An Overview of Binary Arithmetic Architectures & Their Implementation in DSP Systems

By Joseph Waddell

Table of Contents

Figures, Tables, & Equations

Introduction

Many branches of the electrical engineering industry involve applications that use digital signal processing. Almost any type of signal that comes in analog form, such as sound, video, and radio or microwaves, must use digital signal processing for implementation in electrical devices. Digital signal processors (DSPs) are devices designed specifically for use in these kinds of applications and provide fast and efficient calculations needed for digital signal processing.

DSPs possess many important characteristics that make them ideal for digital signal processing, which involves rapid, repetitive calculations, making speed one of the most essential of these characteristics. DSPs come in a wide variety of speeds for a multitude of applications. The binary arithmetic architectures DSPs employ to multiply and add during calculations play a large role in determining the speeds at which they operate because faster binary arithmetic calculations leads to faster DSP operation. Like many instances of hardware engineering, balancing arithmetic component speed demands a tradeoff between component size and power consumption. Making a multiplier or adder faster requires more hardware, requiring more power and more physical space.

Students at Cal Poly have almost no resources that investigate the relationship between binary arithmetic component architectures and DSP performance. The DSP Starter Kit (DSK) development platform from Texas Instruments, one of the few hardware outlets on campus to examine this relationship, grants a very limited glimpse of this correlation. Testing the extremities of this device shows how reaching limits, such as maximum filter length, can adversely affect a desired output signal. However, the DSK does not allow for the analysis of its arithmetic architecture, nor does it accommodate viewing and changing arithmetic components for performance comparisons. Other resources include Xilinx ISE Design Suite and MathWorks MATLAB. However, these software tools only simulate hardware performance and students would benefit from a hands-on hardware example.

Project Goals

This project aims to provide a resource for Cal Poly students to explore the relationship between binary arithmetic component architecture and DSP performance. This report outlines the design of a modular DSP system, created with VHDL and implemented on the Digilent Nexys2 development board as a digital filter. The DSP system has many modifiable capabilities, including options to change the sampling rate, to change multiplier and adder combinations, to select multiple filter realization structures in either serial or parallel implementation, and to use scaled integer coefficients for filter calculations. Students may use the system to compare the hardware and filter performance limitations imposed by different types of binary adder and multiplier architectures while operating in real time on the Nexys2.

This report also intends to provide a comparison of the binary arithmetic architectures used to create the DSP building block. Component comparisons include speed, size, and power consumption, as well as observations of limitations on filter length and sampling rate. Component analysis includes detailed block diagrams, functional descriptions, and timing and power simulation analysis.

All hardware used throughout the project is easily and cheaply attainable by any Cal Poly EE student. EE students acquire most of the hardware components used for the design though Cal Poly's CPE/EE 129, 229, and 329 series of classes. Appendix A contains project planning information including budget, Gantt chart, and sustainability analysis of the project.

Design Requirements

This section of the report contains necessary requirements for the final design. Functional requirements describe observable processes the design must exhibit. Performance specifications include various specifications that must be met in the final design.

Functional Requirements

It is important the DSP system under design is easily modifiable, facilitated through a VHDL package of constants that can be changed for desired results. The constants contained in the package allow changing filter structure, serial or parallel calculation implementations, the number of bits used for integer scaling, adder type, multiplier type, and filter coefficients.

All binary arithmetic components must handle a necessary bit size for data calculations. All adders must accommodate 32-bit integers and each multiplier must handle up to 16-bit integers. The arithmetic components of the filter include four adder architectures and two multiplier architectures, in addition to the dedicated Spartan 3E MULT18X18 multiplier, to handle DSP calculations. Data communication between the Digilent ADCPModAD1 analog to digital converter (ADC) and the DIgilent DACPModDA2 digital to analog converter (DAC) and the Nexys2 requires the design of a serial peripheral interface (SPI) control module. All sampling takes place in real time, requiring the design of a sampling rate control module and shift register to handle sample timing and sample delay during filter operation.

FIR and IIR filter realization will take place in two modules, one for normal direct form I, and another for multi-stage cascade direct form II, both capable of serial and parallel calculation implementations. Once implemented in the Nexys2, the filter will accept either an electrical or audio signal and deliver the appropriate output signal, expressing results on either an oscilloscope for electrical inputs, or a stereo speaker for audio inputs.

Performance Specifications

Verifying operation requires the construction of some obligatory infinite-impulse response (IIR) and finite-impulse response (FIR) filters. Specifically, filter realization includes normal direct form I for FIR and small order IIR filters, and cascade direct form II for more complex multi-stage IIRs, each in both serial calculation and parallel calculation implementations.

Testing arithmetic components will involve using simulation results and real time filter operation to verify maximum sustainable sampling rates for a length 4 moving average filter and maximum filter lengths achievable for a sampling rate of 44.1 kHz. Binary arithmetic components handle filter calculations only. Nexys2 dedicated hardware will perform all other calculations, including any incrementation or counting, indexing, or summation of partial products.

Design Specifications

Using a structural VHDL description for the system design promotes code organization and readability. It also allows for easier functional verification of each module before integration with the larger system. Prior to coding, module design planning included preparation of numerous schematics and diagrams used for reference, including flowcharts, block diagrams, logic circuit diagrams, state transition diagrams, and circuit schematics. This section presents design specifications for each module, explains their purpose and operation, and provides simulation and timing results.

System Design

This section contains design specifications for major system modules used throughout the project. This includes an arithmetic component generation module, the SPI control interface module, the sampling rate control module, and two modules used for filter realization structures.

Figure 1 shows a blackbox diagram of the DSP system under design. Inputs to the Nexys2 consist of either the signal generator (SG) or an audio signal provided by a stereo sound source from a computer, iPod, or cell phone. Monitoring the filtered output signal will incorporate an oscilloscope or a stereo speaker connected with the Digilent PmodCON4 RCA audio jack, for the SG and Audio inputs respectively. Programming the Nexys2 will require a USB connected PC and Digilent's Adept Software. Appendix B summarizes all hardware and software information used for design, testing, and implementation of the system.

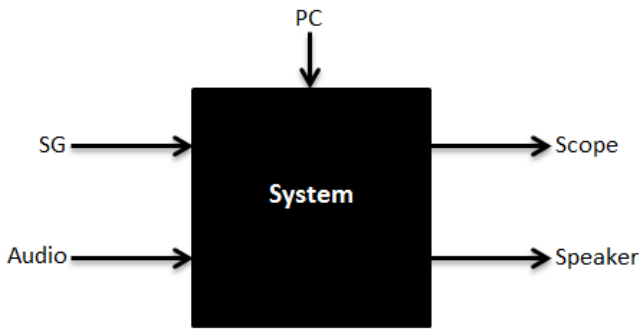


Figure 1. System Blackbox Diagram

Operation

Figure 2 shows a block diagram of the system design. As seen in the figure, the sampling rate control module regulates the SPI control module, taking samples from the ADC at the desired sampling frequency and passing filtered signals to the DAC for output. Two shift registers store and delay inputs and outputs, passing their values to the filter realization module for processing. Adders and multipliers generated by the arithmetic component generator module perform the necessary DSP calculations, and the filtered output is stored in the output shift register for the DAC.

For audio input, a level shifter applies a DC offset to the incoming audio signal to correct it for valid ADC input voltage levels. A DC blocking capacitor removes the offset, bringing the output signal to appropriate voltage levels for the stereo speaker.

