/O modules. The completed filters show the correct inputs and outputs necessary to properly implement filters to work with the I/O controllers and external pmod controllers. Overall, the project successfully samples and processes digital signals, and provides a visual tool for understanding how various audio effects physically work.

# Introduction

A core portion of audio engineering and sound production relies heavily on the technology and implementation of digital signal processors (DSPs). Analog sound waves, as well as all other analog waves, require an analog-to-digital converter in order to properly work with binary arithmetic hardware, such as field-programmable gate array (FPGA) boards. DSP cores provide a fast solution for real-time processing of live sound.

The processing speed of the DSP core has a tradeoff with the physical size and power draw of the hardware. The performance speed of the DSP is determined by binary arithmetic architecture, and how quickly the multiplier and adder functions can calculate processing results. In the scope of this project, the size of the FPGA board will not need to be physically augmented to meet performance requirements, and the power drawn will not be problematic because the board will stay physically connected to a power source via USB at all times.

Students at Cal Poly are currently implementing digital signal processing techniques on top of a non-modular platform, which limits the scope of class-based projects that can be used in testing. A platform that allows modular, "drop in" design would permit students to test their own designs and code architecture on a Nexys2 FPGA board, which they will have already obtained for prerequisite classes, such as CPE 133 and CPE 233. The Nexys2 board is an FPGA that does not come with a built-in DSP core, so the scope of this project pertains to creating a base platform for the DSP core, allowing students to apply their own drop in designs.

In an audio engineering context, this project will be able to demonstrate how the application of DSP can create various sound effects to a sampled input signal in real time. This application allows producers and audio engineers to add studio and live production value for musicians. Through digital filters, engineers are able to enhance a signal to give it presence and reverberation, or remove noise frequencies. The user will be able to interchange different effects by implementing their own DSP blocks and filter realizations to the modular DSP core.

# Project Goals

This project is intended to give Cal Poly students a usable platform for researching digital signal processing and its applications to audio engineering. The DSP core will be developed with VHDL on a Digilent Nexys2 FPGA board. The core will control an input ADC and an output DAC, which it will use to sample input signals and export processed outputs. The architecture will support IIR and FIR filtering and circular buffer shift registers for holding previous inputs and outputs. The modifiable core will allow users to build their own DSP blocks using VHDL, which they can use to replace the default VHDL code in the core.

Overall, the goals are directed toward providing optimal speed, performance, and modularity. To be a reliable tool, the DSP core needs to run in real time. The calculations it makes within the adders and multipliers need to be fast, accurate, and must not run out of bounds. The architecture must be simple to allow for usability of drop in replacements in order to best be used as a learning tool. These goals reflect the marketing and engineering requirements within the scope of the project.

# Background

Digital Signal Processors (DSPs) take in digital input and through arithmetic computations create processed output. Audio signals can be taken as input and converted from analog to a usable digital signal through an analog-to-digital converter (ADC). It is necessary for the signal to be converted in order for computations to take place within the DSP core. Once a signal has been successfully processed it can be reconverted to an analog signal using a digital-to-analog converter (DAC), and then analyzed visually and aurally using tools such as oscilloscopes and audio speakers, respectively.

The calculations that allow the DSP to create processed output signals use binary adders and multipliers on the input signals. Calculations are variable in speed depending on implementation. The two standard implementations of binary arithmetic blocks are cascade and parallel. Adders and multipliers with a cascaded implementation perform their processing in a queued, ordered fashion, and adders and multiplier with a parallel implementation perform multiple computations concurrently. The output signal is dependent on the implementation as well as the order that processing is performed.

Digital filtration can be implemented using infinite impulse response (IIR) filters and finite impulse response (FIR) filters. FIR filters are the more stable of the two digital filters, and have an input impulse response of a finite duration. IIR filters are usually unstable since they typically have internal feedback and have a decaying looped response. Both of these digital filters have their purpose for producing various audio effects and require testing to verify proper handling in cases of arithmetic overflow.

# Requirements

The following are the requirements for the final design of the project. Functional requirements, similar to marketing requirements, describe the major functions that the final design must accomplish. Performance specifications, similar to engineering requirements, are quantifiable performance benchmarks that the final design must achieve.

## Functional Requirements

In order to be of use in an educational setting, the DSP core must be modular and intuitive to use. The core will act as a platform and allow the user to modify the default design, or add drop in replacements for any block in the core. The default core must be intuitive for the user to understand which constants to modify in order to produce intended results. The platform will perform using DSP blocks in order to produce a wide variety of audio processing on input signals.

The DSP core will be able to take in samples from the input and produce desired processing using various finite state machines for the different modules. The DSP will convert all inputs from analog-to-digital in order to perform the required processing, and then convert from digital-to-analog when the input has finished being processed. Output will be displayed both visually and aurally using an oscilloscope and speakers, respectively.

The core must perform calculations in real time. Speed is a priority for the performance of the DSP core. Power draw is not an issue because the Nexys2 will always be connected by USB to a PC.

## Performance Specifications

The final design requires that the DSP core is able to use both IIR and FIR filters that are able to perform processing calculations through adders and multipliers using both cascaded implementations. This will make up the default base for the platform. The final platform must be able to take any “drop in” DSP block that is properly implemented in VHDL and integrate it with the rest of the core.

The final design must be able to produce the following audio effects to audio inputs: echo, reverb, and cascaded IIR. Testing for creating these effects will occur in real time using a sampling rate of 44.1kHz, a standard sampling rate in the field of audio engineering.

# Project Design and Alternatives

Code structure will be implemented using modular DSP blocks written in VHDL. The Nexys2 FPGA board will have the responsibilities of allowing I/O functionality through an SPI controller, and processing signals through a filter realization controller. After researching the benefits and drawbacks of the various binary multipliers and adder, the Booth Multiplier and Carry-Select Adder were chosen for their processing speed. All external hardware used will convert all analog audio signals into digital signals that the data processing unit can use, and vice versa.

### Black Box

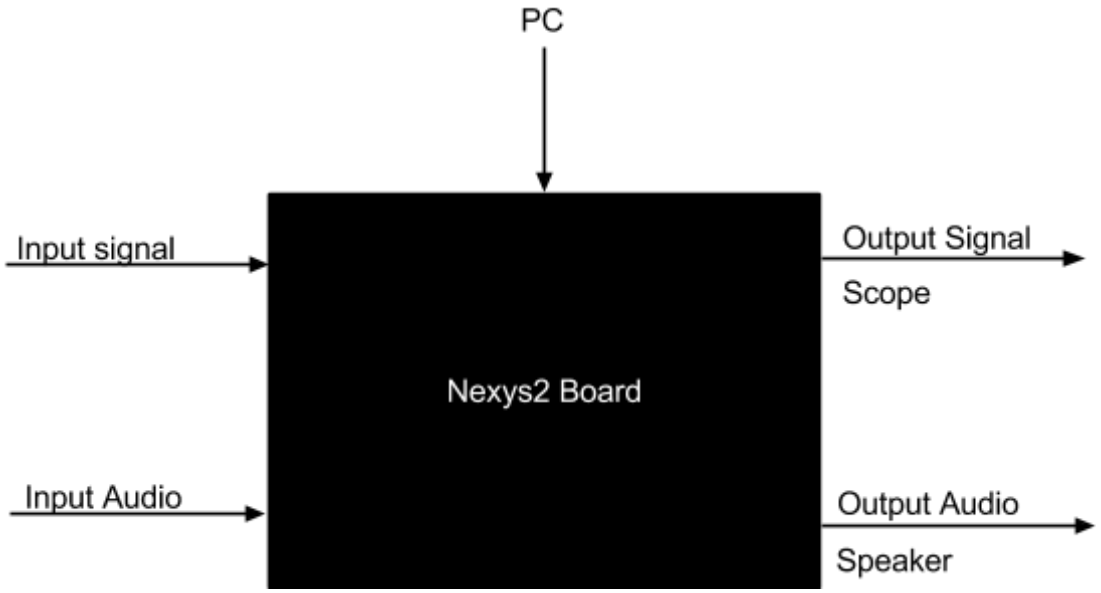



**Figure 1:** Nexys2 Black Box Diagram

The black box realization of the project consists of the input types allowed by the DSP core, and the expected outputs provided. The DSP core will take in generated signals and audio inputs and produce both processed signals and audio. The input signal is created using a signal generator, and the input audio is created using a microphone or instrument. The output signal is connected to an oscilloscope to verify correct processing, and the output audio is connected to a speaker to aurally analyze the processing effects on the input audio. The PC connection is to ensure that power is supplied to the Nexys2 board via USB. The PC connection will also install the DSP core onto the FPGA board whenever a user changes the core by adding or altering a DSP block.

## Subsystem Diagram

The core subsystem is made up of two main modules: the I/O controller and the Data Processing Unit (DPU).

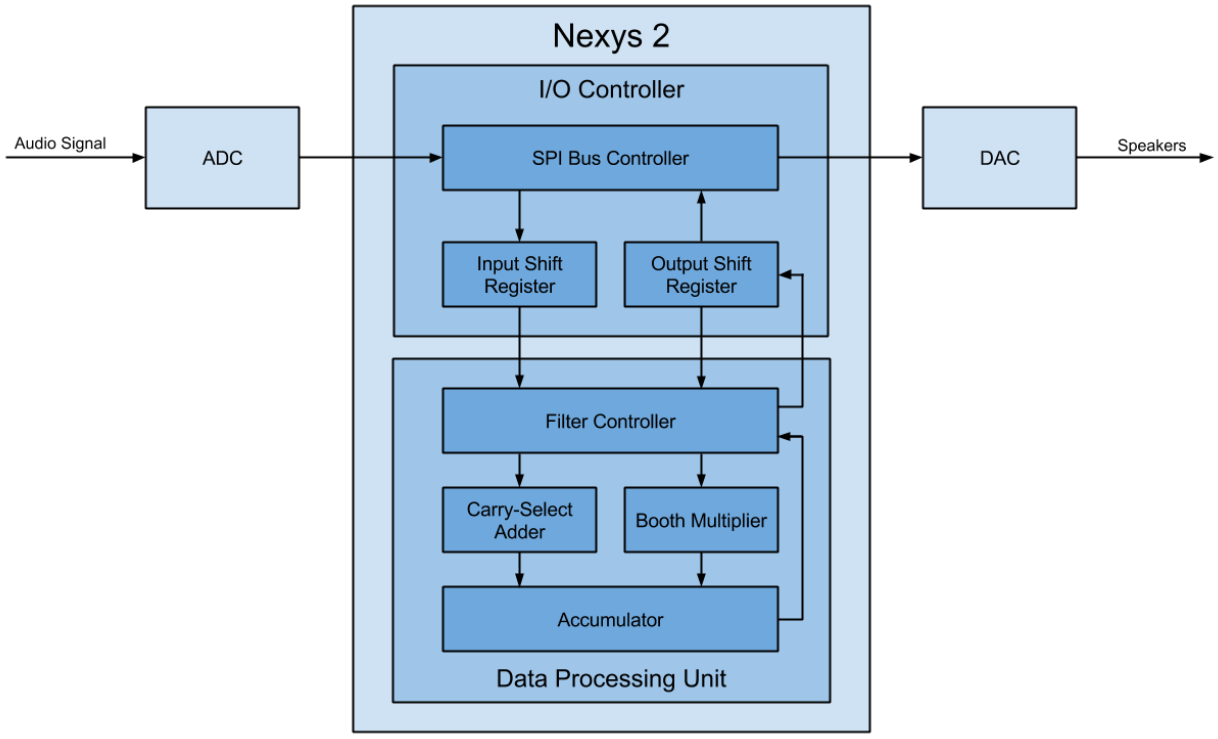



**Figure 2:** DSP core subsystem diagram.

***The I/O Controller***

This module is in charge of ensuring proper communication between the analog converter (ADC), DPU, and digital-to-analog converter (DAC). Correct input and output of digital signals occurs in the serial peripheral interface (SPI), a databus that communicates with I/O components. The SPI sends input signals to the input shift register and receives processed signals from the output shift register. These two shift registers are the primary communication between the I/O controller and the DPU.

The ADC is an additional hardware module that samples the input audio channel into a binary representation. The design of the DSP core depends on a binary Data Processing Unit (DPU) for proper functionality and testing. The DAC creates an analog signal from the processed data and directs the output to a speaker to be analyzed. Proper verification of DSP core can be analyzed visually through an oscilloscope, and aurally through a speaker.

***The Data Processing Unit***

The data processing unit is where all the audio processing occurs. The filter controller is set in VHDL by the user's intended audio effects on the signal. Realizations include IIR and FIR filters that can produce effects such as echo, reverb, flanging, low-pass, high-pass, and band-pass. The Filter Controller utilizes the Carry-Select Adder and the Booth Multiplier to perform any arithmetic manipulation to the input signal being processed. Depending on the filter realization, an accumulator could be accessed by both the adder and multiplier. The accumulator, as well as the output shift register in the I/O Controller, provides feedback to the Filter Controller, allowing more complex data processing.

The Carry-Select and Booth Multiplier were both chosen primarily for their speed. Alternatives to the Carry-Select include the Ripple-Carry and the Carry-Lookahead, both slower than the Carry-Select. The Carry-Select Adder does take more processing utilization than the alternatives, however, the processing capacity of the Nexys2 is able to manage it. Processing speed is the highest priority with the DSP core in order to perform real time, so selecting an adder with higher processor utilization is necessary. The Booth Multiplier was selected over

## Audio Effects

Applying various implementations to the DPU, mainly in the Filter Controller, produces different audio effects. Below are the effects that the project seeks to create and verify.

***Delay***

Delay is the simplest audio effect and works by taking an input signal and outputting the same signal, but at a given delay. The only DSP block necessary for this effect would be a delay block, which will be utilized by both FIR and IIR filter realizations.

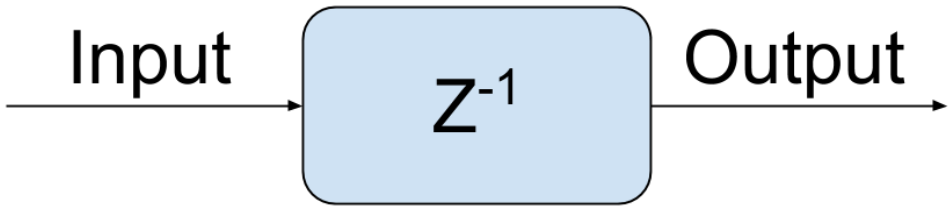



**Figure 3:** Delay filter realization.

***Echo***

Echo is very similar to delay except it has an extra layer of complexity, feedback. Feedback is when an output signal is added into the input of the next processed signal. This allows for an echo effect by adding a signal used previously into the processing of the current signal. The effect should have a saturation component included in order for the echoed signal to diminish over time to avoid overflowing the processor.

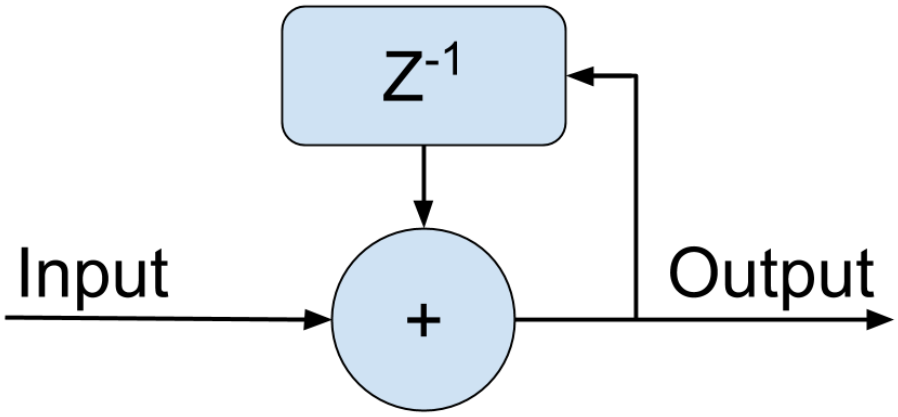



**Figure 4:** Echo filter realization.

***Reverberation***

Reverb is a type of resonance that a sound produces when the sound wave reflects off of surfaces in a given area. This effect gives more presence to an audio signal, and is typically applied to a featured sound such as vocals or lead guitar. The way the filter works is similar to echo, but with more accumulators and delay modules. There are several ways to implement reverberation depending on the desired amount of reverb and the frequency range to emphasize. Echo is a very basic implementation of reverb known as plain reverb. Adding some additional DSP blocks one could create a more complex variation of reverberation known as all-pass reverb, shown in figure 5. This effect utilizes adders, multipliers, and delay DSP blocks.

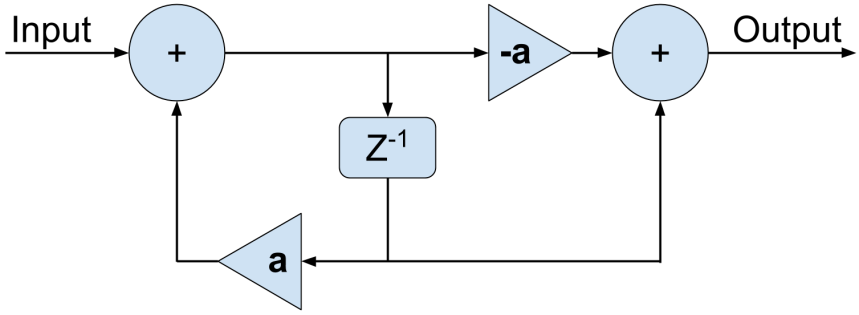



**Figure 5:** All-pass reverberation filter realization.

# I/O Modules and Testing

The I/O Controller, as defined in the previous section, is responsible for sampling. It accomplishes this task by properly controlling the AD1 and DA2 external Pmod devices, and by retaining a circular buffer to store past and present input and output values. This section will explore further how the individual modules operate, and how they can be manipulated to allow integration with different external devices.

## CLK_DIV.vhd

CLK_DIV is one of the most crucial modules within the I/O controller because it allows control of the internal clock. The Nexys2 has a clock speed of 50 MHz, which is a clock period of 20 ns. This fast clock allows the FPGA board to compute internal logic functions very rapidly, however, it is too fast of a clock cycle to control the AD1 and DA2 Pmods.