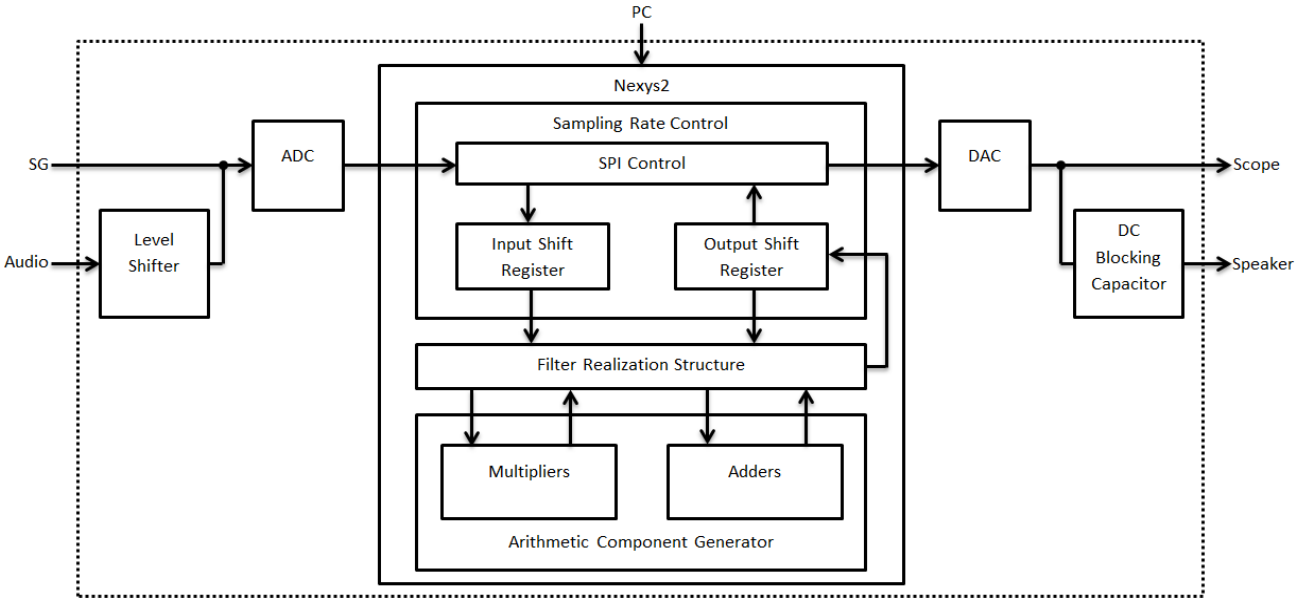


Figure 2. System Block Diagram

The behavior of the system relies heavily on the user defined constants contained in the *CTRL_CONSTANTS.vhd* package. Setting these constants allows for changing the filter realization structure, choosing serial calculation or parallel calculation configurations, choosing the sampling rate, selecting which multipliers and adders to use for DSP calculations, selecting bit sizes for scaled filter coefficients, and defining digital filters for implementation. Table 1 lists all the constants contained in the *CTRL_CONSTANTS.vhd* package and a short description of their function.

Table 1. *CTRL_CONSTANTS.vhd* Description of Constants

	Constant	Function
General	STRUCTURE	Chooses normal direct form I or cascade direct form II filter realization structure
	SorP	Chooses serial calculation or parallel calculation configuration
	SAMP_DIV	Selects sampling rate
	SCALE	Chooses bit size for scaling filter coefficients
	MULTIPLIER	Selects desired multiplier architecture
	ADDER	Selects desired adder architecture
Normal Direct Form I Filters	F_LENGTH	Length of filter specified filter
	N	Number of Ak terms
	M	Number of Bk terms
	Ak	Ak filter coefficients
	Bk	Bk filter coefficients
Cascade Direct Form II Filters	F_STAGES	Number of filter stages
	Aki	Ak filter coefficients
	Bki_S	Bk filter coefficients for serial calculations
	C	Input coefficient
	Bki_P	Bk filter coefficients for parallel calculations

DSP_BB.vhd, *Arithmetic Component Generator Design*

The arithmetic component generator module generates and organizes the binary arithmetic components used for filter calculations. All filter calculations take place using the hardware it generates. Based on the values contained in the *CTRL_CONSTANTS.vhd* package, *DSP_BB.vhd* performs the following functions:

- Generate user specified adders and multipliers used for filter calculations
- Configure arithmetic components for either serial or parallel calculation implementation
- Configure arithmetic components for direct form I or cascade direct form II filter realization
- Convert integer filter coefficients to signed vector equivalents
- Provide products and sums for desired values

Operation

During Xilinx compilation, *DSP_BB.vhd* generates and organizes the desired binary arithmetic components depending on the constants contained in the *CTRL_CONSTANTS.vhd* package. During filter operation all adder and multiplier inputs pass to this module for calculation, returning their respective sums and products. It also takes the specified integer filter coefficients and converts them to 16-bit vectors for use with arithmetic components.

CONVERTER_CTRL.vhd, *SPI Control Design*

Both the ADC and DAC require an SPI for data communication with the Nexys2. The SPI control module *CONVERTER_CTRL.vhd*, fulfills this role. Data communication with each converter requires a different clock frequency, provided by the clock divider module, *CLK_DIV.vhd*. Two of these modules produce a 25 MHz signal for DAC timing and a 12.5 MHz signal for ADC timing from the 50 MHz Nexys2 system clock signal.

The SPI control module contains two separate finite state machines (FSMs). The first controls data transfer from the ADC, state transition diagram shown in Figure 3, and the second controls data sent to the DAC, state transition diagram shown in Figure 4.

Operation

For each desired sample, the assertion of RD_EN causes the FSM to change to the SET_ADC state, beginning ADC data transfer sequence by setting the ADC chip select signal, CS, high. On the next rising edge of the ADC clock the RUN_ADC state brings CS low, beginning serial data transfer from the ADC and asserting RD_CNT_EN to begin timing the data transfer. Each ADC clock period increments RD_CNT until it reaches a value of 15. This signals the completion of ADC data transfer and the state changes to SET_SAMP, where the assertion of the GOT_SAMP flag notifying that ADC data transfer is complete. The FSM then returns to the ADC_IDLE state to await the next sample.

DAC operation executes similarly. After the first sample, the assertion of WRT_EN sends the FSM to the SET_DAC state, where the assertion of the DAC chip select signal, SYNC, begins the DAC data transfer sequence. After bringing SYNC back low in the RUN_DAC state during the next rising edge of the DAC clock, serial data transfer to the DAC starts. When the counter WRT_CNT reaches 15, DAC data transfer concludes and the state changes back to SET_DAC to immediately begin another DAC output sequence.

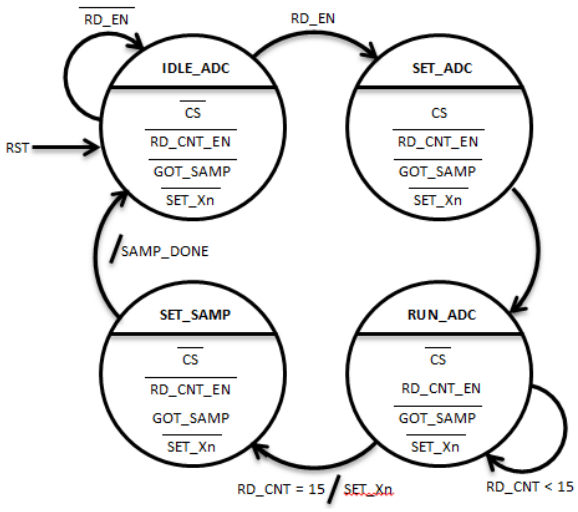


Figure 3. ***CONVERTER_CTRL.vhd*** **ADC State Transition Diagram**

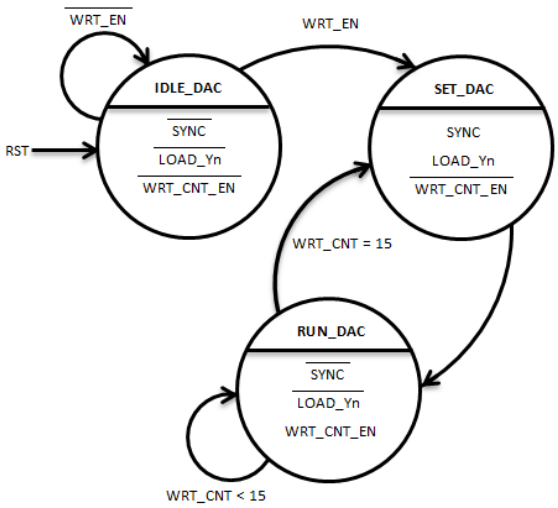


Figure 4. ***CONVERTER_CTRL.vhd*** **ADC State Transition Diagram**

SAMPLE_CTRL.vhd, *Sampling Timing Control Design*

The sampling timing control module controls timing for ADC input and begins DAC output of filtered signals. It also handles storing and delaying input samples and filtered output signals through the single-in parallel-out (SIPO) shift register module *SIPO_SHR.vhd*. The module contains one small FSM, shown in Figure 5, which determines timing for the RD_EN and WRT_EN signals that determine ADC and DAC operation.