According to the AD1 datasheet, the serial bus on the Pmod can run up to 20 MHz. Using a clock divider, the Nexys2 can extend the clock period by $2^2$. This would make the clock period for the AD1 80 ns, which is 12.5 MHz. The DA2 can handle a faster clock cycle, but still not on the scale of the Nexys2, so a clock divider that extends the clock period by $2^1$ would make the DA2 clock period 40 ns, which is 25 MHz. The clock divider module is also used to control the sampling rate of the entire I/O controller. It divides the Nexys2 signal by (2 * n) for any value of n to produce a desired sample rate.

The clock divider module is very simple structurally. It takes in an input clock signal and an integer to divide the clock cycle, and outputs the divided clock signal. An internal counter increments every rising edge from the input clock. Once the counter has reached the desired divider, set by the integer input, the output clock signal will flip its bit. The following equation determines the output clock given an input integer.

$$SAMP\_RATE = 50{,}000{,}000 / (2 * DIV)$$

In the test cases used in this project, a DIV of 556 was used to produce a sampling rate close to 44.1 kHz, which is the common sampling rate for audio systems. Below is an example of clock rate changing with different DIV values.

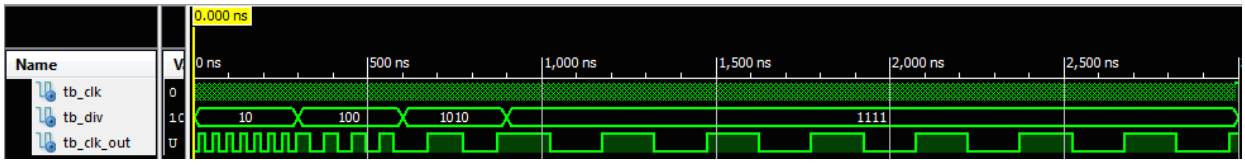



**Figure 6:** Output clock period changing with div values.

## SHIFT_REGISTER.vhd

The shift register is responsible for storing the most recent input readings and processed output results. The registers work as circular arrays, meaning n amount of samples are stored for n * SAMP_DIV amount of time. When a new sample is added to the array it is added to array slot 0, and samples 0 through n-1 are shifted up to slots 1 through n. Slot n is outside the size of the array so it is just deleted, leaving only the n most recent samples. For this project n was set to 30, allowing the DSP to reference input values x(t) through x(t-29) and output values y(t) through y(t-29).

A new sample flag is responsible for letting the shift register know when a new sample is ready to be added to the shift register. For the input shift register, the ADC controller will set a flag when a new sample has been read. All filters will need to set an "output done" flag when a processed output is complete in order to control the output shift register. The shift register takes the flag as an input, as well as a 16-bit value to store. The output is a 16-bit vector array of n values.

Shift registers are crucial for DSP filters to use previous samples in the output equations. Using an output shift register allows a user to design more complex IIR filters that require previous output samples in its filter realizations. Below is the resulting simulation for the shift register test bench in Appendix E. The simulation begins after tb_res (reset) is set to 0. The new sample reading tb_d is set to 6 different values, which updates each array slot in the shift register every clock cycle. The clock cycle is just a flag for the IO controller to set when new samples have been read in and new output values have been processed.

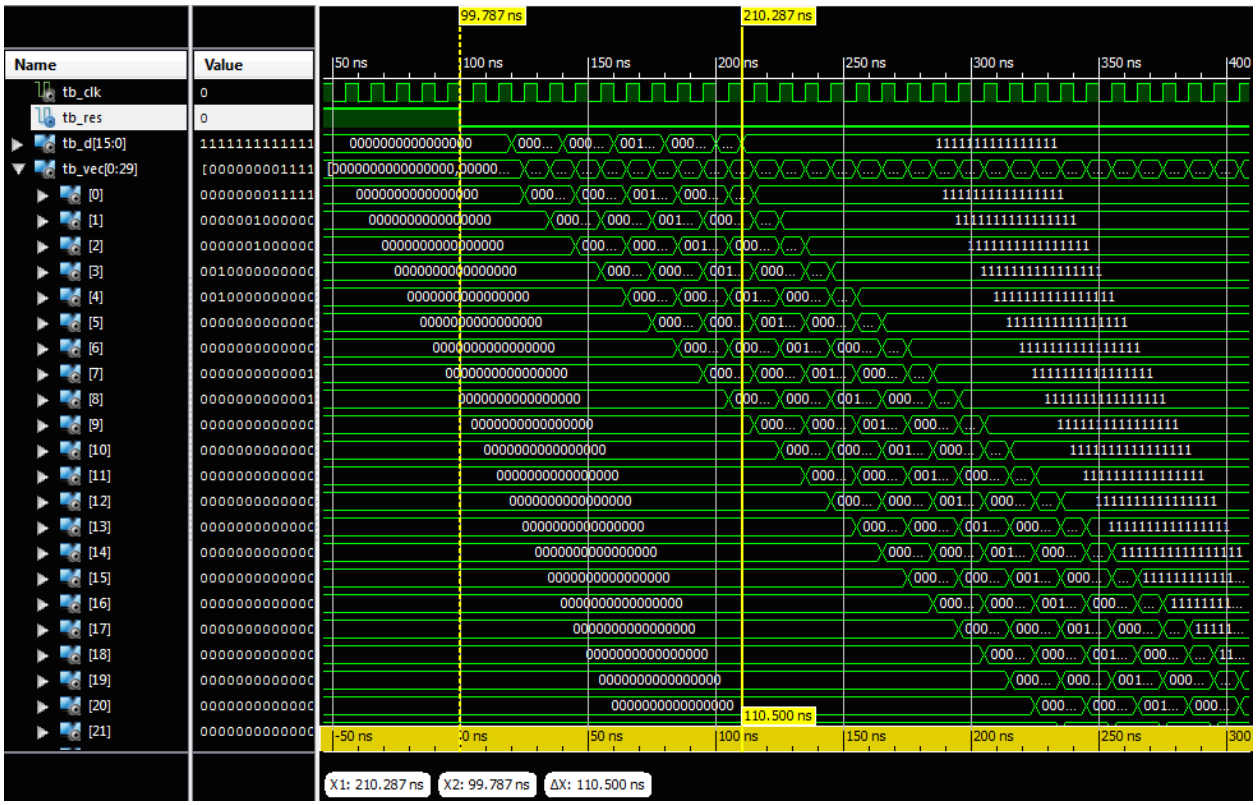



**Figure 7:** Output shift register simulation.

### SPI_ADC_CTRL.vhd and SPI_DAC_CTRL.vhd

The SPI ADC controller and the SPI DAC controller are responsible for properly handling the AD1 and DA2 Pmods. Both modules are designed specifically for the I/O Pmods selected for the project and can be interchanged with other user generated SPI controllers. The following section describes the architecture of the SPI controllers and how their respective finite state machines operate.

The ADC controller takes in a clock signal, a reset flag, a read flag, and a Master-In-Slave-Out bit, and outputs an ADC clock signal, a CS flag, a sampled signal vector, and a done flag. The input clock is the 50 MHz clock signal produced by the Nexys2 board. The reset signal is simply a signal that is used asynchronously throughout the DSP core to reset all sampling and clear all shift registers. The read signal is the flag that lets the ADC controller know to start sampling the next input and the MISO bit is the produced reading bit received by the ADC Pmod once a sample has started. With these inputs, the ADC controller uses a clock divider to create a clock that the Pmod can use to read each bit of the sample, which the module outputs to the I/O controller. The ADC controller also outputs the CS flag, which the ADC Pmod used to start the sampling finite state. When the ADC has finished sampling the 12-bit signal, it outputs the sampled signal along with a done flag.

The finite state machine for the ADC controller has four states: IDLE, SETUP, START, and FINISH. The module stays in the IDLE state until the I/O controller permits sampling by raising the RD bit to 1. Once doing so the next state of the controller is the SETUP state, which the state machine stays in for one clock cycle. During this period the CS flag is raised to 1, alerting the ADC that a sample should be started. The next state is the START state where the ADC is sampled for 16 ADC clock cycles. For the AD1, the first four bits are always 0, which is why only 12-bit signals are produced. A counter is used to verify when the sample has completed. Once the 12-bit signal has been successfully sampled, the finite state machine changes to the FINISH state and the DONE signal is set to 1. The I/O controller uses this signal to control the input shift register and store the sampled signal. The finite state machine then returns to IDLE and waits for a new sample to be requested. The finite state machine is diagrammed in Figure 8.

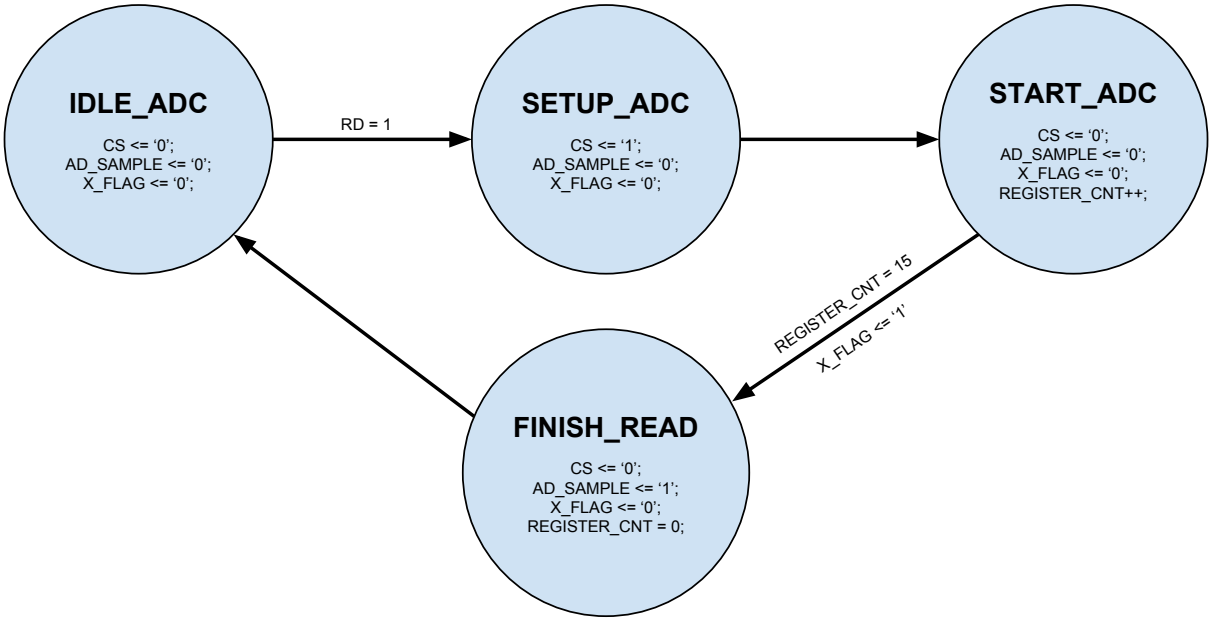



**Figure 8:** Finite state machine for ADC Controller.

Below is a simulation of the ADC controller based on the test bench found in Appendix E. The simulation shows an output change after 16 ADC clock cycles since the last CS raise. The MISO signal shows the input signal that the ADC would read in when the finite state machine enters the START state, and the x vector shows the signal read by the MISO during sampling.

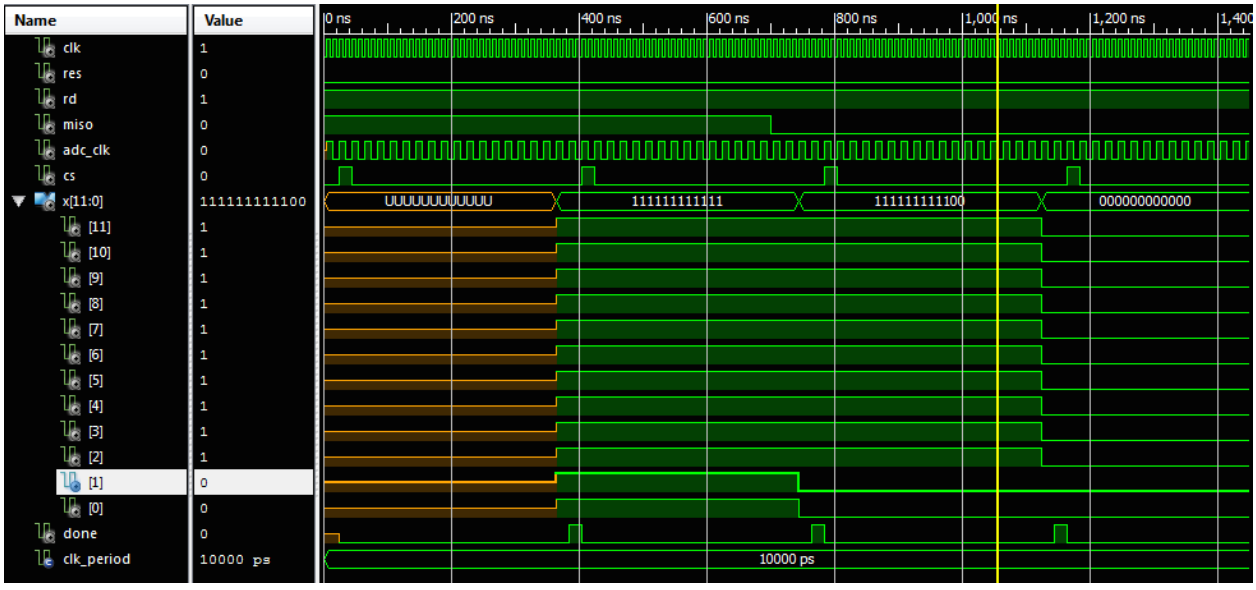



**Figure 9:** ADC controller simulation.

The DAC controller, like the ADC controller, takes in the Nexys2 clock signal and the reset flag to control its state machine. The DAC controller also takes a write signal WT and a 12-bit output signal as input. WT is the flag that lets the DAC controller know to start outputting the 12-bit signal Y. For the scope of this project, the DSP should be constantly writing to the DAC. This is reflected in the finite state machine, Figure 10, where the WT flag is only used to take the state machine out of IDLE mode. The DAC controller creates a DAC clock output using the clock divider module, and uses it to properly control the DA2. Additionally, the DAC outputs a SYNC signal and a MOSI signal. The SYNC signal, similar to the CS signal in the ADC, lets the DAC know that a new signal is to be exported as a Master-Out-Slave-In bit (MOSI).

The finite state machine for the DAC controller has three states: IDLE, SETUP, and START. The module, like the ADC, stays in IDLE state until the I/O controller permits output by raising the WT bit to 1. The finite state machine then moves to the SETUP state and SYNC is raised to 1, alerting the DAC that a signal is to be output. Using a counter, the DAC controller exports the Y signal starting with the MSB. The DA2 waits for four clock cycles before reading from the MOSI bit, so four 0's are set in front of the MSB to make the output signal 16-bits in length. Once the signal has finished exporting, the finite state machine returns to the SETUP state and repeats the cycle. The finite state machine is diagrammed in Figure 10.

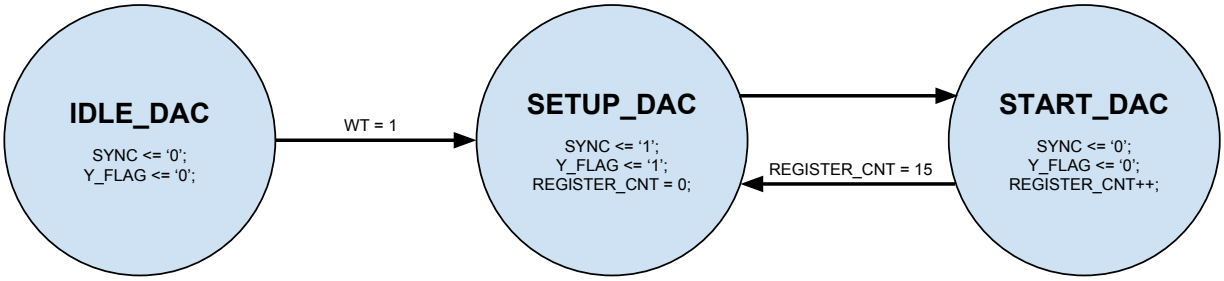



**Figure 10:** Finite state machine for DAC Controller.

Below is a simulation of the DAC controller based on the test bench found in Appendix E. The MOSI bit does not start to export to the DAC until four DAC clock cycles after the SYNC bit is raised. All output is based on the Y vector signal that is passed to the DAC controller.

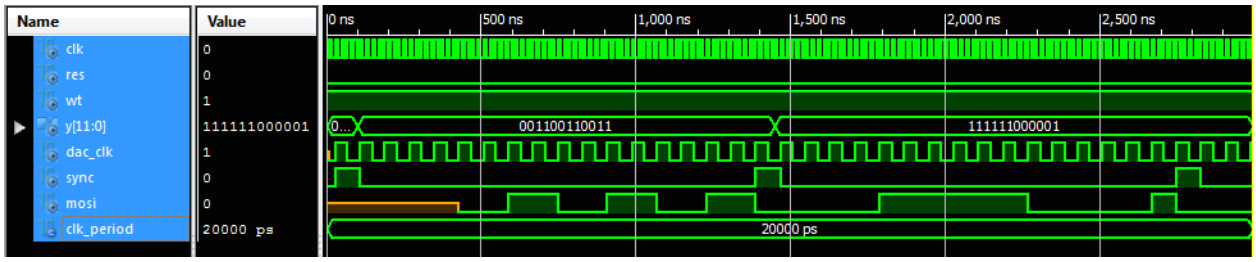



**Figure 11:** DAC controller Simulation

### IO_CTRL.vhd

The I/O controller is responsible for setting the sampling rate of the DSP and for selecting the controller modules to interface with the I/O devices. The sampling rate is set inside of the DEFINITIONS.vhd file and is created using another clock divider. The I/O controller is set using the provided ADC and DAC controller modules, shown in the previous section, but can be easily modified to accommodate new controllers for additional or substitute Pmod devices. A majority of the sampling work is taken care of by the SPI controllers, so the overall architecture of the I/O controller is fairly simple. To work as intended, the controller should constantly write to the DAC, but only read from the ADC *once* per sample period. This is accomplished using two flags: one alerting when a new sample has been retrieved and one alerting when the sample clock has completed a period. Using this method and the finite state machine shown in Figure 12, a single sample is read per sample period.