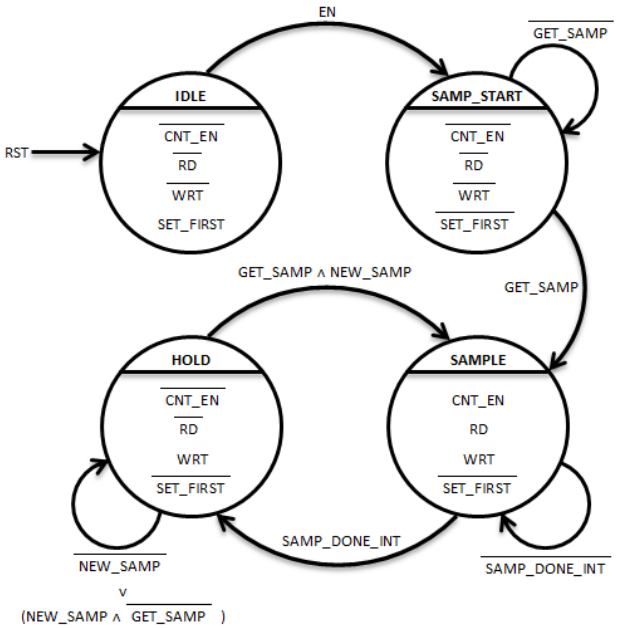


Figure 5. ***SAMPLE_CTRL.vhd*** **State Transition Diagram**

Operation

The internal clock signal SAMP_CLK, again generated by the *CLK_DIV.vhd* module, determines the sampling rate of the system. First, assertion of the system enable, EN, causes a state change to the SAMP_START state, where the FSM waits for the appropriate time to take the first sample. A separate, small process triggers the assertion of the GET_SAMP flag for each required sample period, causing the FSM to change to the SAMPLE state, where the assertion of RD_EN starts the ADC data transfer sequence. The assertion of SAMP_DONE_INT signals the completion of sample acquisition and causes another state change. Once in the HOLD state, the assertion of WRT_EN begins output through the DAC and the FSM returns to the SAMPLE state to wait for the next sample interval.

Filter Structure Design

Testing the operation of the entire system led to the design of two filter realization structures, *NORMAL.vhd* to implement normal direct form I, and *CASCADE.vhd* to implement cascade direct form II. This section contains specifications and diagrams used for their design and VHDL implementation.

NORMAL.vhd, Normal Direct Form I Filter Realization

The *NORMAL.vhd* module performs required operations to execute DSP calculations using direct form I filter realization. Figure 6 displays a diagram showing direct form I realization and Eq. 1 shows the difference equation used for its VHDL implementation.

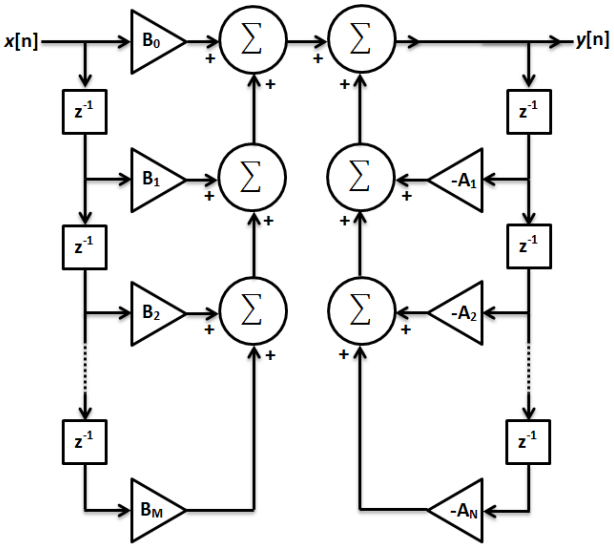


Figure 6. Normal Direct Form I Filter Realization

Eq. 1. Direct Form I Difference Equation

$$y[n] = B_0x[n] + B_1x[n-1] + B_2x[n-2] + \cdots + B_Mx[n-M]$$
$$-A_1y[n-1] - A_2y[n-2] - \cdots - A_Ny[n-N]$$

Operation

Figure 7 shows a state transition diagram for the FSM that composes the *NORMAL.vhd* module. It determines appropriate signals to send to the arithmetic components to perform necessary DSP calculations. State changes for both serial and parallel calculation implementations are shown in the figure. Dummy states are used to create delays so filter operations may complete before moving on to the next state.

Beginning in the IDLE state, the FSM waits for the EN system enable signal, causing a state change to the WAIT4Xn state to wait for the next input sample. Assertion of the SAMP_DONE flag in the converter control module signals the acquisition of a new sample, where the state changes to MULT to begin filter calculations. The module's following behavior is determined by calculation implementation type.

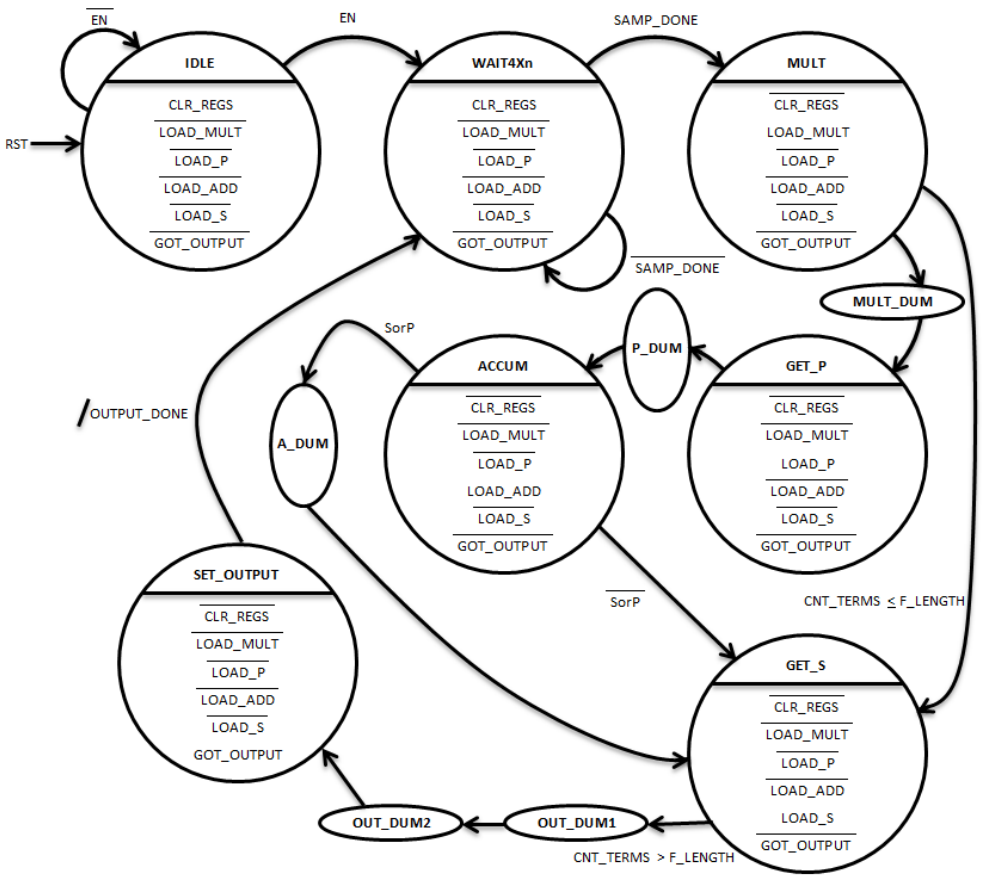


Figure 7. *NORMAL.vhd* State Transition Diagram

Serial Calculation Implementation

Figure 8 shows a block diagram for the *NORMAL.vhd* module configured for serial calculation implementation. First, multiplier inputs are loaded in the MULT state with values to calculate one filter term. Next, in the GET_P state, the multiplier's output gets stored into the product register for accumulation. In the ACCUM state, adder inputs are loaded with the current filter term from the product register and the current value of accumulated filter terms, stored in the sum register

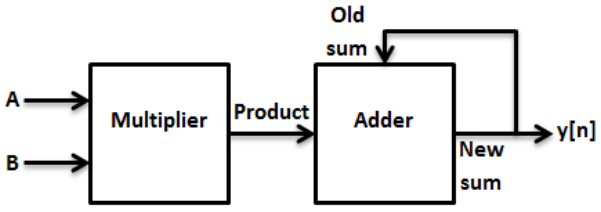


Figure 8. *NORMAL.vhd* Serial Calculation Implementation Block Diagram

In the GET_S state, the FSM checks if filter calculations are complete for the current sample. If not, the FSM returns to the MULT state, where a new filter term is multiplied and then accumulated as before. If complete, GET_S assigns the output, the state changes to SET_OUTPUT causing the assertion of the GOT_OUTPUT flag. The FSM then returns to the WAIT4Xn state to wait for the next input sample.

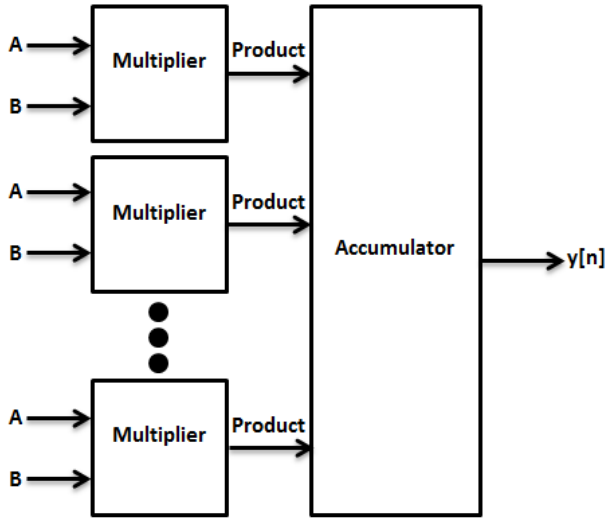


Figure 9. Parallel Calculation Implementation

Parallel Calculation Implementation

Figure 9 shows a diagram for the using *NORMAL.vhd* configured for parallel calculation implementation. In the MULT state values are assigned to the inputs of each multiplier. The product of each multiplier is loaded into the product register in the GET_P state and corresponds to one filter term. After changing to state ACCUM, the accumulator inputs are loaded with the products of each multiplier from the product register. The output value is assigned next in the GET_S. Finally, state SET_OUTPUT asserts GOT_OUTPUT before returning to the WAIT4Xn state to wait for the next sample.

CASCADE.vhd, Cascade Direct Form II Realization

The *CASCADE.vhd* module performs necessary operations for DSP calculations using a cascaded, multi-stage filter approach with direct form II realization. The diagram in Figure 10 contains a filter realization of 2 filter stages arranged for serial calculations and Eq. 2 shows the difference equations describing each stage of the filter used for VHDL design and implementation.

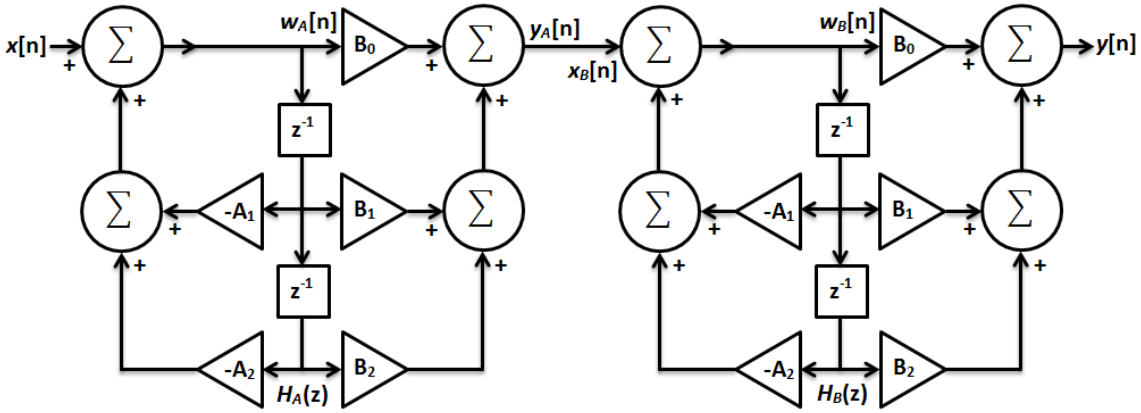


Figure 10. Cascade Direct Form II Filter Realization for Serial Calculations

Eq. 2. Direct Form II Difference Equations for Each 2nd Order Filter Stage for Serial Calculations

$$w_i = x_i[n] - A_{1i}w_i[n-1] - A_{2i}w_i[n-2]$$

$$y_i = B_{0i}w_i[n] + B_{1i}w_i[n-1] + B_{2i}w_i[n-2]$$

$$x_i[n] = y_{i-1}[n] \;\; \& \;\; x_1[n] = x[n]$$

As shown by the difference equations in Eq. 2, calculating the output y_i requires the preceding calculation of w_i for each stage of the filter. The output of each stage passes to the next as its input. The two stages cascaded in Figure 10 form a fourth order IIR filter.

Calculation parallelization for cascaded filter stages improves with a slightly different realization, shown in Figure 11. As with serial calculations, the calculation for w_i precedes the calculation for y_i. However, this realization allows for the simultaneous accumulation of outputs from each 2nd order filter stage, a characteristic not possible with the serial realization structure.

Eq. 3. Direct Form II Difference Equations for Each 2nd Order Filter Stage for Parallel Calculations

$$w_i = x_i[n] - A_{1i}w_i[n-1] - A_{2i}w_i[n-2]$$

$$y_i = B_{0i}w_i[n] + B_{1i}w_i[n-1]$$

$$y[n] = Cx[n] + y_A[n] + y_B[n]$$

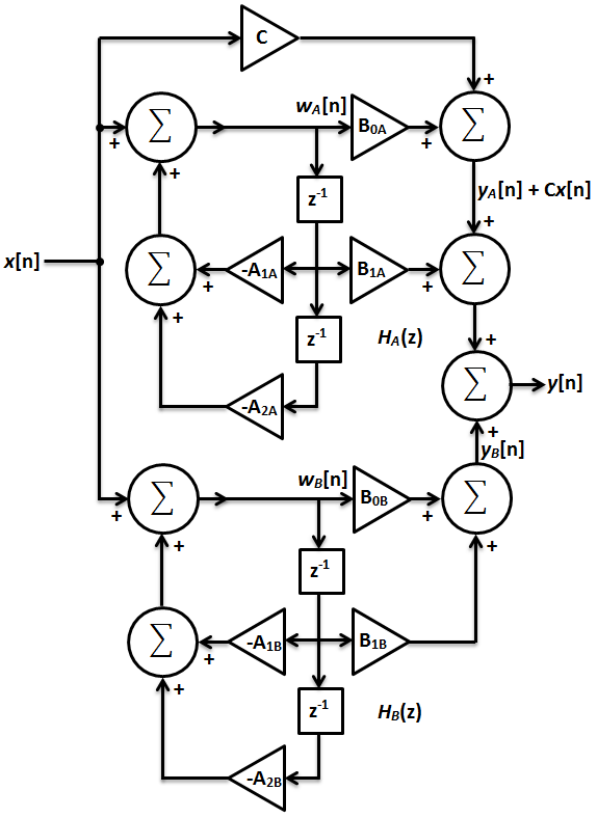


Figure 11. Cascade Direct Form II Filter Realization for Parallel Calculations

Operation

Figure 12 contains the FSM that *CASCADE.vhd* uses to perform DSP operations. Table 2 gives further details about state changes in the figure. Because of the differences in structure between serial and parallel implementations, this module's FSM becomes more complex than the one used for *NORMAL.vhd.*

Once again, the FSM starts in the IDLE state. After assertion of the EN signal, the state changes to WAIT4Xn to wait for a new input sample. The assertion of SAMP_DONE signals the acquisition of a new sample and the state changes to MULT to begin multiplication of filter terms.