The finite state machine for the I/O controller has only two states: HOLD_IO and SAMPLE_IO. The HOLD state is the default state and basically keeps the most recently sampled value for the shift register. The HOLD state keeps the WT_FLAG raised to 1 so that the DSP can constantly write to the DAC, and keeps the RD_FLAG set to 0 so that the ADC stays IDLE. The only time the RD_FLAG should be raised to 1 is in the SAMPLE state. The SAMPLE state only occurs when there is a new sample and it is the first sample in a given sample period. Once that sample has been successfully added to the input shift register, the state machine returns to the HOLD state until the next sample period has begun. This looped architecture allows for simple and infinite sampling and exporting.

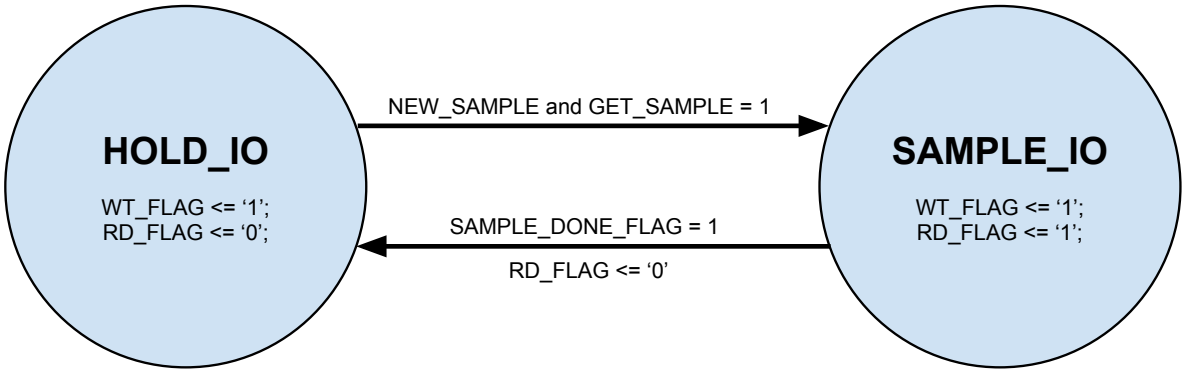



**Figure 12:** FSM for the I/O Controller.

In the hierarchy of the project, the I/O controller is only called by the DSP_CORE module, which controls the entire system. All interfacing with external Pmods is sent as output from the I/O controller to the core via the SPI controllers. Additionally, the input and output shift registers are monitored by the I/O controller and are sent to the core for use by the filter realization controller.

# Filter Modules and Testing

The Data Processing Unit is made up of shift registers, adders, and multipliers and is responsible for the filtering side of the DSP core. Essentially, this side of the DSP uses the tools necessary to create the desired output. The following section will introduce the arithmetic logic chosen for the project and detail the example filter realizations built during the project.

The adder and multiplier selected and used in the verification of the filters are the Carry-Select Adder and a modified Booth Multiplier. The multiplier takes two 16-bit inputs and produces one 32-bit product. The adder takes two 32-bit inputs and produces one 32-bit output. In order for the two modules to work properly, the sampled inputs must be the proper bit length. This can be accomplished either by always using a multiplier before using an adder, or concatenating the sample to the end of leading zeros. These arithmetic modules were leveraged from Joseph Waddell's Cal Poly Senior Project. Waddell built a variety of arithmetic modules and tested the speed at which each computed. The fastest arithmetic modules were selected for this project. The user can swap the adder and multiplier for their own to test the limitations of real-time processing given their selected sampling rate.

Every arithmetic module must be compatible with the overflow module, also leveraged from Waddell's project. This module is responsible for keeping the arithmetic operations done on the processed sample unsigned. Keeping the sample unsigned will result in a ceiling for multipliers and adders, instead of some kind of twos compliment occurring. The result is commonly known in audio engineering as clipping. Clipping usually occurs when a sample is too loud (outside of a devices decibel range) and distortion occurs to the signal (shown in Figure 13). All values sampled will be positive so the high rail will be all ones and the low rail will be all zeros for this system.

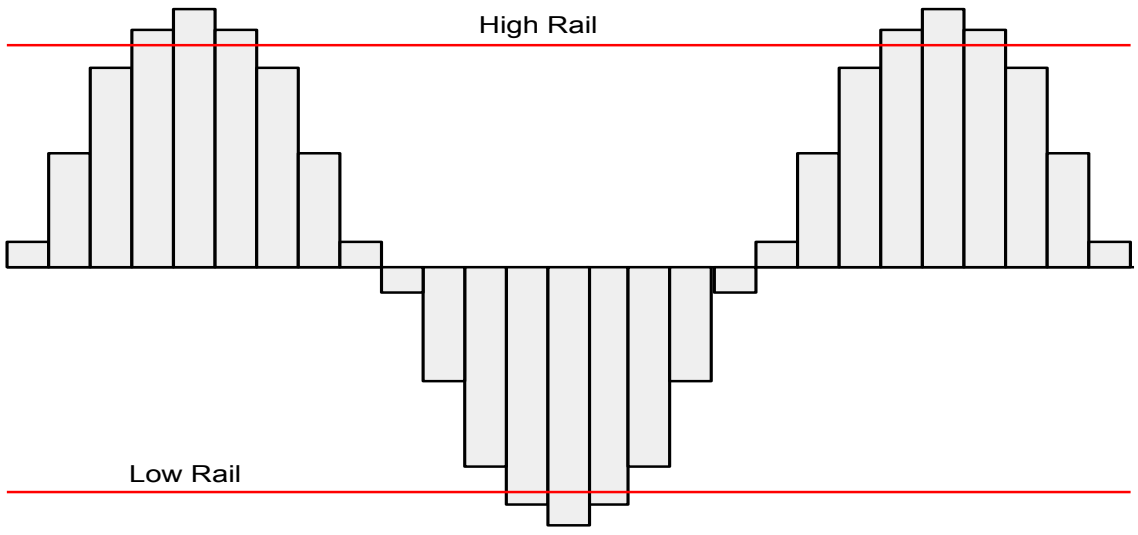



**Figure 13:** Clipped waveform.

## DELAY_FILTER.vhd

The delay filter was the simplest filter that was tested on the DSP core. This filter relies solely on the input shift register. All filters have an input that comes from the I/O controller alerting them when a new sample has been created. Once delay receives the sample ready flag it returns a specified previous value from the input shift register. For this project both input and output shift registers were set to 30 values in length, and the delay filter returns the oldest input value. The code used to produce this filter is provided in Appendix C, and the code used to create the simulation in Figure 14 can be found in Appendix E.

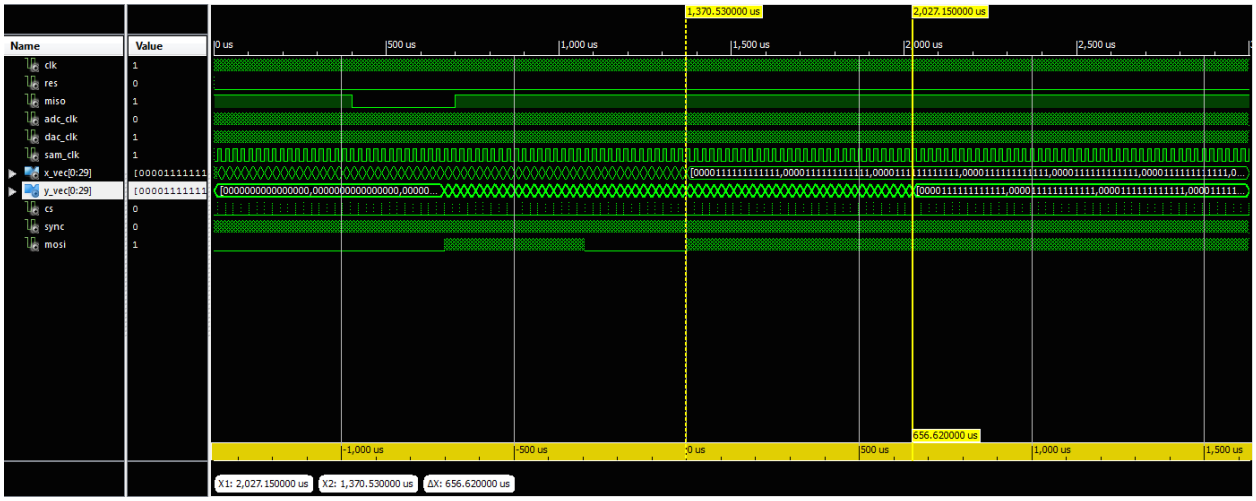



**Figure 14:** Test Bench simulation for the Delay Filter.

The simulation in Figure 14 is shown from the perspective of the DSP_CORE.vhd module, which integrates all the filters with the I/O controller. In the simulation the Y_VECT, output register, waits until the last register in the X_VECT changes before it updates its first register. The clearest representation that the delay filter is properly functioning can be observed from the MISO and MOSI bits. These bits show the proper pattern of high low high that is first produced by the MISO and then repeated by the MOSI later on in the simulation. This simulation verifies that the shift registers are properly function as delay blocks.

## ECHO_FILTER.vhd

The echo filter was designed to properly emulate the sound of a modifiable echo. To do this an adder was needed to combine an echoed signal to the current sampled signal, and a multiplier was needed to diminish the echo comparatively to the current sampled signal. If the echo is not reduced then a sampled signal will be echoed forever. To keep the filter from constantly overflowing the multiplier was initialized to a moderately low value. Increasing the multiplication value gives the listener a sense of highly sustained echoes, and lowering the multiplication value reduces the amount of repeated echoes that will be produced. Changing which sample in the shift register is to be the echoed signal can modify the filter further. The greater the register number to be echoed, the longer the echo period, and vice versa. The architecture of this filter imitates a very simple, first-order FIR filter.

This filter was created using a finite state machine with the following states: STANDBY, MULT, MULT_HOLD, COMB, COMB_HOLD, HOLD, SEND. STANDBY is fairly synonymous to the IDLE

state in the SPI controllers in the sense that it is a busy wait period. The STANDBY state waits for a new sample to be received and then enters the MULT state. In the MULT and MULT_HOLD states the new sample is multiplied to a predetermined constant while the echoed sample is multiplied to a lower constant. The MULT_HOLD, COMB_HOLD, and HOLD states were implemented to guarantee the filter had enough computational time to correctly process the next output value. The COMB state (short for combinational) signals the filter that the multiplication state has finished and that the output values are ready to be added together using the Carry-Select adder. After holding for two clock cycles, the finite state machine enters the SEND state, where the OUTPUT_DONE flag is raised to 1. This flag controls the output shift register in the I/O controller, and adds the new signal to the register.

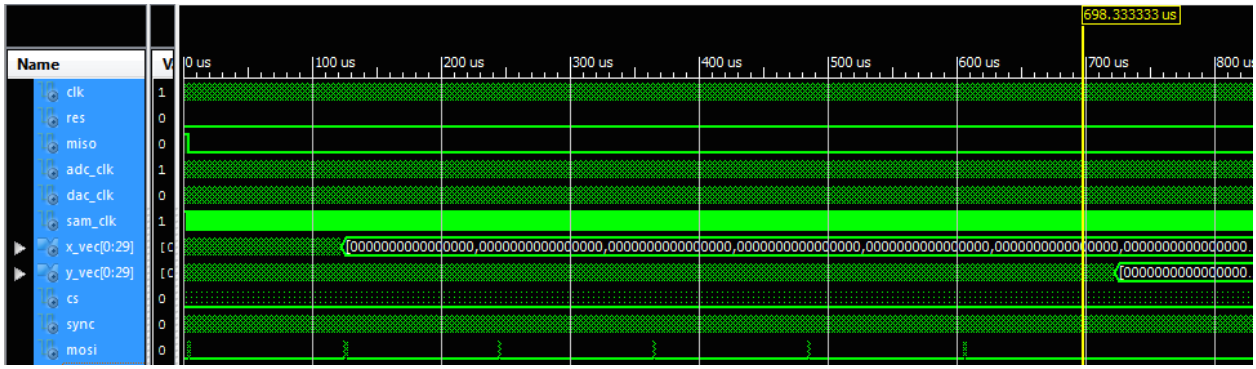



**Figure 15:** Test bench simulation of the Echo Filter.

Figure 15 shows a test bench simulation of the echo filter. In the test bench, one sample was taken during the very first sample cycle (~10us). This sample is used as output on the MOSI line as soon as it is sampled. Since there weren't any previous registers in use the output is purely the sampled signal. This signal repeats 6 more times throughout the simulation before reducing to 0. This decay is scalable by altering the multiplication value set to the echo signal.

## REVERB_ALL_PASS_FILTER.vhd

The reverb all-pass filter is designed as a first-order FIR filter, using two adders and four multipliers total. It is designed to not remove any frequencies from the reverb filter. Structurally it is similar to the Echo Filter, except that it requires an additional shift register. In order to obtain the center delay block in Figure 5, a shift register must be added and updated after a new sample from the input shift register has been added with the previous sample in the inner shift register. The inner shift register is then multiplied by a requested value and added to the previous register in the inner shift register as processed output for the output shift register. The finite state machine for the reverb filter is the same as the echo filter, except doubled. One pass through the finite state machine to calculate the inner shift registers next value, and then a second pass through to obtain the output registers next value.

Figure 16 – 18 show the resulting output of the test bench simulation from Figure 15 using the reverb all-pass filter. Throughout the simulation the output goes through waves of growth and decay as the sample is echoed and sustained over time. Eventually the signal fades entirely, as expected. Similar to the echo filter, the reverb filter can be altered by increasing the time delay and the multiplication values to make the sustains and echoes more or less present in the filter.

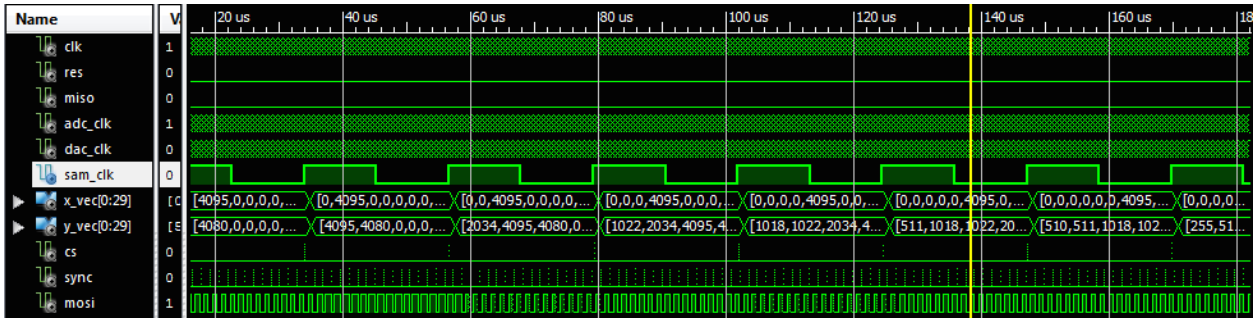



**Figure 16:** Test Bench simulation of the Reverb Filter – Part 1.

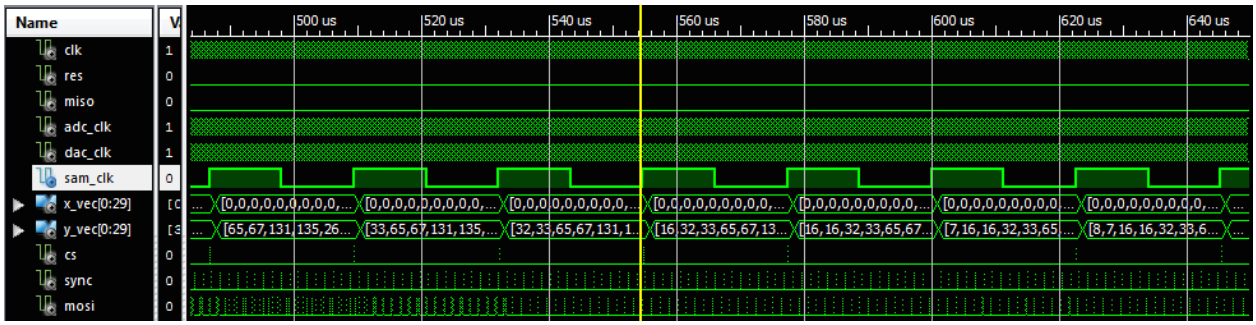



**Figure 17:** Test Bench simulation of the Reverb Filter – Part 2.

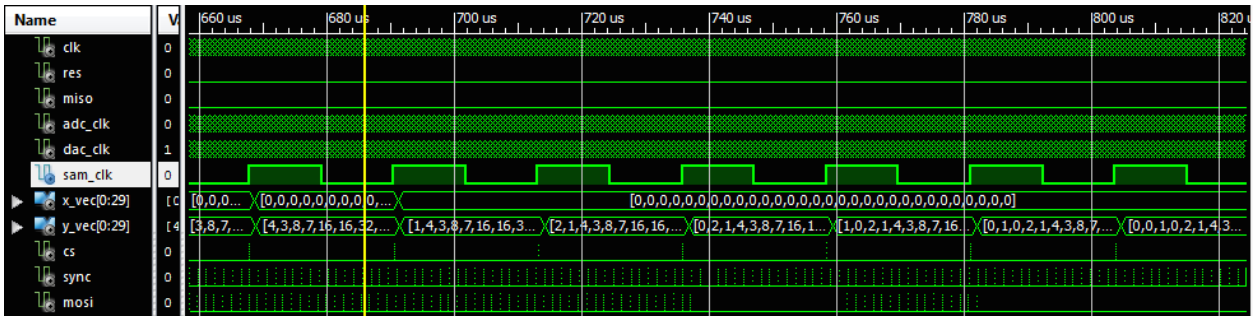



**Figure 18:** Test Bench simulation of the Reverb Filter – Part 3.

## SYMMETRICAL_FORM.vhd

This is the last filter created for the project, and serves as a finite impulse response DSP block that can be tuned to meet desired output. This filters purpose is to demonstrate the power of the filter realization unit within the DSP core. The symmetrical form filter produces the following filter.

$$Y(t) = B_0[X(t) + X(t-7)] + B_1[X(t-1) + X(t-6)] + B_2[X(t-2) + X(t-5)] + B_3[X(t-3) + X(t-4)]$$

This filter is completed using three stages of computations. After a new sample has been verified and added to the input shift register, the symmetrical form filter takes the eight most recent samples and performs addition, finding the summation of register 0 with 7, register 1 with 6, register 2 with 5, and register 3 with 4. Once these summations are found, each is multiplied by their respective B value. Each B value is a 16-bit constant to accommodate the 16-bit modified booth multiplier used in the project. The default value set for each B value is "0000000000000001". The final computation stage uses three more adders to find the total summation of the all the products created. In total, seven adders and four multipliers were used in this filter.

The final output from the last adder is a 32-bit value, which is expected. This value, however, is not compliant with the 12-bit expected output, so it must be reduced in length. This requires setting a window in the 32-bit register to select the range in which to select the 12-bits. The constant N sets the MSB from this 12-bit window. An N value should accurately change the 16-bit B value into the proper floating bit binary value. It is most common to have an N value greater than any B value. Users can fine-tune their filters by setting the N value, as well as the four B values, to see how the results vary spectrally.

Figure 19 shows a simulation test bench where a repeated sample of 4095 is received after a single sample of 511. The filter has the following settings. $B_0 = B_1 = B_2 = B_3 = 1$ and $N = 11$. The output after the first sample is the same as the sample because there are no previous input values. With each successive sample there is a gradual decay.

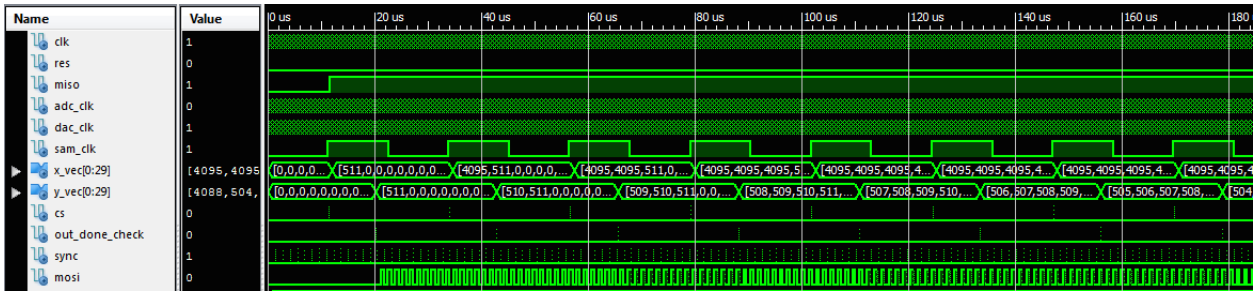



**Figure 19**: Test Bench Simulation of the Symmetrical Form Filter

Through further testing users can find the full potential of this filter and the spectral outputs that result. This filter can be used as a DSP block as well, in either a parallel or cascaded filter design. Using two symmetrical filters with two output complete flags, a cascaded filter could be designed by a user to have even more complex results.

# Physical Construction and Integration

The physical construction for this project is very simple and requires four main components: DAC, ADC, Nexys2 FPGA board, and PC. Both the DAC and ADC will be physically connected to the Nexys2 through its I/O ports. The other end of the ADC and DAC connect to a CON4 Pmod to allow the Nexys2 to interact with signal producers and a speakers. The integrated converter with the CON4 is displayed in Figure 20. Power is supplied to the FPGA board via its USB port, which is connected to a PC. The complete physical construction is showed in Figure 21.

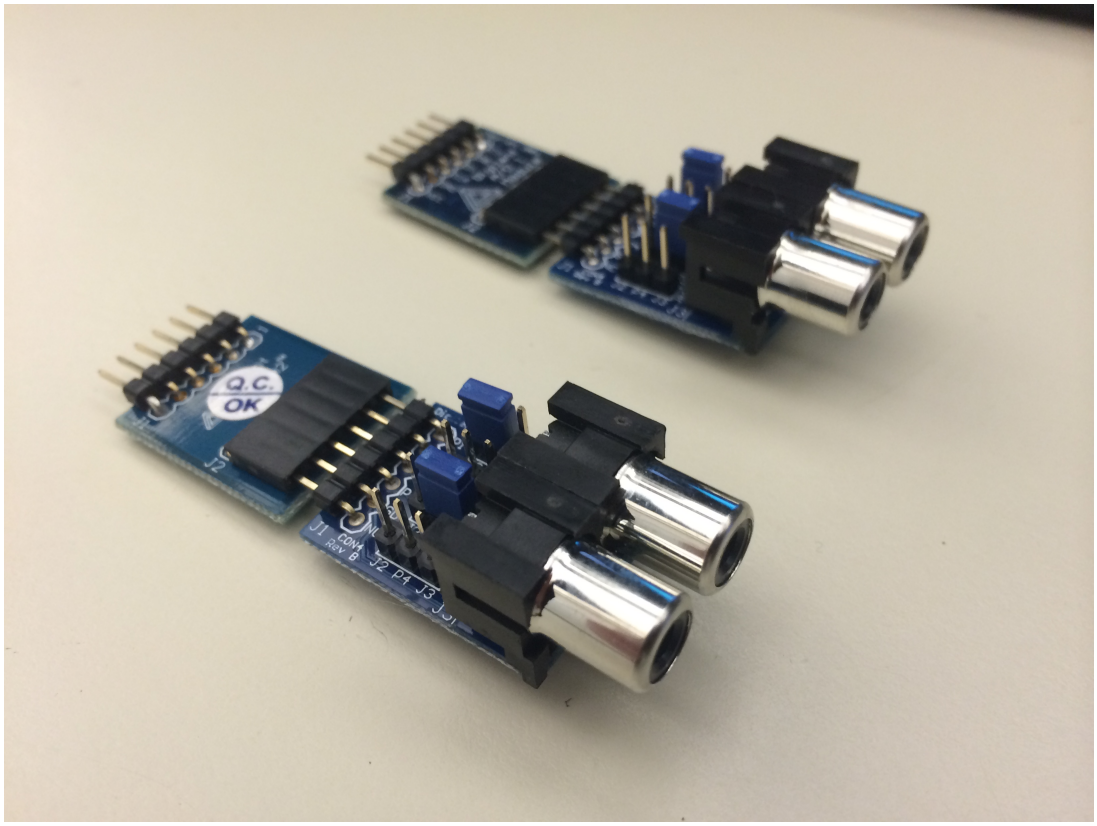



**Figure 20:** ADC and DAC Pmods connected to CON4 Pmods.

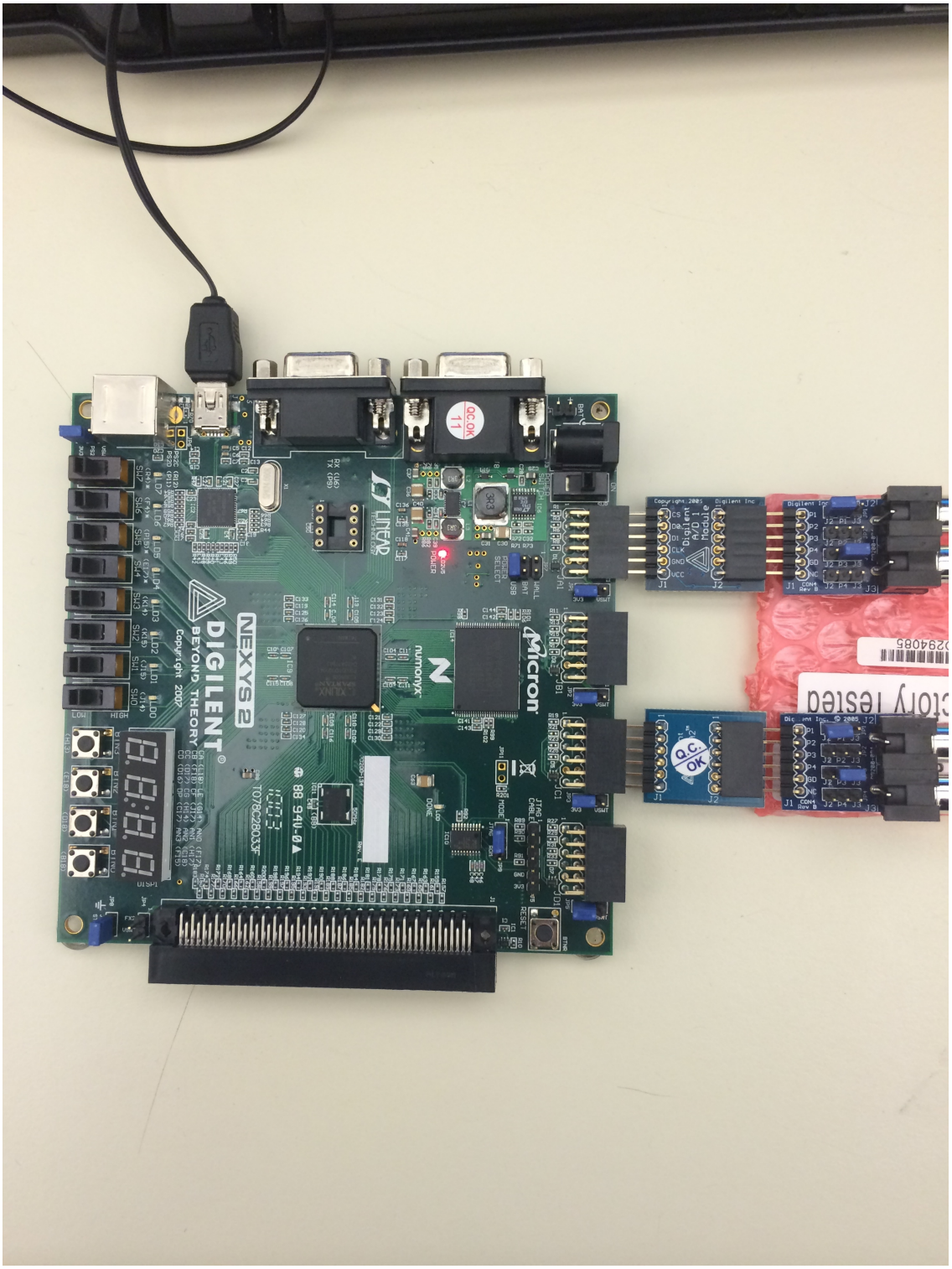



**Figure 21:** Completed physical integration.

# Conclusions

Designing and building a system to be modular requires simple architectures. The more a large controller can be broken up into several smaller controllers, the more modifications the system can support with lower chances of breaking the system. This is the underlining design philosophy used with the architecture of this project.

The I/O controller is responsible for setting a sampling rate at which to pull a new signal from the ADC. Setting a sampling rate that is too slow can cause aliasing and produce distorted or unwanted results. This is a result of having frequencies over the set Nyquist limit. Setting a sampling rate too high could also cause distorted results by not giving the adders inside of the digital filter enough time to fully process an output. In audio engineering, the typical sampling rate used that compliments the audible sound range for humans is 44.1 kHz, which is the default with this project.

The length of the digital value being sampled is related to the resolution of the DSP. A high bit length allows for greater range in values for a sampled signal, and thus more clarity. A low bit length causes distortion, unclear sound, and small range of possible values. This project uses an ADC and DAC that work with a 12-bit sample. This allows a signal to have a possible value in the range of 0 to 4095, "000000000000" to "111111111111". Within the context of this project, this resolution combined with the selected sampling rate produce suitable output at real time operating speeds. If a user has access to an ADC and DAC with a higher bit resolution that is also compliant with the Nexys2's I/O ports, the user could easily change the bit range values within the project to accommodate their I/O devices. Most of the Pmod products available through Digilent only support 12-bit length, so users can also write their own controllers for other Pmod devices and switch the platform's SPI controller without any additional modifications.

The input and output shift registers are accessible by all filters, allowing a user to create both finite impulse response (FIR) and infinite impulse response (IIR) filters with ease. Their array-based architecture allows a filter designer to confidently select which past sample to use in a filter realization. Additional shift registers can be added to add further complexity to a filter design and allow for more values to be stored in registers. An example of an "inner" shift register is demonstrated in the reverb filter.

The filters provided are intended to act mainly as a guide for users to create their own filters. They explore both the basic DSP techniques that can be applied to an audio signal and demonstrate how to create higher-order filter realizations. These filters are intended to be modifiable and possibly integrated by users. Using additional bit flags, a user could link multiple filters (either platform based or their own) to create a cascaded filter. By combining filters that are in parallel or sequentially cascaded can provide a user with even more output results.

## Bibliography

Julius O. Smith III. “Introduction to Digital Filters with Audio Applications.” Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, September 2007 Edition. Retrieved from https://ccrma.stanford.edu/~jos/filters/filters.html

Waddell, Joseph. “An Overview of Binary Arithmetic Architectures & Their Implementations in DSP Systems.” Electrical Engineering Department, California Polytechnic State University, June 2012.

Prof. Dr. B. Schwartz. “Digital Signal Processing With FPGAs.” Department of Electrical Engineering and Computer Sciences, University of Applied Sciences Hamburg, Edition SS 2004. Retrieved from http://users.etech.haw-hamburg.de/users/Schwarz/En/Lecture/IE8/Notes/Introduction.pdf

Meyer-Baese, U. *Digital Signal Processing with Field Programmable Gate Arrays*. Third ed. Heidelberg: Springer, 2007. Print.
Electronically retrieved from
http://opencores.org/websvn,filedetails?repname=phr&path=/phr/trunk/doc/references/PLDs_doc/Digital+Signal+Processing+with+FPGAs+-+3rd+Edition.pdf&rev=129&isdir=0

"Designing a Custom DSP Circuit Using VHDL." *IEEE Xplore*. N.p., n.d.
Electronically retrieved from
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=60525

"VHDL Modeling and Model Testing for DSP Applications." *IEEE Xplore*. N.p., n.d. Electronically retrieved from
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=744371

"Design of a High-quality Audio-specific DSP Core." *IEEE Xplore*. N.p., n.d. Web. 10 Mar. 2014. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1579921>

"Efficient DSP Architecture for High-quality Audio Algorithms." *IEEE Xplore*. N.p., n.d. Web. 10 Mar. 2014. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1465245>

Daniel Mlynek, Y. L. (1998, November 10). Design of VLSI Systems. Retrieved September 5, 2012, from

Micro Electronic Systems Laboratory: http://lsmwww.epfl.ch/Education/former/2002-2003/VLSIDesign/ch06/ch06_print.html

"A Framework for Teaching Real-Time Digital Signal Processing With Field-Programmable Gate Arrays." *IEEE Xplore*. N.p., n.d. Web. 10 Mar. 2014. <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1495664>

"Introducing Signal Processing through an Advanced Digital Design Course." *IEEE Xplore*. N.p., n.d. Web. 10 Mar. 2014. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5739195>

“Lab 5 – Digital Audio Effects (n.d.): n. pag. Lab 5 - Digital Audio Effects.” *Rutgers ECE Department*. Web. <http://eceweb1.rutgers.edu/~orfanidi/ece348/lab5.pdf>

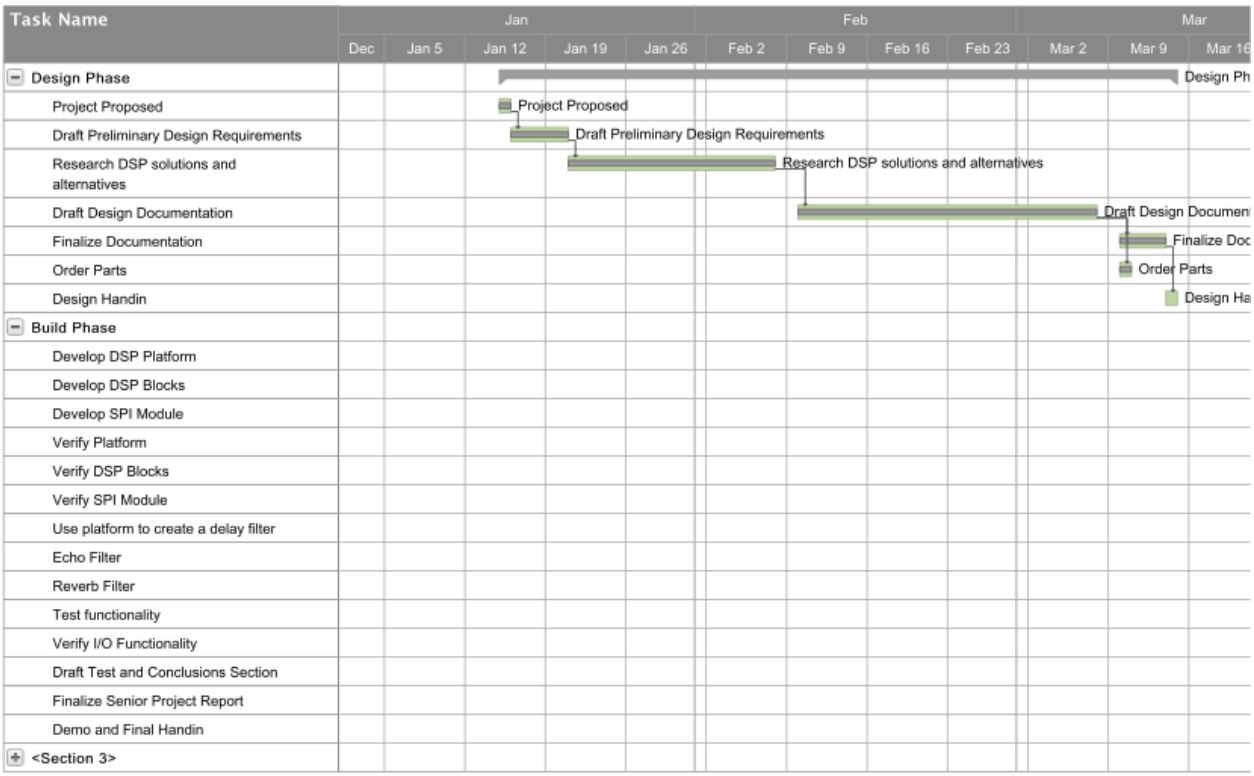
“Digilent PmodAD1™ Analog To Digital Module Converter Board Reference Manual.” *Digilent*. Web. <http://www.digilentinc.com/Data/Products/PMOD-AD1/Pmod%20AD1_rm.pdf>