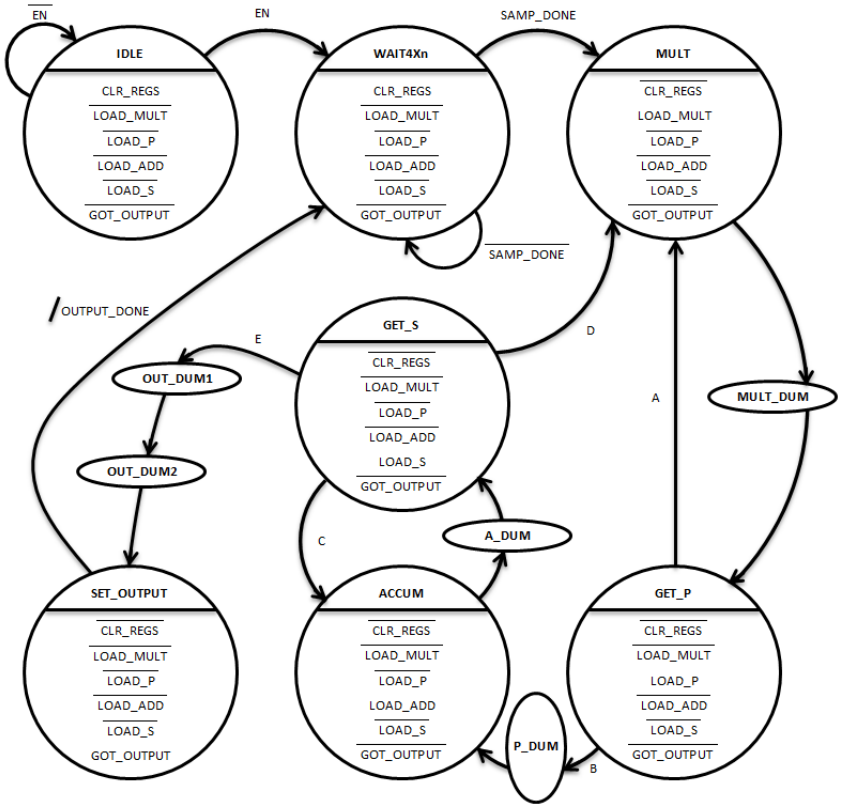


Figure 12. *CASCADE.vhd* State Transition Diagram

Transition	Condition
A	$\overline{SorP} \wedge (CNT\_TERMS \neq 2) \wedge (CNT\_TERMS \neq 5)$
B	$\overline{SorP} \wedge [(CNT\_TERMS = 2) \vee (CNT\_TERMS = 5)]$ $\vee$ $SorP$
C	$\overline{SorP} \wedge \overline{ACCUM\_TOG} \wedge (CNT\_TERMS = 2)$ $\vee$ $\overline{SorP} \wedge \overline{ACCUM\_TOG} \wedge (CNT\_TERMS = 5)$ $\vee$ $SorP \wedge \overline{ACCUM\_TOG} \wedge \overline{ACCUM\_OUTPUT}$
D	$\overline{SorP} \wedge ACCUM\_TOG \wedge (CNT\_TERMS = 2)$ $\vee$ $\overline{SorP} \wedge ACCUM\_TOG \wedge (CNT\_TERMS = 5) \wedge (CNT\_STAGE < F\_STAGES - 1)$ $\vee$ $SorP \wedge ACCUM\_TOG \wedge \overline{ACCUM\_OUTPUT}$
E	$\overline{SorP} \wedge ACCUM\_TOG \wedge (CNT\_TERMS = 5) \wedge (CNT\_STAGE = F\_STAGES - 1)$ $\vee$ $SorP \wedge ACCUM\_OUTPUT$

Table 2. *CASCADE.vhd* State Transition Logic

Serial Calculation Implementation

Figure 13 shows a block diagram summarizing FSM behavior while organized to perform serial calculations. Multiplier inputs are loaded with values to multiply the first term of w_i in state MULT. Next, the product is loaded into the product register in the GET_P state. Multiplication of the next term of w_i occurs immediately after, loading multiplier inputs in state MULT and placing the resulting product in the product register in state GET_P. The two products get loaded into the adder in state ACCUM before changing to state GET_S to load the sum register with the result. The final term of w_i, the current stage input, is accumulated immediately after in the same manner.

Calculating stage output, y_i, occurs similarly. Serial multiplications take place with the sequential execution of the MULT and GET_P states, calculating the 3 terms of y_i one by one and storing results into the product register. The ACCUM and GET_S states then accumulate the 3 terms before either starting another output calculation sequence for the next filter stage or finalizing filter output and asserting the OUTPUT_DONE flag in the SET_OUTPUT state.

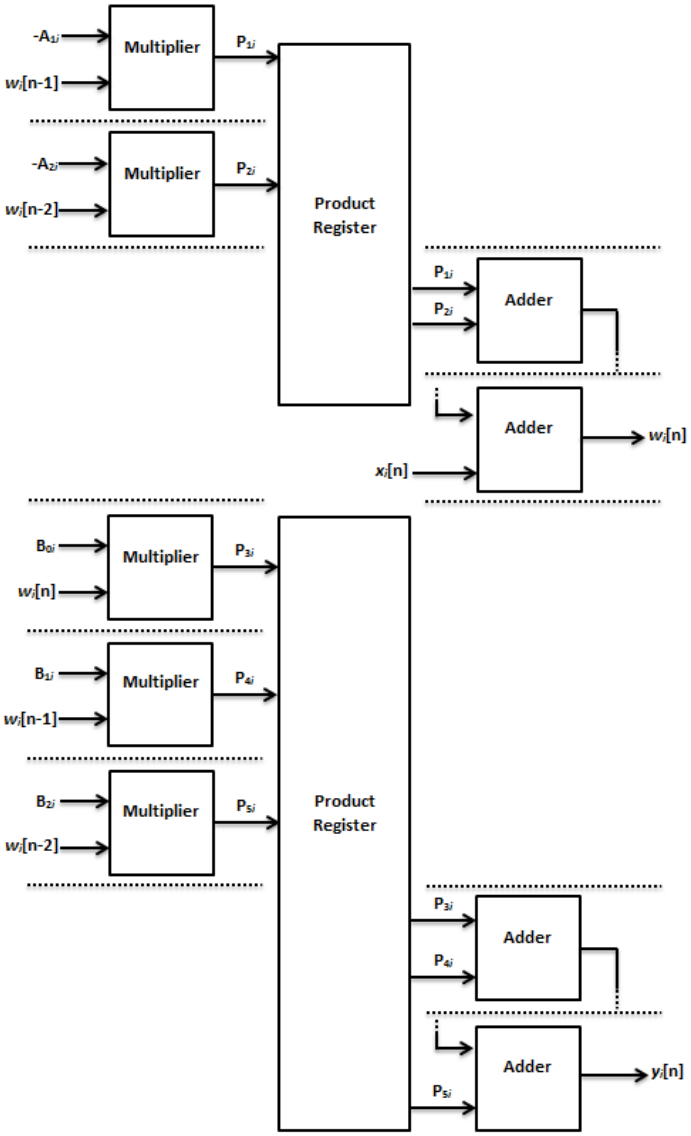
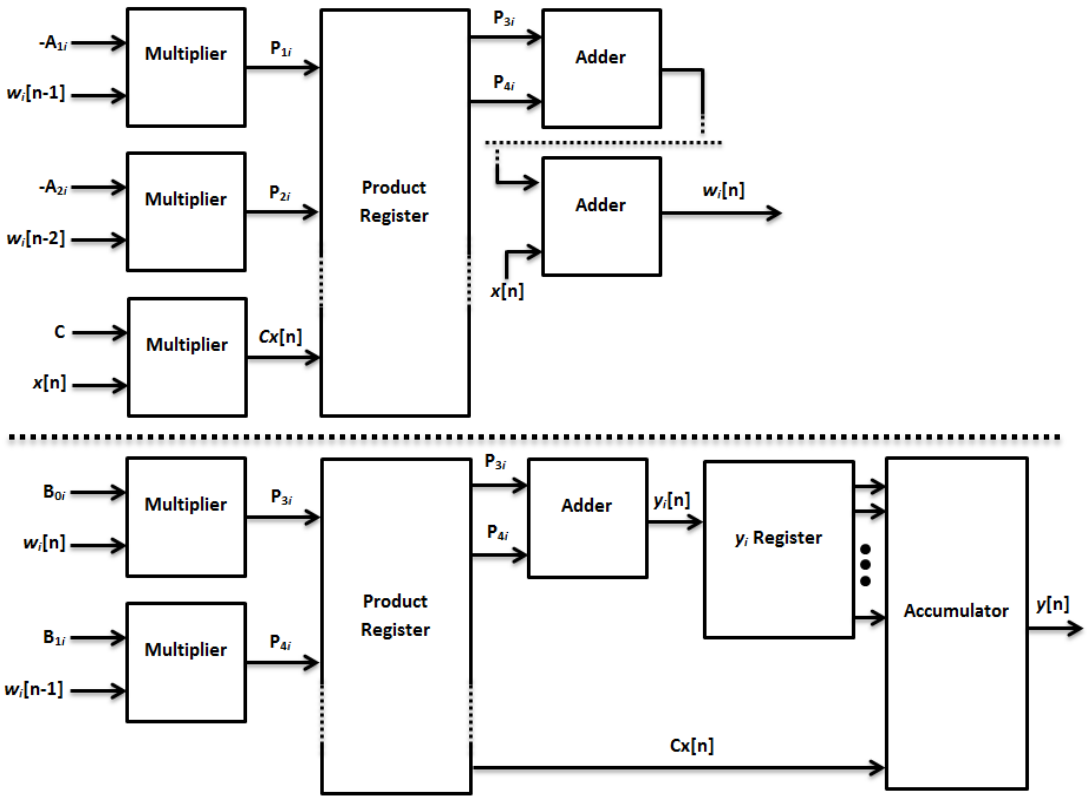


Figure 13. *CASCADE.vhd* Serial Calculation Implementation Block Diagram

Parallel Calculation Implementation

Figure 14 displays a diagram for *CASCADE.vhd* while performing parallel calculations. Three multipliers are generated to handle multiplications and one adder accumulates terms for w_i and y_i. The CSA accumulator takes care of accumulating the stage outputs, y_i.