# Appendix A: Project Planning

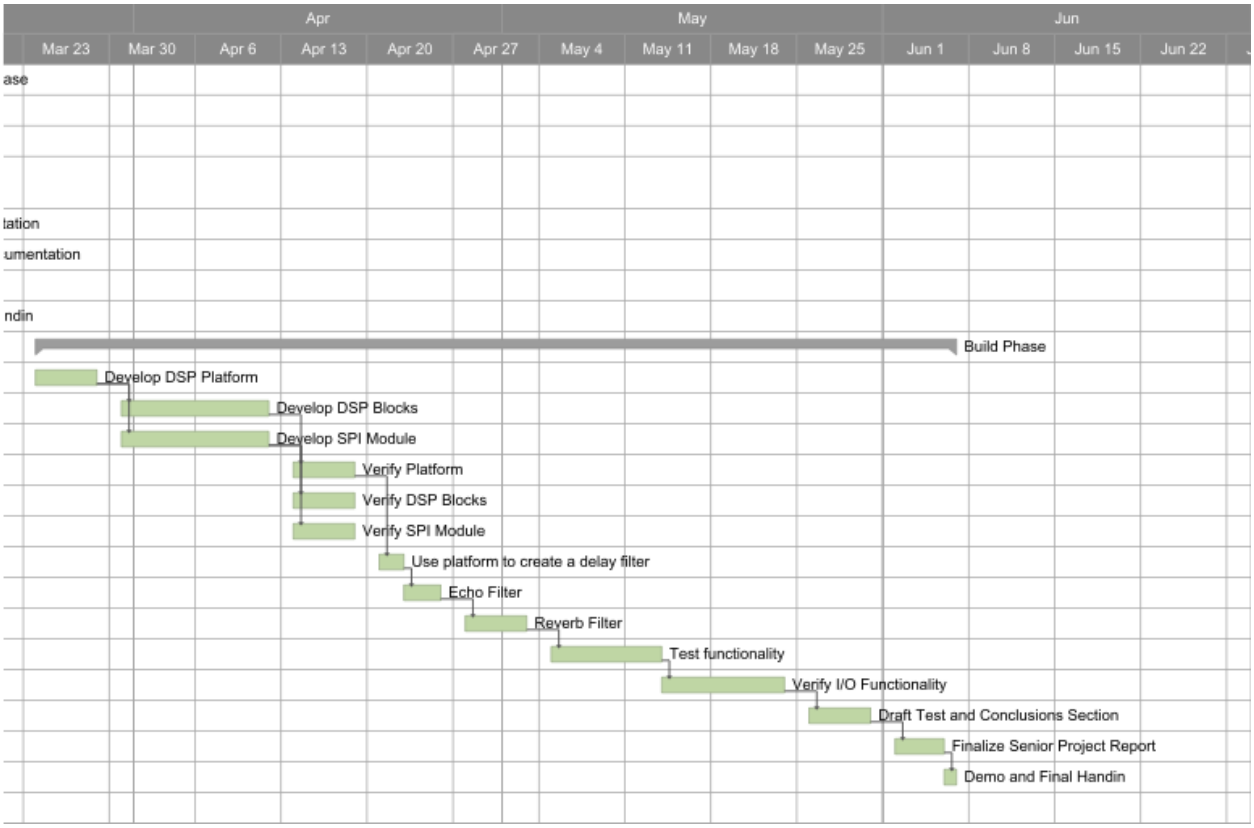
This appendix contains the following information: proposed Gantt chart and project budget.

## Gantt Chart

Below is the Gantt chart for quarter 1. This timeline displays the tasks fulfilled in terms of research and preliminary work.




Below is the Gantt chart for quarter 2. This timeline displays the tasks that will be fulfilled in terms of implementation and verification for the final product.




## Proposed Budget

| **Part Name** | **Cost** |
|---|---|
| Digilent Nexys2 FPGA | $99 |
| PmodDA2 | $21.59 |
| PmodAD1 | $34.99 |
| PmodCON4 (x2) | $30 |
| | |
| **Total** | **$185.58** |

# Appendix B: Analysis of Senior Project Design

## Summary of Functional Requirements

The DSP core is modular and intuitive to use. The core acts as a platform and allows the user to modify the default design, or add "drop in" replacement modules for any platform module in the core. The default core must be intuitive for the user to understand which constants to modify in order to produce intended results. The platform performs using DSP blocks in order to produce a wide variety of audio processing on input signals.

The DSP core is able to take in samples from the input and produce desired processing using various finite state machines for the different modules. The DSP converts all inputs from analog-to-digital in order to perform the required processing, and then converts from digital-to-analog when the input has finished being processed. Output is displayed both visually and aurally using an oscilloscope and speakers, respectively.

The core performs calculations in real time. Speed is a priority for the performance of the DSP core. Power draw is not an issue because the Nexys2 will always be connected by USB to a PC.

## Primary Constraints

The hardware is the main constraint in the project. The Nexys2 board has a limited amount of I/O ports that can connect to a limited amount of peripheral module (Pmod) devices. The Pmods available to use as analog-to-digital and digital-to-analog converters only produce and use signals that are 12-bits in length. This limits the resolution of both the sampled input and the produced output. The Nexys2's built in clock is a second limitation. Using a clock divider, there are a limited amount of sample clock cycles that can be used that are close to the desired sample rate of 44.1 kHz. Lastly, time is a significant constraint in getting some of the more complex filters fully functional.

The main challenge incurred during the project was designing a sampling unit that was modular. The sampling unit (IO_CTRL.vhd in the project) controlled the sampling clock, the ADC controller, and the DAC controller. All three of the controllers in the sampling unit have their own, independent finite state machine. Getting all three FSM to properly operate and be easily replaced or removed took a lot of redesign time during the build phase of the project. Over the span of the project the design for the sampler became more simple, and thus, easier to navigate, implement, and modify.

## Economic

The original estimated cost is the same as the final cost: $185.58. The following lists the cost of all materials used in the base project. Note: additional I/O Pmods can be purchased that also work with the Nexys2's I/O ports.

| Part Name | Cost |
|---|---|
| Digilent Nexys2 FPGA | $99 |
| PmodDA2 | $21.59 |
| PmodAD1 | $34.99 |
| PmodCON4 (x2) | $30 |
| | |
| **Total** | **$185.58** |

Additional equipment needed to support project include: a PC with Xilinx, a signal generator, an oscilloscope, and speakers. All of items except for the speakers can be openly accessed in the Senior Project lab on campus.

The original estimated development time was ~8 weeks. The actual development time ended up being ~9.5 weeks to complete.

### If Manufactured On a Commercial Basis

The project is intended to be a platform for student use. If the project were to be manufactured commercially it would most likely sell on a per CPE/EE program basis to universities. If every student in a CPE/EE department took a class in DSP and were required to get this project for lab purposes, anywhere between 100 - 300 students at that school would buy this device per year. If at least 50 universities have a DSP class that required this project, anywhere between 5000 - 15000 units would be sold per year.

The manufacturing cost is only the cost of the hardware from Xilinx. All of the produced work from the project is VHDL source code, which can be easily copied and transferred onto all of the Nexys2 boards. There is no manufacturing cost if the cost of all hardware is ignored. Otherwise the manufacturing cost would be $185.58 per unit, unless Xilinx were to make a deal with large unit orders.

To accommodate the cost of purchasing all the hardware to go with the VHDL source the retail price would be set to $360 to allow a 100% price margin. Using the estimate units sold per year (5k - 15k), a net profit per year in the range of $872k - $2.6 million if just 50 schools were to add this project to their DSP curriculum.

### Environmental

There is little to no environmental impact caused by the project. All manufacturing of FPGA boards and Pmod devices is done by third party hardware companies. All product made through the project is VHDL source code.

### Manufacturability

There are no issues on the projects side of manufacturing because all hardware is built by Xilinx or other FPGA and pmod device building companies. Some possible challenges that could arise include proper integration of Pmods with FPGA board and proper integration of signal generators and speakers with Pmod devices.

### Sustainability

Issues can arise with new versions of Xilinx. Any new versions of the system will require the source code to be migrated to that new addition. This does have the potential to cause errors or produce incorrect results if the platform changes significantly. Also it is possible to fry the Nexys2 board, and even more likely to fry either the AD1 and/or the DA2 Pmod devices.

This project impacts the sustainability use of resources by using widely distributed hardware for the platform and by being highly modular. All work from the project was dedicated to writing VHDL source code to be used with a variety of different hardware. Modular design allows the project to integrate with a wide variety of resources.

Possible upgrades that could improve the project are higher resolution ADC and DAC devices, and a newer FPGA board. If the sampling devices were to sample and export signals of longer bit-length then more complex computations could be done on them with greater accuracy. If a newer FPGA board than the Nexys2 were to be used then more resources could be allocated in the filters allowing the system to run faster and more effectively. Both of these upgrades would most likely have a cost drawback.

### Ethical

There are no real ethical implications that can arise as a direct result of this project. The platform is a highly modular learning tool and is designed to be only that.

### Health and Safety

With all electronic devices health and safety is of concern with use. This project should not interact with anything that is wet and/or submerged in water. This could damage the FPGA board and cause harm to anyone surrounding the device. This project also has the capabilities to damage eardrums if amplified loud enough using speakers. All aural testing of filters should be done in non-extensive time intervals, and at a reasonable volume.

### Social and Political

There are no direct social and/or political implications of the project to anything. Users could indirectly apply the filter realizations of the DSP core to political related audio files, but this is as a direct result of the user, not the project.

## Development

New development techniques I learned through the course of the project includes hardware integration through VHDL, building shift registers using circular buffer arrays, and sampling signals. Most of the aspects of DSP explored during the project were fairly new and/or vague before starting my research for the project design. This project served as a refresher course in digital design and VHDL programming, which most students in the CPE program don't touch on too much after CPE 233.

# Appendix C: VHDL Source Code

The following section contains all the VHDL source code built during the project.

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Design Name: Design Definitions
-- Module Name: DEFINITIONS
-- Project Name: DSP
-- Description: DEFINITIONS contains all component declarations used
-- throughout the project.
----------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package DEFINITIONS is
   type vect_16x20 is array (0 to 19) of std_logic_vector(15 downto 0);
   type vect_16x30 is array (0 to 29) of std_logic_vector(15 downto 0);

   -- CHANGE THE VALUE TO CHANGE FILTER
   -- CHANGE THE RANGE TO ADD MORE FILTERS TO THE DSP_CORE MODULE
   constant FILTER : integer range 0 to 2 := 2;

   -- SETS SAMPLE RATE IN THE IO_CTRL
   -- SAMP_RATE = 50 MHz / (2 * Desired Sampling Frequency)
   -- CHANGE SAMP_RATE TO CHANGE THE RATE AT WHICH THE DSP SYSTEM SAMPLES FROM THE ADC
   -- Use 566 for 44.1 kHz
   constant SAMP_RATE : integer range 1 to 50000000 := 566;

   -- DSP Filter Controller and Filter Selector ------------------------------------------------
   component DSP_CORE is
   port(
      CLK : in STD_LOGIC;
      RES : in STD_LOGIC;
      MISO : in STD_LOGIC;
      MOSI : out STD_LOGIC;
      ADC_CLK : out STD_LOGIC;
      DAC_CLK : out STD_LOGIC;
      SAM_CLK : out STD_LOGIC;
      CS : out STD_LOGIC;
      SYNC : out STD_LOGIC);
   end component;

   -- Filters ----------------------------------------------------------------------------------
   component DELAY_FILTER
   port(
      CLK : IN  std_logic;
      RES : IN  std_logic;
      SAMPLE_DONE : IN std_logic;
      X_REG : IN  vect_16x30;
      Y : OUT  std_logic_vector(11 downto 0);
      OUT_DONE : OUT  std_logic);
   end component;

   component ECHO_FILTER
   port(
      CLK : in   STD_LOGIC;
      RES : in   STD_LOGIC;
      SAMPLE_DONE : in   STD_LOGIC;
```

```
   X_REG : in   vect_16x30;
   Y_REG : in   vect_16x30;
   Y   : out  STD_LOGIC_VECTOR(11 downto 0);
   ADDER : out  STD_LOGIC_VECTOR(31 downto 0);
   OUT_DONE : out STD_LOGIC);
end component;

component REVERB_ALL_PASS_FILTER
port(
   CLK : in   STD_LOGIC;
   RES : in   STD_LOGIC;
   SAMPLE_DONE : in   STD_LOGIC;
   X_REG : in   vect_16x30;
   Y_REG : in   vect_16x30;
   Y : out  STD_LOGIC_VECTOR(11 downto 0);
   OUT_DONE : out STD_LOGIC);
end component;

component SYMMETRICAL_FORM is
port(
   CLK : IN  std_logic;
   RES : IN  std_logic;
   SAMPLE_DONE : IN std_logic;
   X_REG : IN  vect_16x30;
   Y : OUT  std_logic_vector(11 downto 0);
   OUT_DONE : OUT  std_logic);
end component;


-- IO Controller Modules ---------------------------------------------------------------
component CLK_DIV is
port (
   CLK : in STD_LOGIC;
   DIV : in integer;
   CLK_OUT : out STD_LOGIC);
end component;

component SHIFT_REGISTER is
port (
   CLK : in   STD_LOGIC;
   RES : in   STD_LOGIC;
   D   : in   STD_LOGIC_VECTOR(15 downto 0);
   VEC : out  vect_16x30);
end component;

component SPI_DAC_CTRL is
port (
   CLK : in  STD_LOGIC;
   RES : in  STD_LOGIC;
   WT : in  STD_LOGIC;
   Y : in  STD_LOGIC_VECTOR(11 downto 0);
   DAC_CLK : out STD_LOGIC;
   SYNC : out STD_LOGIC;
   MOSI : out STD_LOGIC);
end component;

component SPI_ADC_CTRL is
port (
   CLK : in  STD_LOGIC;
   RES : in  STD_LOGIC;
   RD : in  STD_LOGIC;
   MISO : in  STD_LOGIC;
   ADC_CLK : out STD_LOGIC;
   CS : out STD_LOGIC;
   X : out STD_LOGIC_VECTOR(11 downto 0);
```

```
   DONE : out STD_LOGIC);
end component;

component IO_CTRL is
port (
   CLK : in  STD_LOGIC;
   RES : in  STD_LOGIC;
   MISO : in  STD_LOGIC;
   Y : in  STD_LOGIC_VECTOR(11 downto 0);
   OUT_DONE   : in STD_LOGIC;
   ADC_CLK : out STD_LOGIC;
   DAC_CLK    : out STD_LOGIC;
   SAM_CLK : out STD_LOGIC;
   CS : out STD_LOGIC;
   SYNC : out STD_LOGIC;
   MOSI : out STD_LOGIC;
   X_SHIFT : out  vect_16x30;
   Y_SHIFT : out  vect_16x30;
   X : out STD_LOGIC_VECTOR(11 downto 0);
   IN_DONE : out STD_LOGIC);
end component;

-- ALU Controller Modules --------------------------------------------------------------------
component FULL_ADDER is
port(
   A  : in  std_logic;
   B  : in  std_logic;
   CI : in  std_logic;
   CO : out std_logic;
   S  : out std_logic );
end component;

component RC_ADDER_4BIT is
port(
   A : in std_logic_vector(3 downto 0);
   B : in std_logic_vector(3 downto 0);
   CI : in std_logic;
   CO : out std_logic;
   S : out std_logic_vector(3 downto 0));
end component;

component CSe_ADDER_32BIT is
port(
   A  : in  std_logic_vector(31 downto 0);
   B  : in  std_logic_vector(31 downto 0);
   CO : out std_logic;
   S  : out std_logic_vector(31 downto 0));
end component;

component BOOTH_MULT_16BIT is
port(
   A : in std_logic_vector(15 downto 0);
   B : in std_logic_vector(15 downto 0);
   P : out std_logic_vector(31 downto 0));
end component;

component OVERFLOW is
port(
   A : in  std_logic;
   B : in  std_logic;
   S_INT : in  std_logic_vector(31 downto 0);
   S_ADJ : out std_logic_vector(31 downto 0));
end component;
```

```
end package;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: DSP_CORE - Behavioral
-- Project Name: DSP
-- Description: Oversees entire project.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity DSP_CORE is
   port( CLK : in STD_LOGIC;
         RES : in STD_LOGIC;
         MISO : in STD_LOGIC;
         MOSI : out STD_LOGIC;
         ADC_CLK : out STD_LOGIC;
         DAC_CLK : out STD_LOGIC;
         SAM_CLK : out STD_LOGIC;
         CS : out STD_LOGIC;
         SYNC : out STD_LOGIC);
end DSP_CORE;

architecture Behavioral of DSP_CORE is
signal X_REG : vect_16x30 := (others => (others => '0'));
signal Y_REG : vect_16x30 := (others => (others => '0'));
signal SAMPLE_DONE : STD_LOGIC := '0';
signal OUT_DONE : STD_LOGIC := '0';
signal X : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');
signal Y : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');
signal ADDER : STD_LOGIC_VECTOR(31 downto 0);

begin
   delay: if FILTER = 0 generate
      delay : DELAY_FILTER PORT MAP(CLK, RES, SAMPLE_DONE, X_REG, Y, OUT_DONE);
   end generate delay;

   echo: if FILTER = 1 generate
      echo : ECHO_FILTER PORT MAP(CLK, RES, SAMPLE_DONE, X_REG, Y_REG, Y, ADDER, OUT_DONE);
   end generate echo;

   reverb_all_pass: if FILTER = 2 generate
      reverb_all_pass : REVERB_ALL_PASS_FILTER PORT MAP(CLK, RES, SAMPLE_DONE, X_REG, Y_REG, Y,
         OUT_DONE);
   end generate reverb_all_pass;

   sample: IO_CTRL PORT MAP(CLK, RES, MISO, Y, OUT_DONE, ADC_CLK, DAC_CLK, SAM_CLK, CS, SYNC,
                            MOSI, X_REG, Y_REG, X, SAMPLE_DONE);
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: DELAY_FILTER - Behavioral
-- Project Name: DSP
-- Description: Filter creates a delay effect.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity DELAY_FILTER is
   Port ( CLK : in STD_LOGIC;
          RES : in STD_LOGIC;
          SAMPLE_DONE : in   STD_LOGIC;
          X_REG : in vect_16x30;
          Y : out STD_LOGIC_VECTOR(11 downto 0);
          OUT_DONE : out STD_LOGIC);
end DELAY_FILTER;

architecture Behavioral of DELAY_FILTER is
signal Y_OUT  : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');
type FILTER_STATE is (STANDBY, DELAY, SEND);
signal PS : FILTER_STATE := STANDBY;
signal NS : FILTER_STATE := STANDBY;
signal DELAY_DONE : STD_LOGIC := '0';
signal SEND_DELAY : STD_LOGIC := '0';

begin
   Y <= Y_OUT;

   delay_proc : process(CLK, RES) is
   begin
      if RES = '1' then
         PS <= STANDBY;
      elsif rising_edge(CLK) then
         if DELAY_DONE = '1' then
            Y_OUT <= X_REG(29)(11 downto 0);
         end if;

         if SEND_DELAY = '1' then
            OUT_DONE <= '1';
         else
            OUT_DONE <= '0';
         end if;

         PS <= NS;
      end if;
   end process;

   delay_fsm : process(PS, SAMPLE_DONE) is
   begin
      case PS is
      when STANDBY =>
         DELAY_DONE <= '0';
         SEND_DELAY <= '0';
         if SAMPLE_DONE = '1' then
            NS <= DELAY;
         else
            NS <= STANDBY;
         end if;

      when DELAY =>
         DELAY_DONE <= '1';
         SEND_DELAY <= '0';
         NS <= SEND;

      when SEND =>
```

```
         DELAY_DONE <= '0';
         SEND_DELAY <= '1';
         NS <= STANDBY;

      when others =>
         NS <= STANDBY;

      end case;

   end process;
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: ECHO_FILTER - Behavioral
-- Project Name: DSP
-- Description: Filter creates an echo effect.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity ECHO_FILTER is
   Port ( CLK : in STD_LOGIC;
          RES : in STD_LOGIC;
          SAMPLE_DONE : in   STD_LOGIC;
          X_REG : in vect_16x30;
          Y_REG : in vect_16x30;
          Y : out  STD_LOGIC_VECTOR(11 downto 0);
          ADDER : out STD_LOGIC_VECTOR(31 downto 0);
          OUT_DONE : out STD_LOGIC);
end ECHO_FILTER;

architecture Behavioral of ECHO_FILTER is
signal Y_OUT  : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');
type FILTER_STATE is (STANDBY, MULT, MULT_HOLD, COMB, COMB_HOLD, HOLD, SEND);
signal PS : FILTER_STATE := STANDBY;
signal NS : FILTER_STATE := STANDBY;
signal MULT_FLAG : STD_LOGIC := '0';
signal COMB_FLAG : STD_LOGIC := '0';
signal DELAY_DONE : STD_LOGIC := '0';
signal SEND_DELAY : STD_LOGIC := '0';
-- Inputs to ALU modules from x_reg
signal MULT_AX : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal MULT_BX : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
-- Inputs to ALU modules from y_reg
signal MULT_AY : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal MULT_BY : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal ADDER_A : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_B : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_OUT : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_CO : STD_LOGIC := '0';

begin
   Y <= Y_OUT;
   ADDER <= ADDER_OUT;

   adder_module : CSe_ADDER_32BIT PORT MAP(ADDER_A, ADDER_B, ADDER_CO, ADDER_OUT);
   mult_module_x : BOOTH_MULT_16BIT PORT MAP(MULT_AX, MULT_BX, ADDER_A);
   mult_module_y : BOOTH_MULT_16BIT PORT MAP(MULT_AY, MULT_BY, ADDER_B); -- This is the echo

   delay_proc : process(CLK, RES) is
   begin
      if RES = '1' then
         PS <= STANDBY;
      elsif rising_edge(CLK) then
         if MULT_FLAG = '1' then
            MULT_AX <= X_REG(0);
            MULT_BX <= "1000000000000000";
            MULT_AY <= Y_REG(29);
            -- The greater this value is, the more times an attack echoes
            MULT_BY <= "0010000000000000";
         end if;
         if DELAY_DONE = '1' then
            Y_OUT <= ADDER_OUT(26 downto 15);
         end if;

         if SEND_DELAY = '1' then
            OUT_DONE <= '1';
```

```
         else
            OUT_DONE <= '0';
         end if;

         PS <= NS;
      end if;
   end process;

   delay_fsm : process(PS, SAMPLE_DONE) is
   begin
      case PS is
      when STANDBY =>
         MULT_FLAG <= '0';
         COMB_FLAG <= '0';
         DELAY_DONE <= '0';
         SEND_DELAY <= '0';
         if SAMPLE_DONE = '1' then
            NS <= MULT;
         else
            NS <= STANDBY;
         end if;

      when MULT =>
         MULT_FLAG <= '1';
         COMB_FLAG <= '0';
         DELAY_DONE <= '0';
         SEND_DELAY <= '0';
         NS <= MULT_HOLD;

      when MULT_HOLD =>
         MULT_FLAG <= '1';
         COMB_FLAG <= '0';
         DELAY_DONE <= '0';
         SEND_DELAY <= '0';
         NS <= COMB;

      when COMB =>
         MULT_FLAG <= '1';
         COMB_FLAG <= '1';
         DELAY_DONE <= '0';
         SEND_DELAY <= '0';
         NS <= COMB_HOLD;

      when COMB_HOLD =>
         MULT_FLAG <= '1';
         COMB_FLAG <= '1';
         DELAY_DONE <= '0';
         SEND_DELAY <= '0';
         NS <= HOLD;

      when HOLD =>
         MULT_FLAG <= '1';
         COMB_FLAG <= '1';
         DELAY_DONE <= '1';
         SEND_DELAY <= '0';
         NS <= SEND;

      when SEND =>
         MULT_FLAG <= '1';
         COMB_FLAG <= '1';
         DELAY_DONE <= '0';
         SEND_DELAY <= '1';
         NS <= STANDBY;

      when others =>
         NS <= STANDBY;
      end case;
   end process;
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: REVERB_ALL_PASS_FILTER - Behavioral
-- Project Name: DSP
-- Description: Module that synthesizes an input signal and produces an all pass
-- reverb filter.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity REVERB_ALL_PASS_FILTER is
    Port ( CLK : in  STD_LOGIC;
           RES : in  STD_LOGIC;
           SAMPLE_DONE : in  STD_LOGIC;
           X_REG : in  vect_16x30;
           Y_REG : in  vect_16x30;
           Y : out  STD_LOGIC_VECTOR(11 downto 0);
           OUT_DONE : out  STD_LOGIC);
end REVERB_ALL_PASS_FILTER;

architecture Behavioral of REVERB_ALL_PASS_FILTER is
signal Y_OUT  : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');
type FILTER_STATE is (STANDBY, MULT1, MULT_HOLD1, COMB1, COMB_HOLD1, MULT2, MULT_HOLD2, COMB2,
COMB_HOLD2, HOLD, SEND);
signal PS : FILTER_STATE := STANDBY;
signal NS : FILTER_STATE := STANDBY;
signal MULT_FLAG1 : STD_LOGIC := '0';
signal MULT_FLAG2 : STD_LOGIC := '0';
signal DELAY_DONE : STD_LOGIC := '0';
signal SEND_DELAY : STD_LOGIC := '0';
-- Inputs to ALU modules from x_reg
signal MULT_AX : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal MULT_BX : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
-- Inputs to ALU modules from y_reg
signal MULT_AY : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal MULT_BY : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal ADDER_A : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_B : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_OUT : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_CO : STD_LOGIC := '0';
signal CNTR_SIG : STD_LOGIC := '0';
signal D_SHIFT : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal D_REG : vect_16x30 := (others => (others => '0'));

begin
   Y <= Y_OUT;

   adder_module : CSe_ADDER_32BIT PORT MAP(ADDER_A, ADDER_B, ADDER_CO, ADDER_OUT);
   mult_module_x : BOOTH_MULT_16BIT PORT MAP(MULT_AX, MULT_BX, ADDER_A); -- This is the current
sample
   mult_module_y : BOOTH_MULT_16BIT PORT MAP(MULT_AY, MULT_BY, ADDER_B); -- This is the echo

   -- Set up interface with ADC Module    center_shift: SHIFT_REGISTER PORT MAP(CNTR_SIG, RES,
D_SHIFT, D_REG);

   delay_proc : process(CLK, RES) is
   begin
      if RES = '1' then
         PS <= STANDBY;
      elsif rising_edge(CLK) then
         if MULT_FLAG1 = '1' then
            MULT_AX <= X_REG(0);
            MULT_BX <= "1000000000000000";
            MULT_AY <= D_REG(1);
            -- The greater this value is, the more times an attack echoes
            MULT_BY <= "0000010000000000";
```

```
         D_SHIFT <= ADDER_OUT(26 downto 15);
      end if;

      if MULT_FLAG2 = '1' then
         MULT_AX <= D_REG(0);
         MULT_BX <= "1000000000000000";
         MULT_AY <= D_REG(1);
         -- The greater this value is, the more times an attack echoes
         MULT_BY <= "0000010000000000";
      end if;

      if DELAY_DONE = '1' then
         Y_OUT <= ADDER_OUT(26 downto 15);
      end if;

      if SEND_DELAY = '1' then
         OUT_DONE <= '1';
      else
         OUT_DONE <= '0';
      end if;

      PS <= NS;
   end if;
end process;

delay_fsm : process(PS, SAMPLE_DONE) is
begin
   case PS is
   when STANDBY =>
      MULT_FLAG1 <= '0';
      MULT_FLAG2 <= '0';
      DELAY_DONE <= '0';
      SEND_DELAY <= '0';
      CNTR_SIG <= '0';
      if SAMPLE_DONE = '1' then
         NS <= MULT1;
      else
         NS <= STANDBY;
      end if;

   when MULT1 =>
      MULT_FLAG1 <= '1';
      MULT_FLAG2 <= '0';
      DELAY_DONE <= '0';
      SEND_DELAY <= '0';
      NS <= MULT_HOLD1;

   when MULT_HOLD1 =>
      MULT_FLAG1 <= '1';
      MULT_FLAG2 <= '0';
      DELAY_DONE <= '0';
      SEND_DELAY <= '0';
      NS <= COMB1;

   when COMB1 =>
      MULT_FLAG1 <= '1';
      MULT_FLAG2 <= '0';
      DELAY_DONE <= '0';
      SEND_DELAY <= '0';
      NS <= COMB_HOLD1;

   when COMB_HOLD1 =>
      MULT_FLAG1 <= '1';
      MULT_FLAG2 <= '0';
      CNTR_SIG <= '1';
      DELAY_DONE <= '0';
      SEND_DELAY <= '0';
      NS <= MULT2;

   when MULT2 =>
      MULT_FLAG1 <= '0';
      MULT_FLAG2 <= '1';
      CNTR_SIG <= '0';
```

```
            DELAY_DONE <= '0';
            SEND_DELAY <= '0';
            NS <= MULT_HOLD2;

         when MULT_HOLD2 =>
            MULT_FLAG1 <= '0';
            MULT_FLAG2 <= '1';
            DELAY_DONE <= '0';
            SEND_DELAY <= '0';
            NS <= COMB2;

         when COMB2 =>
            MULT_FLAG1 <= '0';
            MULT_FLAG2 <= '1';
            DELAY_DONE <= '0';
            SEND_DELAY <= '0';
            NS <= COMB_HOLD2;

         when COMB_HOLD2 =>
            MULT_FLAG1 <= '0';
            MULT_FLAG2 <= '1';
            DELAY_DONE <= '0';
            SEND_DELAY <= '0';
            NS <= HOLD;

         when HOLD =>
            MULT_FLAG1 <= '0';
            MULT_FLAG2 <= '1';
            DELAY_DONE <= '1';
            SEND_DELAY <= '0';
            NS <= SEND;

         when SEND =>
            MULT_FLAG1 <= '0';
            MULT_FLAG2 <= '1';
            DELAY_DONE <= '0';
            SEND_DELAY <= '1';
            NS <= STANDBY;

         when others =>
            NS <= STANDBY;

         end case;
      end process;
   end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: SYMMETRICAL_FORM - Behavioral
-- Project Name: DSP
-- Description: Filter creates an echo effect.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity SYMMETRICAL_FORM is
    PORT(
        CLK : IN  std_logic;
        RES : IN  std_logic;
        SAMPLE_DONE : IN std_logic;
        X_REG : IN  vect_16x30;
        Y : OUT  std_logic_vector(11 downto 0);
        OUT_DONE : OUT  std_logic);
    end SYMMETRICAL_FORM;

architecture Behavioral of SYMMETRICAL_FORM is
signal Y_OUT  : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal TESTER_Y  : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
type FILTER_STATE is (STANDBY, COMB, COMB_HOLD, COMB_HOLD_AGAIN, MULT, MULT_HOLD, COMB2,
COMB_HOLD2, HOLD, SEND);
signal PS : FILTER_STATE := STANDBY;
signal NS : FILTER_STATE := STANDBY;
signal mult_flag : STD_LOGIC := '0';
signal SEND_DELAY : STD_LOGIC := '0';
-- Inputs to ALU modules from x_reg
signal MULT07 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal MULT07_OUT : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal MULT16 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal MULT16_OUT : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal MULT25 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal MULT25_OUT : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal MULT34 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal MULT34_OUT : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal ADDER_B0 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_B1 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_B2 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_B3 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_B01 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_B012 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal ADDER_CO : STD_LOGIC := '0';
-- Integer denoting where to take the sample from
signal N : integer := 11;
constant LEAD_ZEROS : STD_LOGIC_VECTOR(15 downto 0) := (others  => '0');
constant B0 : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000001";
constant B1 : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000001";
constant B2 : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000001";
constant B3 : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000001";
-- X_REG as 32-bit
signal X_REG32 : vect_32x8 := (others => (others => '0'));
-- Counter
signal counter : integer range 0 to 400 := 0;

begin
    Y <= Y_OUT(N downto (N - 11));
    X_REG32(0) <= LEAD_ZEROS & X_REG(0);
    X_REG32(1) <= LEAD_ZEROS & X_REG(1);
    X_REG32(2) <= LEAD_ZEROS & X_REG(2);
    X_REG32(3) <= LEAD_ZEROS & X_REG(3);
    X_REG32(4) <= LEAD_ZEROS & X_REG(4);
    X_REG32(5) <= LEAD_ZEROS & X_REG(5);
    X_REG32(6) <= LEAD_ZEROS & X_REG(6);
    X_REG32(7) <= LEAD_ZEROS & X_REG(7);
```

```
MULT07_OUT <= MULT07(15 downto 0);

adder_module07 : CSe_ADDER_32BIT PORT MAP(X_REG32(0), X_REG32(7), ADDER_CO, MULT07);
adder_module16 : CSe_ADDER_32BIT PORT MAP(X_REG32(1), X_REG32(6), ADDER_CO, MULT16);
adder_module25 : CSe_ADDER_32BIT PORT MAP(X_REG32(2), X_REG32(5), ADDER_CO, MULT25);
adder_module34 : CSe_ADDER_32BIT PORT MAP(X_REG32(3), X_REG32(4), ADDER_CO, MULT34);
mult_module07 : BOOTH_MULT_16BIT PORT MAP(MULT07_OUT, B0, ADDER_B0);
mult_module16 : BOOTH_MULT_16BIT PORT MAP(MULT16_OUT, B1, ADDER_B1);
mult_module25 : BOOTH_MULT_16BIT PORT MAP(MULT25_OUT, B2, ADDER_B2);
mukt_module34 : BOOTH_MULT_16BIT PORT MAP(MULT34_OUT, B3, ADDER_B3);
adder_module_b01 : CSe_ADDER_32BIT PORT MAP(MULT07, MULT16, ADDER_CO, ADDER_B01);
adder_module_b012 : CSe_ADDER_32BIT PORT MAP(ADDER_B01, MULT25, ADDER_CO, ADDER_B012);
adder_module_b0123 : CSe_ADDER_32BIT PORT MAP(ADDER_B012, MULT34, ADDER_CO, Y_OUT);

delay_proc : process(CLK, RES) is
begin
    if RES = '1' then
        PS <= STANDBY;
    elsif rising_edge(CLK) then
        if SEND_DELAY = '1' then
            OUT_DONE <= '1';
        else
            OUT_DONE <= '0';
        end if;

        if mult_flag = '1' then
            MULT07_OUT <= MULT07(15 downto 0);
            MULT16_OUT <= MULT16(15 downto 0);
            MULT25_OUT <= MULT25(15 downto 0);
            MULT34_OUT <= MULT34(15 downto 0);
            --Y_OUT(15 downto 0) <= MULT07_OUT;
            --Y_OUT <= MULT16;
        end if;

        if PS = HOLD then
            --Y <= ADDER_B0(11 downto 0);
            counter <= counter + 1;
        else
            counter <= 0;
        end if;