To first calculate w_i, in the MULT state two multipliers receive inputs to calculate both A_i terms and the third gets the constant C and the current input to calculate *Cx*[n] for later use. These products get loaded into the product register in state GET_P before passing them to the adder in state ACCUM. The adder's sum is then placed in the sum register in state GET_S before returning to state ACCUM to add the current input to produce w_i. The adder output, w_i, then gets stored into the w_i register in the GET_S state.

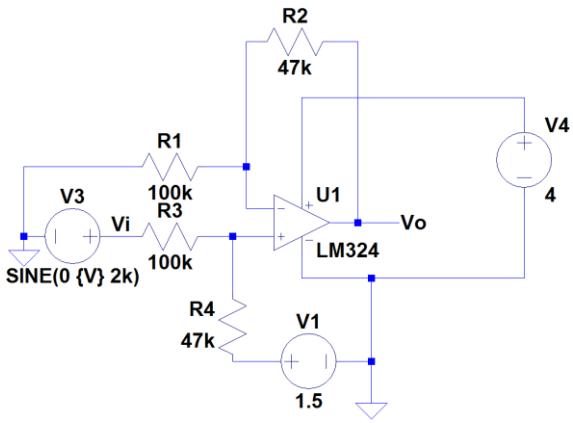


Figre 14. *CASCADE.vhd* Parallel Calculation Implementation Block Diagram

Next, the state returns to MULT to begin calculating filter stage output y_i in a similar fashion. Two multipliers are loaded with input values to calculate the two terms that make up y_i and their products are loaded into the product register in state GET_P. These two products are then loaded into the adder in state ACCUM and the sum is stored in the y_i register in state GET_S. Finally, the state changes back to ACCUM to load the accumulator with values stored in the y_i register to calculate the final output value. The output y[n] is then set in state GET_S before finally asserting the OUTPUT_DONE flag in the SET_OUTPUT state.

Audio Interface Design

The Digilent PmodCON4 RCA audio jack and ADC interface audio input signals to the Nexys2. The device that provides the audio signal, like a computer or stereo system, balances it around 0 V, making it incompatible with ADC input voltage levels. A level shifter circuit corrects this by adding a DC offset to the input signal. The level shifter circuit connects between the RCA audio jack and ADC as seen in the system block diagram in Figure 1. Figure 15 shows a schematic of the level shifter circuit developed in LTpice.



The level shifter uses an LM324 OP amp and is designed to have a gain of about 1. An additional DC power supply provides a DC offset of 1.5 V. This should limit the input voltage to the ADC to around its 3 V maximum.

Figure 15. Level Shifter Circuit LTspice Schematic

Figure 16 shows a simulation of the level shifter circuit using LTspice. For this simulation, the input voltage is swept from 1 V to 5 V. The circuit outputs a replica of the input, close to unity gain, until it reaches 3 V in amplitude. The output clips at 0 V and 3 V as shown in the results, exercising just below the maximum range of the ADC.

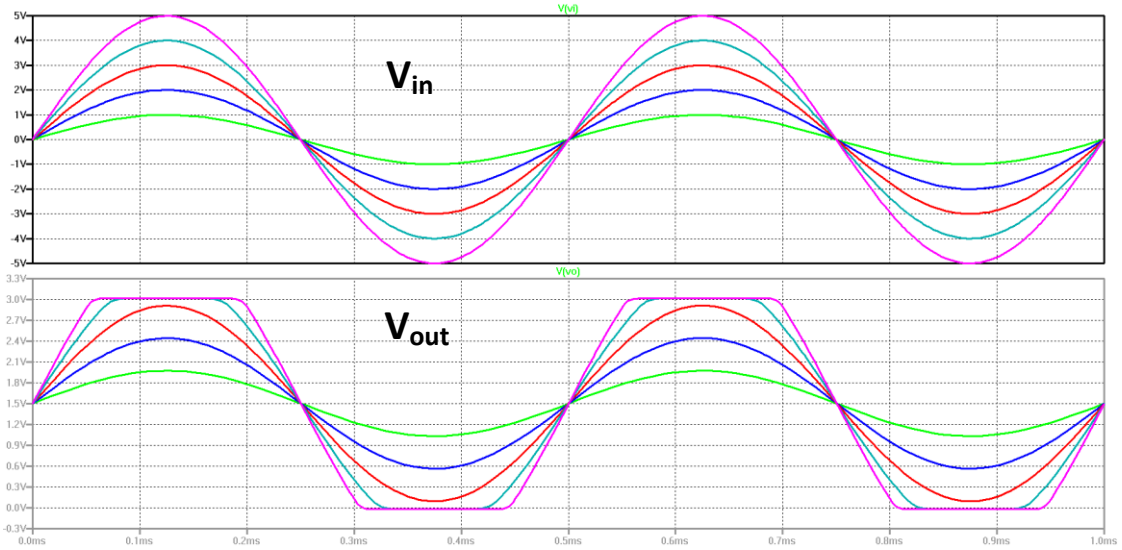


Figure 16. Level Shifter Circuit LTspice Simulation Results

The offset added by the level shifter must be removed before sending the signal to the speakers for listening. A DC blocking capacitor on the DAC output resolves this issue. Its capacitance of 10 µF insures that all range of audible frequencies pass through it without impedance.

Adder Design

The three adders selected for implementation include the Ripple-Carry, Carry-Lookahead, Carry-Select adder architectures. Each provides a unique example for contrast, varying from the others in terms of speed, power consumption, and required FPGA logic. The design of a fourth adder using carry-save architecture, the Carry-Save accumulator, allowed for more efficient accumulation during parallel calculation implementations. Figure 17 and Figure 18 show blackbox diagrams for the adder and accumulator modules respectively.

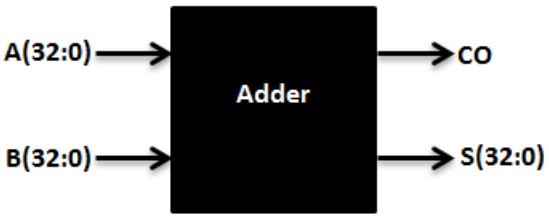


Figure 17. Adder Blackbox Diagram

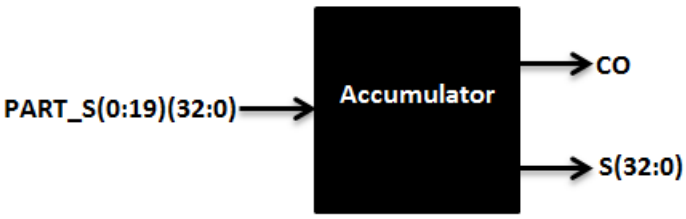


Figure 18. Accumulator Blackbox Diagram

An overflow circuit accompanies the adder and accumulator modules. It detects for overflow in the sum and adjusts it accordingly. Figure 19 below shows a flowchart for the OVERFLOW.vhd module.

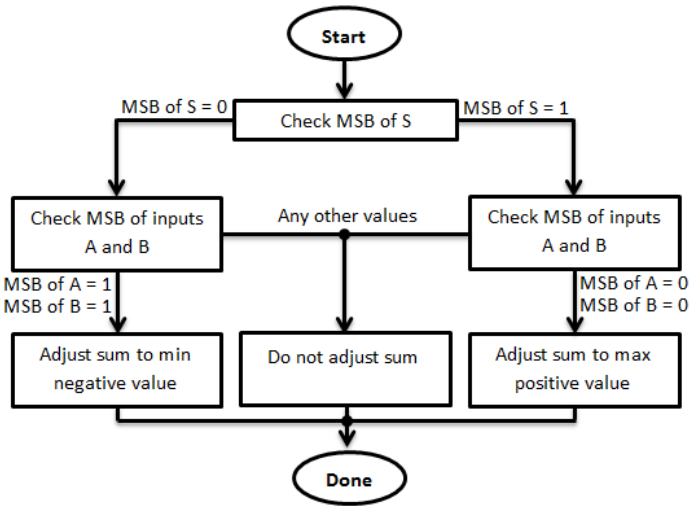


Figure 19. *OVERFLOW.vhd* Flowchart

RC_ADDER_32BIT.vhd, Ripple-Carry Adder Design

Figure 20 shows the Ripple-Carry (RC) adder design schematic used to create the VHDL module. It calculates the 32-bit sum and 1-bit carry-out, S and CO respectively, based on the 32-bit input signals, A and B. VHDL implementation included three modules consisting of the full adder, half adder, and overflow circuit using only concurrent signal assignments.