        PS <= NS;
    end if;
end process;

delay_fsm : process(PS, SAMPLE_DONE, counter) is
begin
    case PS is
    when STANDBY =>
        SEND_DELAY <= '0';
        mult_flag <= '0';
        if SAMPLE_DONE = '1' then
            NS <= COMB;
        else
            NS <= STANDBY;
        end if;

    when COMB =>
       SEND_DELAY <= '0';
        mult_flag <= '0';
        NS <= COMB_HOLD;

    when COMB_HOLD =>
        SEND_DELAY <= '0';
        mult_flag <= '0';
        NS <= COMB_HOLD_AGAIN;

    when COMB_HOLD_AGAIN =>
        SEND_DELAY <= '0';
        mult_flag <= '0';
        NS <= MULT;

    when MULT =>
```

```
            SEND_DELAY <= '0';
            mult_flag <= '1';
            NS <= MULT_HOLD;

        when MULT_HOLD =>
            SEND_DELAY <= '0';
            mult_flag <= '1';
            NS <= COMB2;

        when COMB2 =>
            SEND_DELAY <= '0';
            mult_flag <= '1';
            NS <= COMB_HOLD2;

        when COMB_HOLD2 =>
            SEND_DELAY <= '0';
            mult_flag <= '1';
            NS <= HOLD;

        when HOLD =>
            SEND_DELAY <= '0';
            mult_flag <= '1';
            if counter = 400 then
                NS <= SEND;
            else
                NS <= HOLD;
            end if;

        when SEND =>
            SEND_DELAY <= '1';
            NS <= STANDBY;

        when others =>
            NS <= STANDBY;

        end case;
    end process;
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California PolyTechnic State University
-- Engineer: Kevin Brewer
--
-- Design Name: Clock Divider Module
-- Module Name: CLK_DIV - Behavioral
-- Project Name: DSP
-- Description: Module takes in a clock signal to be divided and an integer value
-- to divide by.  Output is the resulting clock signal.  Operations occure with
-- input clock.  Module used for syncronization with ADC and DAC devices.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CLK_DIV is
   Port ( CLK : in  STD_LOGIC;
          DIV : in  integer;
          CLK_OUT : out  STD_LOGIC);
end CLK_DIV;

architecture Behavioral of CLK_DIV is

begin
   -- Process changes states when the counter reaches the DIV input specified
   clk_divider: process(CLK) is
   -- Buffer for the output clock state
   variable CLK_OUT_TEMP : std_logic := '0';
   -- Counter variable used to compare against DIV intput integer
   variable CNT : integer := 1;

   begin
      if rising_edge(CLK) then
         -- If not at DIV, increment CNT
         if CNT < DIV then
            CNT := CNT + 1;
         -- Check to see if CNT has reached DIV
         else
            -- Flip the state of the output clock buffer bit
            CLK_OUT_TEMP := not(CLK_OUT_TEMP);
            -- Reset counter for the next clock state change
            CNT := 1;
         end if;

         -- State change only occurs when CNT reaches DIV
         CLK_OUT <= CLK_OUT_TEMP;
      end if;
   end process clk_divider;
end behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: SHIFT_REGISTER - Behavioral
-- Project Name: DSP
-- Description: Shift register that behaves as a circular buffer.  This allows a
-- filter to reference past values of both input and ouput frequencies.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity SHIFT_REGISTER is
    Port ( CLK : in   STD_LOGIC;
           RES : in   STD_LOGIC;
           D   : in   STD_LOGIC_VECTOR(15 downto 0);
           VEC : out  vect_16x30);
end SHIFT_REGISTER;

architecture behavioral of SHIFT_REGISTER is
signal VEC_TMP : vect_16x30;

begin
   VEC <= VEC_TMP;

   circular : process (CLK, RES, D) is
   begin
      if RES = '1' then
         for i in 29 downto 0 loop
            VEC_TMP(i) <= "0000000000000000";
         end loop;
      elsif rising_edge(CLK) then
         VEC_TMP(1 to 29) <= VEC_TMP(0 to 28);
         VEC_TMP(0) <= D;
      end if;
   end process;
end behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: SPI_DAC_CTRL - Behavioral
-- Project Name: DSP
-- Description: Manages the DA2 Pmod by Digilent.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity SPI_DAC_CTRL is
   Port ( CLK : in  STD_LOGIC;
          RES : in  STD_LOGIC;
          WT : in  STD_LOGIC;
          Y : in  STD_LOGIC_VECTOR(11 downto 0);
          DAC_CLK : out STD_LOGIC;
          SYNC : out STD_LOGIC;
          MOSI : out STD_LOGIC);
end SPI_DAC_CTRL;

architecture Behavioral of SPI_DAC_CTRL is
-- Signals that map the clock divided signals to the DAC
signal DAC_CLK_TMP : STD_LOGIC := '0';
-- Signals that hold the output to write to the DAC
-- NOTE: 16, as opposed to 12, is used for reading because the DAC has 4 leading zeros
signal DAC_REGISTER : STD_LOGIC_VECTOR(15 downto 0);
signal DAC_REGISTER_CNT : integer range 0 to 15 := 0;
-- Finite States for DAC
type DAC_state is (IDLE_DAC, SETUP_DAC, START_DAC);
signal PS : DAC_state;
signal NS : DAC_state;
-- Flags indicating when Y are done being sampled
signal Y_FLAG : STD_LOGIC := '0';

begin
   DAC_CLK <= DAC_CLK_TMP;

   -- Create the clock cycles that work with the I/O Pmods
   -- Change the second argument to obtain different clock cycles!
   -- NOTE: Using the Nexys2, this will create a DAC_CLK = 12.5 Mhz
   dac_sample : CLK_DIV port map(CLK, 2, DAC_CLK_TMP);

   -- Based on the design of the adders and multipliers the MSB is equivalent to MOSI
   MOSI <= DAC_REGISTER(15);

   -- Update FSM for DAC Controller
   dac_state_change : process(RES, DAC_CLK_TMP, NS) is
   begin
      if RES = '1' then
         PS <= IDLE_DAC;
      elsif rising_edge(DAC_CLK_TMP) then
         -- Change to next finite state
         PS <= NS;
      end if;
   end process dac_state_change;

   dac_state_ctrl : process(PS, WT, DAC_REGISTER_CNT) is
```

```
   begin
      case PS is
      when IDLE_DAC =>
         SYNC <= '0';
         Y_FLAG <= '0';
         if WT = '1' then
            NS <= SETUP_DAC;
         else
            NS <= IDLE_DAC;
         end if;

      when SETUP_DAC =>
         SYNC <= '1';
         Y_FLAG <= '1';
         NS <= START_DAC;

      when START_DAC =>
         SYNC <= '0';
         Y_FLAG <= '0';
         if DAC_REGISTER_CNT = 15 then
            -- DONE! Get another sample!
            NS <= SETUP_DAC;
         elsif DAC_REGISTER_CNT < 15 then
            -- Not done, stay in this state
            NS <= START_DAC;
         end if;

      when others =>
         NS <= IDLE_DAC;
      end case;
   end process dac_state_ctrl;

   dac_register_ctrl : process(DAC_CLK_TMP, RES) is
   begin
      if RES = '1' then
         DAC_REGISTER <= "0000000000000000";
      elsif rising_edge(DAC_CLK_TMP) then
         if Y_FLAG = '1' then
            DAC_REGISTER(11 downto 0) <= Y;
         end if;

         if PS = START_DAC then
            -- Replaces old value with all 0's
            DAC_REGISTER <= DAC_REGISTER(14 downto 0) & '0';
            -- Increment counter
            DAC_REGISTER_CNT <= DAC_REGISTER_CNT + 1;
         else
            DAC_REGISTER_CNT <= 0;
         end if;
      end if;
   end process dac_register_ctrl;
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: SPI_CTRL - Behavioral
-- Project Name: DSP
-- Description: Manages the AD1 Pmod by Digilent.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity SPI_ADC_CTRL is
    Port ( CLK       : in  STD_LOGIC;
           RES       : in  STD_LOGIC;
           RD          : in  STD_LOGIC;
           MISO       : in  STD_LOGIC;
           ADC_CLK    : out STD_LOGIC;
           CS          : out STD_LOGIC;
           X          : out STD_LOGIC_VECTOR(11 downto 0);
           DONE       : out STD_LOGIC);
end SPI_ADC_CTRL;

architecture Behavioral of SPI_ADC_CTRL is
-- Signals that map the clock divided signals to the ADC
signal ADC_CLK_TMP : STD_LOGIC := '0';
-- Signals that hold the input read from the ADC
-- NOTE: 16, as opposed to 12, is used for reading because the DAC has 4 leading zeros
signal ADC_REGISTER : STD_LOGIC_VECTOR(15 downto 0);
signal ADC_REGISTER_CNT : integer range 0 to 15 := 0;
-- Finite States for ADC
type ADC_state is (IDLE_ADC, SETUP_ADC, START_ADC, FINISH_READ);
signal PS : ADC_state := IDLE_ADC;
signal NS : ADC_state := IDLE_ADC;
-- Flag indicating when X is done being sampled
signal X_FLAG : STD_LOGIC := '0';
-- Additional flag for when sampling is done to allow sample to wait one clock cycle
signal AD_SAMPLE : STD_LOGIC := '0';

begin
   ADC_CLK <= ADC_CLK_TMP;
   -- Create the clock cycles that work with the I/O Pmods
   -- Change the second argument to obtain different clock cycles!
   -- NOTE: Using the Nexys2, this will create an ADC_CLK = 50 Mhz
   adc_sample : CLK_DIV port map(CLK, 1, ADC_CLK_TMP);

   -- Update FSM for ADC Controller
   adc_state_change : process(RES, ADC_CLK_TMP, NS) is
   begin
      if RES = '1' then
         PS <= IDLE_ADC;
      elsif rising_edge(ADC_CLK_TMP) then
         -- Change to next finite state
         PS <= NS;
      end if;
   end process adc_state_change;

   adc_state_ctrl : process(PS, RD, ADC_REGISTER_CNT) is
   begin
```

```
   case PS is
   when IDLE_ADC =>
      CS <= '0';
      AD_SAMPLE <= '0';
      X_FLAG <= '0';
      if RD = '1' then
         NS <= SETUP_ADC;
      else
         NS <= IDLE_ADC;
      end if;

   when SETUP_ADC =>
      CS <= '1';
      AD_SAMPLE <= '0';
      X_FLAG <= '0';
      NS <= START_ADC;

   when START_ADC =>
      CS <= '0';
      AD_SAMPLE <= '0';
      if ADC_REGISTER_CNT = 15 then
         X_FLAG <= '1';
         NS <= FINISH_READ;
      elsif ADC_REGISTER_CNT < 15 then
         X_FLAG <= '0';
         NS <= START_ADC;
      end if;

   -- This state is necessary for ADC to allow one clock cycle before adding
   -- sample to input shift register!
   when FINISH_READ =>
      CS <= '0';
      AD_SAMPLE <= '1';
      X_FLAG <= '0';
      NS <= IDLE_ADC;

   when others =>
      NS <= IDLE_ADC;

   end case;
end process adc_state_ctrl;

adc_register_ctrl : process(ADC_CLK_TMP, RES) is
begin
   -- Always check for a reset before checking rising edges
   if RES = '1' then
      ADC_REGISTER <= "0000000000000000";
   elsif rising_edge(ADC_CLK_TMP) then
      if X_FLAG = '1' then
         X <= ADC_REGISTER(11 downto 0);
      end if;

      if AD_SAMPLE = '1' then
         DONE <= '1';
      elsif AD_SAMPLE = '0' then
         DONE <= '0';
      end if;

      if PS = START_ADC then
         ADC_REGISTER <= ADC_REGISTER(14 downto 0) & MISO;
         ADC_REGISTER_CNT <= ADC_REGISTER_CNT + 1;
      else
         ADC_REGISTER_CNT <= 0;
```

```
         end if;
      end if;
   end process adc_register_ctrl;
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Module Name: IO_CTRL - Behavioral
-- Project Name: DSP
-- Description: Controles the ADC and DAC controle modules and samples digital
-- input values given a provided sampling frequency.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

entity IO_CTRL is
    Port ( CLK  : in  STD_LOGIC;
           RES  : in  STD_LOGIC;
           MISO : in  STD_LOGIC;
           Y   : in  STD_LOGIC_VECTOR(11 downto 0);
           OUT_DONE  : in STD_LOGIC;
           ADC_CLK : out STD_LOGIC;
           DAC_CLK  : out STD_LOGIC;
           SAM_CLK : out STD_LOGIC;
           CS   : out STD_LOGIC;
           SYNC  : out STD_LOGIC;
           MOSI  : out STD_LOGIC;
           X_SHIFT : out  vect_16x30;
           Y_SHIFT : out  vect_16x30;
           X : out STD_LOGIC_VECTOR(11 downto 0);
           IN_DONE : out STD_LOGIC);
end IO_CTRL;

architecture Behavioral of IO_CTRL is
-- Clock signal that controles sampling rate
signal SAMPLE_CLK : STD_LOGIC := '0';
-- Flag to allow ADC to read
signal RD : STD_LOGIC := '0';
signal RD_FLAG : STD_LOGIC := '0';
-- Flag to allow DAC to write
signal WT : STD_LOGIC := '0';
signal WT_FLAG : STD_LOGIC := '0';
-- Finite States for IO Controller

type IO_state is (HOLD_IO, SAMPLE_IO);
--type IO_state is (HOLD_IO, SAMPLE_IO);
signal PS : IO_state;
signal NS : IO_state;
-- Start sampling flags
signal NEW_SAMPLE : STD_LOGIC := '0';
signal GET_SAMPLE : STD_LOGIC := '0';
signal SAMPLE_DONE_FLAG : STD_LOGIC := '0';
-- Signal to connect the output of adc controller to be the input of the input shift reg
signal X_TMP : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');

-- Shift register signals
signal X_SHIFT_ZERO : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal Y_SHIFT_ZERO : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal Y_SHIFT_TMP : vect_16x30 := (others => (others => '0'));

begin
```

```
IN_DONE <= SAMPLE_DONE_FLAG;
X <= X_TMP;
-- Concatenate leading zeros to the sampled input and output values for shift reg
X_SHIFT_ZERO <= "0000" & X_TMP;
Y_SHIFT_ZERO <= "0000" & Y;
Y_SHIFT <= Y_SHIFT_TMP;

SAM_CLK <= SAMPLE_CLK;

-- Set up sampling rate for DSP
sampling_rate: CLK_DIV PORT MAP(CLK, SAMP_RATE, SAMPLE_CLK);
-- Set up interface with ADC Module
adc_controler: SPI_ADC_CTRL PORT MAP(CLK, RES, RD, MISO, ADC_CLK, CS, X_TMP,
                                     SAMPLE_DONE_FLAG);
-- Set up interface with DAC Module
dac_controler: SPI_DAC_CTRL PORT MAP(CLK, RES, WT, Y_SHIFT_TMP(0)(11 downto 0), DAC_CLK, SYNC,
                                     MOSI);
-- Set up shift registers
shift_input: SHIFT_REGISTER PORT MAP(SAMPLE_DONE_FLAG, RES, X_SHIFT_ZERO, X_SHIFT);
shift_output: SHIFT_REGISTER PORT MAP(OUT_DONE, RES, Y_SHIFT_ZERO, Y_SHIFT_TMP);

sampler_proc : process (CLK, RES, NS) is
begin
   if RES = '1' then
      PS <= SAMPLE_IO;
      GET_SAMPLE <= '0';
   elsif rising_edge(CLK) then
      if RD_FLAG = '1' then
         RD <= '1';
      elsif RD_FLAG = '0' then
         RD <= '0';
      end if;

      if WT_FLAG = '1' then
         WT <= '1';
      elsif WT_FLAG = '0' then
         WT <= '0';
      end if;

      if SAMPLE_CLK = '1' then
         GET_SAMPLE <= '1';
         if SAMPLE_DONE_FLAG = '1' then
            NEW_SAMPLE <= '0';
         end if;
      elsif SAMPLE_CLK = '0' then
         GET_SAMPLE <= '0';
         NEW_SAMPLE <= '1';
      end if;

      PS <= NS;
   end if;
end process sampler_proc;

sampler_fsm : process (PS, GET_SAMPLE, NEW_SAMPLE, SAMPLE_DONE_FLAG) is
begin
   case PS is
   when HOLD_IO =>
      WT_FLAG <= '1';
      RD_FLAG <= '0';
      if (NEW_SAMPLE and GET_SAMPLE) = '1' then
         NS <= SAMPLE_IO;
      elsif (NEW_SAMPLE and GET_SAMPLE) = '0' then
         NS <= HOLD_IO;
```

```
         end if;

      when SAMPLE_IO =>
         WT_FLAG <= '1';
         if SAMPLE_DONE_FLAG = '1' then
            RD_FLAG <= '0';
            NS <= HOLD_IO;
         elsif SAMPLE_DONE_FLAG = '0' then
            RD_FLAG <= '1';
            NS <= SAMPLE_IO;
         end if;

      when others =>
         NS <= HOLD_IO;

      end case;

   end process sampler_fsm;
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: Cal Poly
-- Engineer: Kevin Brewer
--
-- Design Name: 1-bit Full Adder
-- Module Name: FULL_ADDER - behavioral
-- Project Name: DSP_CORE
-- Description: Takes two 1-bit numbers and a carry-in bit, and returns sum and
-- carry out bit.
----------------------------------------------------------------------------------
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library work;
use work.DEFINITIONS.all;

entity FULL_ADDER is
   port( A  : in  std_logic;
         B  : in  std_logic;
         CI : in  std_logic;
         CO : out std_logic;
         S  : out std_logic );
end FULL_ADDER;

architecture behavioral of FULL_ADDER is
begin
   S <= A xor B xor CI;
   CO <= ((A or B) and CI) or (A and B);
end behavioral;
```

# Appendix D: Leveraged VHDL Source Code

The following appendix contains the leveraged VHDL code.