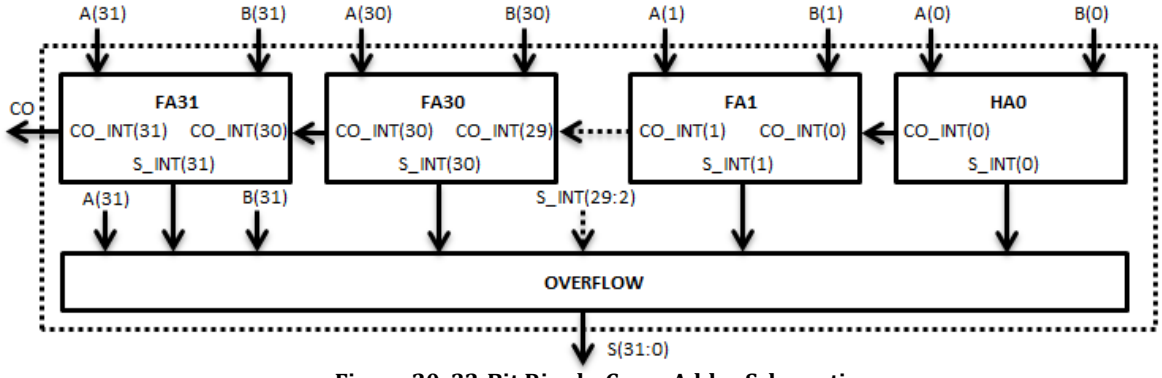


Figure 20. 32-Bit Ripple-Carry Adder Schematic

Thirty full adders, labeled as FA1 to FA31, and one half adder, labeled as HA0, compose the entire ripple-carry architecture. Figure 21 and Figure 22 display logic diagrams for the full adder and half adder modules, respectively, accompanied by the design equations used for VHDL implementation shown respectively in Eq. 4 and Eq. 5.

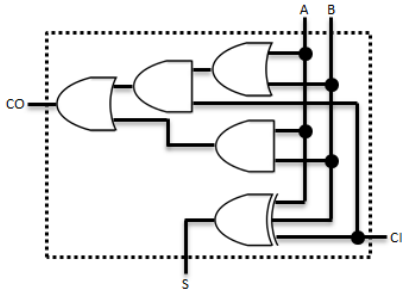


Figure 21. Full Adder Logic Diagram

Eq. 4. Full Adder Design Equations

$$CO = [(A + B) \cdot CI] + (A \cdot B)$$

$$S = A \oplus B \oplus CI$$

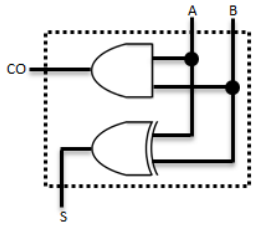


Figure 22. Half Adder Logic Diagram

Eq. 5. Half Adder Design Equations

$$CO = A \cdot B$$

$$S = A \oplus B$$

Operation

The basic design of the ripple-carry adder makes it compact and power efficient but relatively slow at calculations. Each adder in the RC architecture calculates its 1-bit sum and 1-bit carry-out signals from respective 1-bit signals from A, B, and a carry-in. Every full adder in the carry chain passes its carry to the next. This means that each adder must wait for the carry-out bit from the previous adder. The final calculation of the sum and carry-out results after the carry bit propagates through every full adder in the carry chain.

Gate Delay Analysis

The ripple-carry adder architecture makes it the slowest adder used in the project but also the most compact. The 32-bit adder is composed of one half adder and thirty one full adders. From examining Figure 20, the critical delay path through the RC adder is from carry-in to carry-out. The logic diagram for the full adder shows that 2 gate delays are introduced by carry propagation. For a rough estimation, if 2 gate delays are introduced per adder then there are about 64 gate delays for the entire RC adder to calculate a sum.

CLa_ADDER_32BIT.vhd, Carry-Lookahead Adder Design

The Carry-Lookahead (CLA) adder, schematic shown in Figure 23, improves upon the propagation delay of the RC adder. Like the RC adder, it takes two 32-bit binary numbers, A and B, and produces their 32-bit sum, S, and carry-out bit, CO.

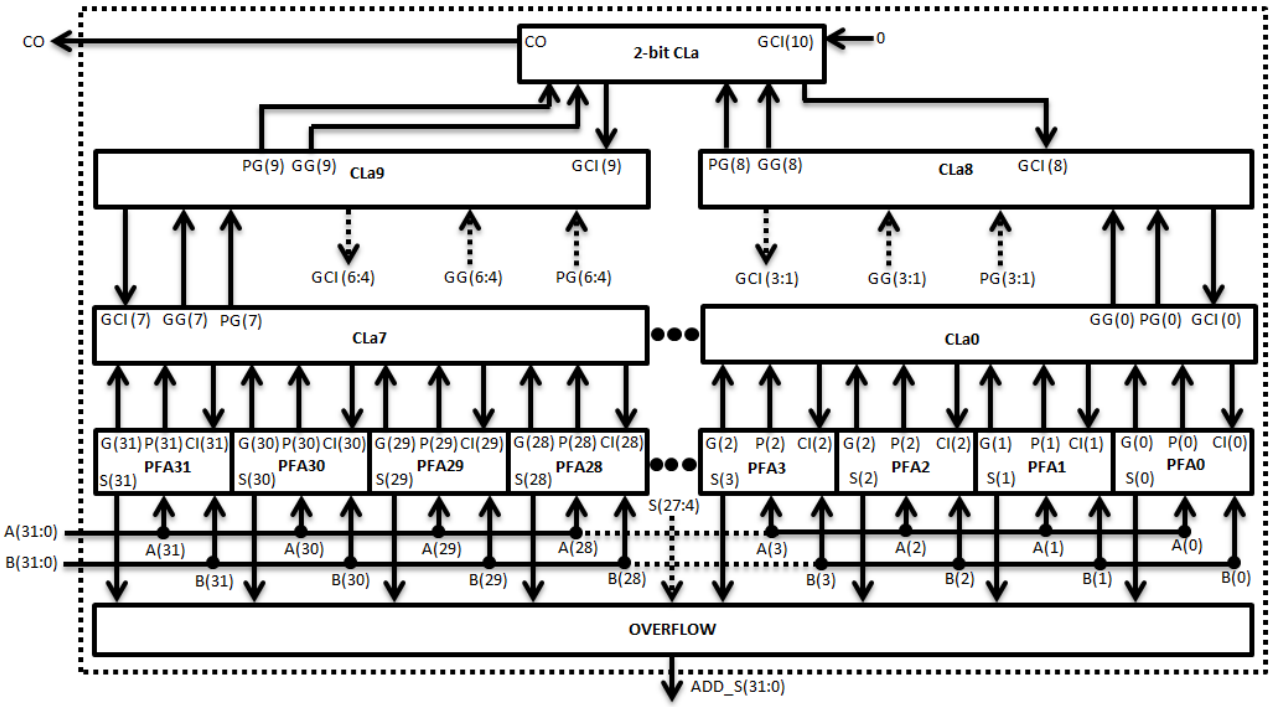


Figure 23. Carry-Lookahead Adder Schematic

Besides the overflow module, VHDL implementation required two modules, the partial full adder and 4-bit CLA unit, using strictly concurrent signal assignments and combinational logic. A total of thirty two partial full adders and ten CLA units, labeled respectively as PFA0 to PFA31 and CLa0 to CLa9 in Figure 23, completed the adder architecture. Figure 24 shows a logic circuit diagram for the partial full adder module, accompanied by the design equations used for VHDL implantation shown in Eq. 6.

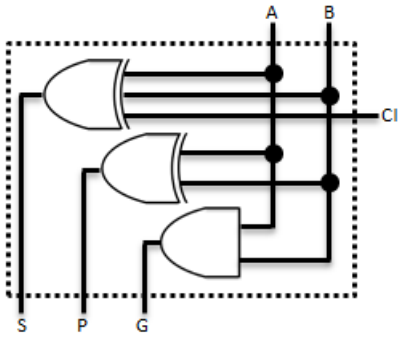


Figure 24. Partial Full Adder Logic Diagram

Eq. 6. Partial Full Adder Design Equations

$$G = A \cdot B$$

$$P = A \oplus B$$

$$S = A \oplus B \oplus CI$$

To calculate the total adder sum, the CLA adder uses CLA logic to interpret the generate and propagate signals produced by the partial full adders to create their carry-in bits. While enabling the adder to produce a sum faster, CLA logic demands a considerable amount of hardware. The logic circuit diagram shown in Figure 25 represents the 4-bit CLA logic module.