```
----------------------------------------------------------------------------------
-- Company: Cal Poly
-- Engineer: Joseph Waddell
--
-- Design Name: 4-bit Ripple-Carry Adder
-- Module Name: RC_ADDER_16BIT - behavioral
-- Project Name: senior_project
-- Description: 4-bit Ripple-Carry Adder accepts two 4-bit inputs
-- and calculates their sum by using ripple-carry method.
----------------------------------------------------------------------------------
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library work;
use work.DEFINITIONS.all;

entity RC_ADDER_4BIT is
   port( A : in std_logic_vector(3 downto 0);
         B : in std_logic_vector(3 downto 0);
         CI : in std_logic;
         CO : out std_logic;
         S : out std_logic_vector(3 downto 0) );
end RC_ADDER_4BIT;

architecture behavioral of RC_ADDER_4BIT is
-- internal carries generated by full adders
signal CO_INT : std_logic_vector(3 downto 0) := (others => '0');

-- Ripple-Carry Adder
begin
   -- generate adders in ripple-carry format
   RC : for i in 0 to 3 generate
   -- generate half adder for bit 0 of sum
   RC0 : if i = 0 generate
   FA0 : FULL_ADDER
   port map( A(i), B(i), CI, CO_INT(i), S(i) );
   end generate RC0;

   -- generate full adders for bits 1 to 3 of sum
   RC1to3 : if i > 0 generate
   FA1to3 : FULL_ADDER
   port map( A(i), B(i), CO_INT(i - 1), CO_INT(i), S(i) );
   end generate RC1to3;
   end generate RC;

   -- set carry-out of adder
   CO <= CO_INT(3);
end behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: Cal Poly
-- Engineer: Joseph Waddell
--
-- Design Name: 32-bit Carry-Select Adder
-- Module Name: CSe_ADDER_16BIT - behavioral
-- Project Name: senior_project
-- Description: 32-bit Carry-Select Adder accepts two 32-bit inputs
-- and calculates their sum by using carry-select method.
----------------------------------------------------------------------------------
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library work;
use work.DEFINITIONS.all;

entity CSe_ADDER_32BIT is
   port( A  : in  std_logic_vector(31 downto 0);
         B  : in  std_logic_vector(31 downto 0);
         CO : out std_logic;
         S  : out std_logic_vector(31 downto 0) );
end CSe_ADDER_32BIT;

architecture behavioral of CSe_ADDER_32BIT is
-- sums of each ripple-carry block
signal S0, S1 : std_logic_vector(31 downto 0) := (others => '0');
signal S_INT, S_ADJ : std_logic_vector(31 downto 0) := (others => '0');
-- carries for each block
signal C, C0, C1 : std_logic_vector(7 downto 0) := (others => '0');

-- Carry-Select Adder
begin
   S <= S_ADJ;
   -- generate ripple-carry blocks
   CSe : for i in 0 to 7 generate
      -- ripple-carry block 0 for 4 LSBs of sum
      CSe0 : if i = 0 generate
         RC_blk0 : RC_ADDER_4BIT
         port map( A(4 * i + 3 downto 4 * i), B(4 * i + 3 downto 4 * i),
                   '0', C(i), S_INT( 4 * i + 3 downto 4 * i) );
      end generate CSe0;
      -- ripple-carry blocks 1 to 7 for remaining bits of sum
      CSe1to7 : if i > 0 generate
         -- ripple-carry sum for carry in of 0
         RC_blk1to7_0 : RC_ADDER_4BIT
         port map( A(4 * i + 3 downto 4 * i), B(4 * i + 3 downto 4 * i),
                   '0', C0(i), S0( 4 * i + 3 downto 4 * i) );

         -- ripple-carry sum for carry in of 1
         RC_blk1to7_1 : RC_ADDER_4BIT
         port map( A(4 * i + 3 downto 4 * i), B(4 * i + 3 downto 4 * i),
                   '1', C1(i), S1( 4 * i + 3 downto 4 * i) );
      end generate CSe1to7;
   end generate CSe;

   -- set carries for each block of adder
   carries : for i in 1 to 7 generate
      C(i) <= (C(i-1) and C1(i)) or C0(i);
   end generate carries;

   -- finalize sum based on carries from each block
```

```
   set_sum : for i in 1 to 7 generate
      S_INT(4 *i + 3 downto 4 * i) <= S0(4 * i + 3 downto 4 * i)
      when C(i - 1) = '0' else
         S1(4 * i + 3 downto 4 * i)
      when C(i - 1) = '1';

   end generate set_sum;

   -- set carry-out of adder
   CO <= C(7);

   -- adjust sum for overflow
   OV : OVERFLOW
   port map( A(31), B(31), S_INT, S_ADJ );
end behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: Cal Poly
-- Engineer: Joseph Waddell
--
-- Design Name: 16-bit Modified Booth Multiplier
-- Module Name: BOOTH_MULT_16BIT - behavioral
-- Project Name: senior_project
-- Description: 16-bit Modified Booth Multiplier accepts two 16-bit inputs
-- and computes their 32-bit product using the Booth algorithm.
----------------------------------------------------------------------------------
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library work;
use work.DEFINITIONS.all;
use work.CTRL_CONSTANTS.all;

entity BOOTH_MULT_16BIT is
   port( A : in std_logic_vector(15 downto 0);
         B : in std_logic_vector(15 downto 0);
         P : out std_logic_vector(31 downto 0) );
end BOOTH_MULT_16BIT;

architecture behavioral of BOOTH_MULT_16BIT is
begin
   -- modified Booth multiplier
   mult_proc : process(A, B) is
   -- temp product
   variable P_INT : std_logic_vector(36 downto 0) := (others => '0');
   -- 2 times multiplicand
   variable A2 : std_logic_vector(17 downto 0) := (others => '0');
   -- -2 times multiplicand
   variable NA2 : std_logic_vector(17 downto 0) := (others => '0');
   -- -multiplicand
   variable NA : std_logic_vector(17 downto 0) := (others => '0');
   -- padded multiplier and multiplicand
   variable A_INT : std_logic_vector(17 downto 0) := (others => '0');
   variable B_INT : std_logic_vector(17 downto 0) := (others => '0');
   begin
      -- pad multiplier and multiplicand
      A_INT := "00" & A;
      B_INT := "00" & B;

      -- assign potential partial products
      NA := not(A_INT) + 1;
      A2 := A_INT(16 downto 0) & '0';
      NA2 := not(A2) + 1;

      -- place multiplier in right half of product recister
      P_INT(18 downto 1) := B_INT;
         -- perform booth algorithm
         for i in 0 to 8 loop
            case P_INT(2 downto 0) is
            when "001" =>
               P_INT(36 downto 19) := P_INT(36 downto 19) + A_INT;
            when "010" =>
               P_INT(36 downto 19) := P_INT(36 downto 19) + A_INT;
            when "101" =>
               P_INT(36 downto 19) := P_INT(36 downto 19) + NA;
            when "110" =>
               P_INT(36 downto 19) := P_INT(36 downto 19) + NA;
```

```
            when "011" =>
               P_INT(36 downto 19) := P_INT(36 downto 19) + A2;
            when "100" =>
               P_INT(36 downto 19) := P_INT(36 downto 19) + NA2;
            when others =>
               null;
            end case;
            -- shift product register
            P_INT(34 downto 0) := P_INT(36 downto 2);
         end loop;

      -- set final value of product
      P <= P_INT(32 downto 1);
   end process mult_proc;
end behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: Cal Poly
-- Engineer: Joseph Waddell
--
-- Design Name: Overflow Detection
-- Module Name: OVERFLOW - behavioral
-- Project Name: senior_project
-- Description: OVERFLOW checks and corrects for overflow resulting from the
-- summation of two signals.
----------------------------------------------------------------------------------
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library work;
use work.DEFINITIONS.all;

entity OVERFLOW is
   port( A     : in  std_logic;
         B     : in  std_logic;
         S_INT : in  std_logic_vector(31 downto 0);
         S_ADJ : out std_logic_vector(31 downto 0) );
end OVERFLOW;

architecture behavioral of OVERFLOW is

begin
   ovflow_proc : process( A, B, S_INT )
   begin
      -- check if sum is positive
      if S_INT(31) = '0' then
         -- good sum value
         if ((A nor B) xor (A xor B)) = '1' then
            S_ADJ <= S_INT;
         -- negative overflow
         elsif (A and B) = '1' then
            S_ADJ(31) <= '1';
            S_ADJ(30 downto 0) <= (others => '0');
         end if;
      -- check if sum is negative
      elsif S_INT(31) = '1' then
         -- good sum value
         if ((A and B) xor (A xor B)) = '1' then
            S_ADJ <= S_INT;
         -- positive overflow
         elsif (A nor B) = '1' then
            S_ADJ(31) <= '0';
            S_ADJ(30 downto 0) <= (others => '1');
         end if;
      end if;
   end process ovflow_proc;
end behavioral;
```

# Appendix E: VHDL Test Benches

The following appendix contains the test bench code used to verify the DSP core project.

```
--------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Design Name:   DSP_CORE Testbench
-- Module Name:   dsp_core_tb.vhd
-- Project Name:  DSP
-- VHDL Test Bench Created by ISE for module: DSP_CORE
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.DEFINITIONS.all;

ENTITY dsp_core_tb IS
END dsp_core_tb;

ARCHITECTURE behavior OF dsp_core_tb IS

-- Component Declaration
COMPONENT DSP_CORE is
port( CLK      : in STD_LOGIC;
         RES       : in STD_LOGIC;
         MISO      : in STD_LOGIC;
         MOSI      : out STD_LOGIC;
         ADC_CLK   : out STD_LOGIC;
         DAC_CLK   : out STD_LOGIC;
         SAM_CLK   : out STD_LOGIC;
         CS        : out STD_LOGIC;
         SYNC      : out STD_LOGIC);
end COMPONENT;

   --Inputs
   signal CLK : std_logic := '0';
   signal RES : std_logic := '1';
   signal MISO : std_logic := '1';

    --Outputs
   signal ADC_CLK : std_logic;
   signal DAC_CLK : std_logic;
   signal SAM_CLK : std_logic;
   signal CS : std_logic;
   signal SYNC : std_logic;
   signal MOSI : std_logic;

   -- Clock period definitions
   constant CLK_period : time := 20 ns;

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: DSP_CORE PORT MAP (
          CLK => CLK,
          RES => RES,
```

```
          MISO => MISO,
          MOSI => MOSI,
          ADC_CLK => ADC_CLK,
          DAC_CLK => DAC_CLK,
          SAM_CLK => SAM_CLK,
          CS => CS,
          SYNC => SYNC
        );

   -- Clock process definitions
   CLK_process :process
   begin
      CLK <= '0';
      wait for CLK_period/2;
      CLK <= '1';
      wait for CLK_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      wait for 100 ns;
      RES <= '0';
      MISO <= '1';

      wait for 15 us;
      MISO <= '0';

      wait;
   end process;
END;
```

```
--------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Design Name:   IO_CTRL Testbench
-- Module Name:   io_ctrl_tb.vhd
-- Project Name:  DSP
-- VHDL Test Bench Created by ISE for module: IO_CTRL
--------------------------------------------------------------------------------
library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.NUMERIC_STD.ALL; use
IEEE.STD_LOGIC_UNSIGNED.ALL;  library work; use work.DEFINITIONS.all;

ENTITY io_ctrl_tb IS
END io_ctrl_tb;

ARCHITECTURE behavior OF io_ctrl_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT IO_CTRL
    PORT(
         CLK : IN  std_logic;
         RES : IN  std_logic;
         MISO : IN  std_logic;
         Y : IN  std_logic_vector(11 downto 0);
         OUT_DONE : IN  std_logic;
         ADC_CLK : OUT  std_logic;
         DAC_CLK : OUT  std_logic;
         SAM_CLK : OUT  std_logic;
         CS : OUT  std_logic;
         SYNC : OUT  std_logic;
         MOSI : OUT  std_logic;
         X_SHIFT : OUT  vect_16x30;
         Y_SHIFT : OUT  vect_16x30;
         X : OUT  std_logic_vector(11 downto 0);
         IN_DONE : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal CLK : std_logic := '0';
   signal RES : std_logic := '1';
   signal MISO : std_logic := '1';
   signal Y : std_logic_vector(11 downto 0) := "110011001100";
   signal OUT_DONE : std_logic := '1';

    --Outputs
   signal ADC_CLK : std_logic;
   signal DAC_CLK : std_logic;
   signal SAM_CLK : std_logic;
   signal CS : std_logic;
   signal SYNC : std_logic;
   signal MOSI : std_logic;
   signal X_SHIFT : vect_16x30;
   signal Y_SHIFT : vect_16x30;
   signal X : std_logic_vector(11 downto 0);
   signal IN_DONE : std_logic;

   -- Clock period definitions
   constant CLK_period : time := 20 ns;

BEGIN
```

```
   -- Instantiate the Unit Under Test (UUT)
   uut: IO_CTRL PORT MAP (
          CLK => CLK,
          RES => RES,
          MISO => MISO,
          Y => Y,
          OUT_DONE => OUT_DONE,
          ADC_CLK => ADC_CLK,
          DAC_CLK => DAC_CLK,
          SAM_CLK => SAM_CLK,
          CS => CS,
          SYNC => SYNC,
          MOSI => MOSI,
          X_SHIFT => X_SHIFT,
          Y_SHIFT => Y_SHIFT,
          X => X,
          IN_DONE => IN_DONE
        );

   -- Clock process definitions
   CLK_process :process
   begin
      CLK <= '0';
      wait for CLK_period/2;
      CLK <= '1';
      wait for CLK_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      wait for 100 ns;
      out_done <= '0';
      -- hold reset state for 100 ns.
      wait for 300 ns;
      RES <= '0';
      out_done <= '0';

      wait for 40 ns;
      out_done <= '1';

      wait for CLK_period*10;

      -- insert stimulus here

      wait;
   end process;

END;
```

```
--------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Design Name:   SPI_ADC_CTRL Testbench
-- Module Name:   spi_adc_ctrl_tb.vhd
-- Project Name:  DSP
-- VHDL Test Bench Created by ISE for module: SPI_ADC_CTRL
--------------------------------------------------------------------------------
library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.NUMERIC_STD.ALL; use
IEEE.STD_LOGIC_UNSIGNED.ALL;  library work; use work.DEFINITIONS.all;

ENTITY spi_adc_ctrl_tb IS
END spi_adc_ctrl_tb;

ARCHITECTURE behavior OF spi_adc_ctrl_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT SPI_ADC_CTRL
    PORT(
         CLK : IN  std_logic;
         RES : IN  std_logic;
         RD : IN  std_logic;
         MISO : IN  std_logic;
         ADC_CLK : OUT  std_logic;
         CS : OUT  std_logic;
         X : OUT  std_logic_vector(11 downto 0);
         DONE : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal CLK : std_logic := '0';
   signal RES : std_logic := '0';
   signal RD : std_logic := '1';
   signal MISO : std_logic := '1';

    --Outputs
   signal ADC_CLK : std_logic;
   signal CS : std_logic;
   signal X : std_logic_vector(11 downto 0);
   signal DONE : std_logic;

   -- Clock period definitions
   constant CLK_period : time := 10 ns;

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: SPI_ADC_CTRL PORT MAP (
          CLK => CLK,
          RES => RES,
          RD => RD,
          MISO => MISO,
          ADC_CLK => ADC_CLK,
          CS => CS,
          X => X,
          DONE => DONE
        );

   -- Clock process definitions
```

```
   CLK_process :process
   begin
      CLK <= '0';
      wait for CLK_period/2;
      CLK <= '1';
      wait for CLK_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 700 ns;
      MISO <= '0';

      -- insert stimulus here

      wait;
   end process;

END;
```

```
--------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Design Name:   SPI_DAC_CTRL Testbench
-- Module Name:   spi_dac_ctrl_tb.vhd
-- Project Name:  DSP
-- VHDL Test Bench Created by ISE for module: SPI_DAC_CTRL
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY spi_dac_ctrl_tb IS
END spi_dac_ctrl_tb;

ARCHITECTURE behavior OF spi_dac_ctrl_tb IS

    COMPONENT SPI_DAC_CTRL
    PORT(
        CLK : IN  STD_LOGIC;
        RES : IN  STD_LOGIC;
        WT : IN  STD_LOGIC;
        Y : IN  STD_LOGIC_VECTOR(11 downto 0);
        DAC_CLK : OUT STD_LOGIC;
        SYNC : OUT STD_LOGIC;
        MOSI : OUT STD_LOGIC
        );
   END COMPONENT;


   --Inputs
   signal tb_clk  : std_logic := '0';
   signal tb_res  : std_logic := '1';
   signal tb_wt   : std_logic := '1';
   signal tb_y     : std_logic_vector(15 downto 0) := (others => '0');

   --Outputs
   signal tb_dac_clk : std_logic;
   signal tb_sync    : std_logic;
   signal tb_mosi    : std_logic;

   -- Nexys2 Clock Period is 20 ns (50 MHz)
   constant CLK_period : time := 20 ns;


BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: SPI_DAC_CTRL PORT MAP (
          tb_clk,
          tb_res,
          tb_wt,
          tb_y,
          tb_dac_clk,
          tb_sync,
          tb_mosi
        );

    -- Clock process definitions
   CLK_process :process
   begin
```

```
      tb_clk <= '0';
      wait for CLK_period/2;
      tb_clk <= '1';
      wait for CLK_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
      tb_res := '0';
      tb_rd := '1';

      l1 : loop
        if rising_edge(tb_dac_clk) then
           tb_miso <= not(tb_miso);
        end if;
      end loop;


      wait;
   end process;

END;
```

```
--------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Kevin Brewer
--
-- Design Name:   CLK_DIV Testbench
-- Module Name:   clk_div_tb.vhd
-- Project Name:  DSP
-- VHDL Test Bench Created by ISE for module: CLK_DIV
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY clk_div_tb IS
END clk_div_tb;

ARCHITECTURE behavior OF clk_div_tb IS

    -- signals
   signal tb_clk : std_logic;
    signal tb_div : integer := 2;
   signal tb_clk_out : std_logic;

    COMPONENT CLK_DIV
    PORT(
         CLK : IN  std_logic;
         DIV : IN  integer;
         CLK_OUT : OUT  std_logic
        );
    END COMPONENT;

   -- Nexys2 Clock Period is 20 ns (50 MHz)
   constant CLK_period : time := 20 ns;


BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: CLK_DIV PORT MAP (
          tb_clk,
          tb_div,
          tb_clk_out
        );

   -- Clock process definitions
   CLK_process :process
   begin
      tb_clk <= '0';
      wait for CLK_period/2;
      tb_clk <= '1';
      wait for CLK_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 300 ns;
      tb_div <= 4;

      wait for 300 ns;
      tb_div <= 10;
```

```
      wait for 300 ns;
      tb_div <= 15;

      wait;
   end process;

END;
```