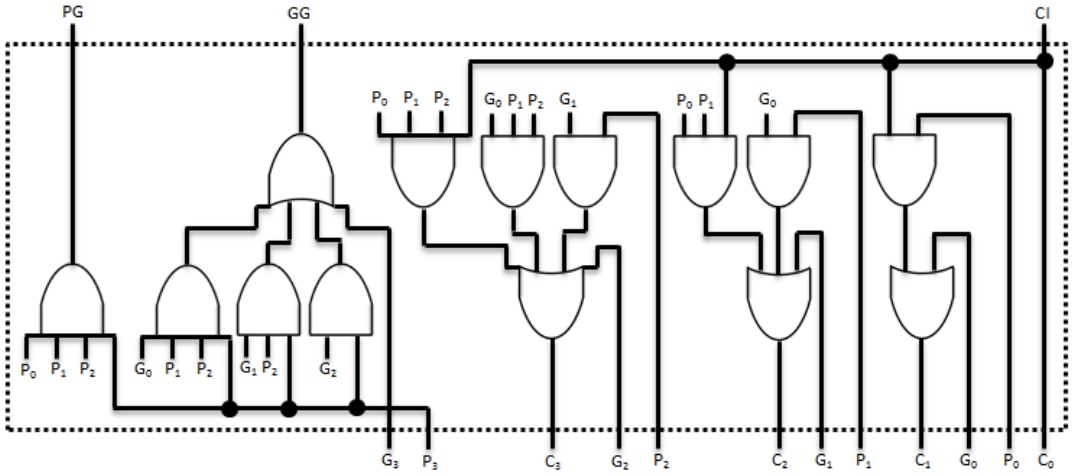


Figure 25. 4-Bit Carry-Lookahead Logic Diagram

For this particular adder design, CLA units were limited to handle a maximum of 4-bits. Designing them to handle larger bit sizes is possible but requires more hardware. The design equations used for the VHDL implementation of the 4-bit CLA module are listed below in Eq. 7.

Eq. 7. 4-Bit CLA Unit Design Equations

$$C_0 = CI$$

$$C_1 = (CI \cdot P_0) + G_0$$

$$C_2 = (CI \cdot P_0 \cdot P_1) + (G_0 \cdot P_1) + G_1$$

$$C_3 = (CI \cdot P_0 \cdot P_1 \cdot P_2) + (G_0 \cdot P_1 \cdot P_2) + (G_1 \cdot P_2) + G_2$$

$$GG = (G_0 \cdot P_1 \cdot P_2 \cdot P_3) + (G_1 \cdot P_2 \cdot P_3) + (G_2 \cdot P_3) + G_3$$

$$PP = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$

Operation

In addition to the 1-bit sum produced by the full adder, the partial full adder produces 1-bit generate (G) and propagate (P) signals, as seen in Figure 24. The assertion of G indicates that the addition of A and B will always create a carry-out bit regardless of the possibility of producing carry-out bits by adding less significant digits. Similarly, the assertion of P indicates that a carry-in will always propagate through the adder to the left.

Each partial adder accepts 1-bit from signals A and B, and the 1-bit signal CI, generated by the 4-bit CLA units. With these inputs, they produce 1-bit signals S, G, and P. As seen in Figure 23, 4-bit CLA units CLa0 to CLa7 create CI bits for the partial full adders based on the signals G and P, and a carry-in to the unit. The 4-Bit CLA units CLa8 and CLa9 take group generate (GG) and group propagate (GP) signals, produced by CLA units CLa0 to CLa7, to create group carry-in (GCI) signals for the other CLA units included in the architecture. The highest level CLA unit of the adder, shown in Figure 23 as a 2-bit CLA, produces 2 GCI bits for CLA logic units CLa8 and CLa9, along with the carry-out bit of the adder.

Gate Delay Analysis

The critical delay path for the CLA adder is from carry-in to carry-out. Carries propagate through each 4-bit CLA unit in at most 2 gate delays and it takes another gate delay for each group of 4 partial full adders to calculate their 1-bit sum. There are 10 4-bit CLA units the carry must propagate through, contributing 20 gate delays. Each group of 4 partial full adders adds 1 gate delay and there are 8 of these groups, contributing 8 more gate delays. This gives the total gate delay of the CLA adder as about 28 gate delays.

CSe_ADDER_32BIT.vhd, Carry-Select Adder Design

Figure 26 shows the design schematic for the carry-select adder. As with the other adder modules, it produces a 32-bit sum and a carry-out bit, labeled as S and CO respectively, from the 32-bit input signals A and B.

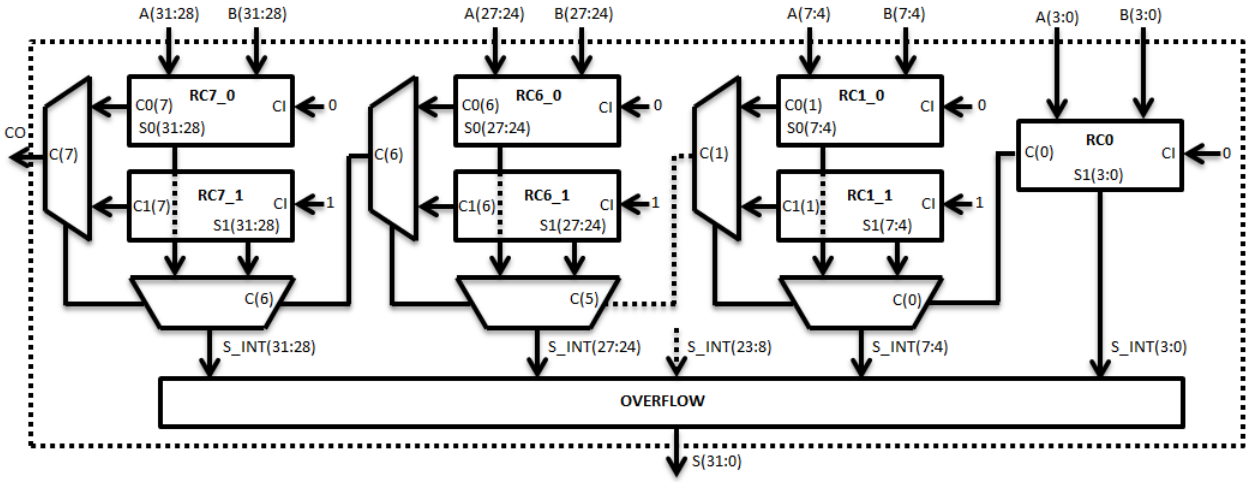


Figure 26. Carry-Select Adder Schematic

A total of fifteen 4-bit RC adder modules compose most of the adder architecture. A pair of these 4-bit adders, and two small 2-to-1 multiplexors, makes up each stage of the adder. Concurrent signal assignments and combinational logic proved sufficient for VHDL implementation.

For this design, stages of the adder were limited to 4-bits. A more time efficient architecture involves designing stages based on MUX propagation delay. This works by making stages that calculate the more significant bits of the sum use larger ripple-carry adders than the preceding stages. Designing each stage so that its propagation delay equals the total propagation delay of all the previous stages means no stage waits idly for the carry signal from the previous stage.

Operation

At its most basic level, the CSE adder consists of two RC adders working in parallel. One RC adder calculates a sum based on a carry-in of 0 and the other for a carry-in of 1, labeled respectively as RC0_0 to RC7_0 and RC1_1 to RC7_1 in Figure 26. Each stage calculates four bits of the sum using 4-bit RC adders. Based on carry signals passed from preceding stages, MUXs select which of these sums to use in the final value and which carry signal passes to the next adder stage.

Gate Delay Analysis

The critical delay path for each 4-bit RC adder is from carry-in to carry-out, about 8 gate delays for the 4 full adders included per stage. Each stage uses a 2-to-1 MUX to propagate carries to the next stage, adding 2 gate delays per MUX for a total of 14 gate delays for the 7 MUXs included in the carry chain. This gives a total of about 22 gate delays to calculate the sum, less than half the delay of the 32-bit RC adder module.

CSa_ACCUM_32BIT.vhd, Carry-Save Accumulator Design

One final adder, the carry-save (CSA) accumulator shown in Figure 27, is designed to execute faster accumulation of filter terms while using parallel filter realization structures. This module takes signal PART_S, a 2-D register of up to twenty 32-bit values, and produces its accumulated total, the 32-bit sum, ADD_S, and carry-out bit, ADD_CO

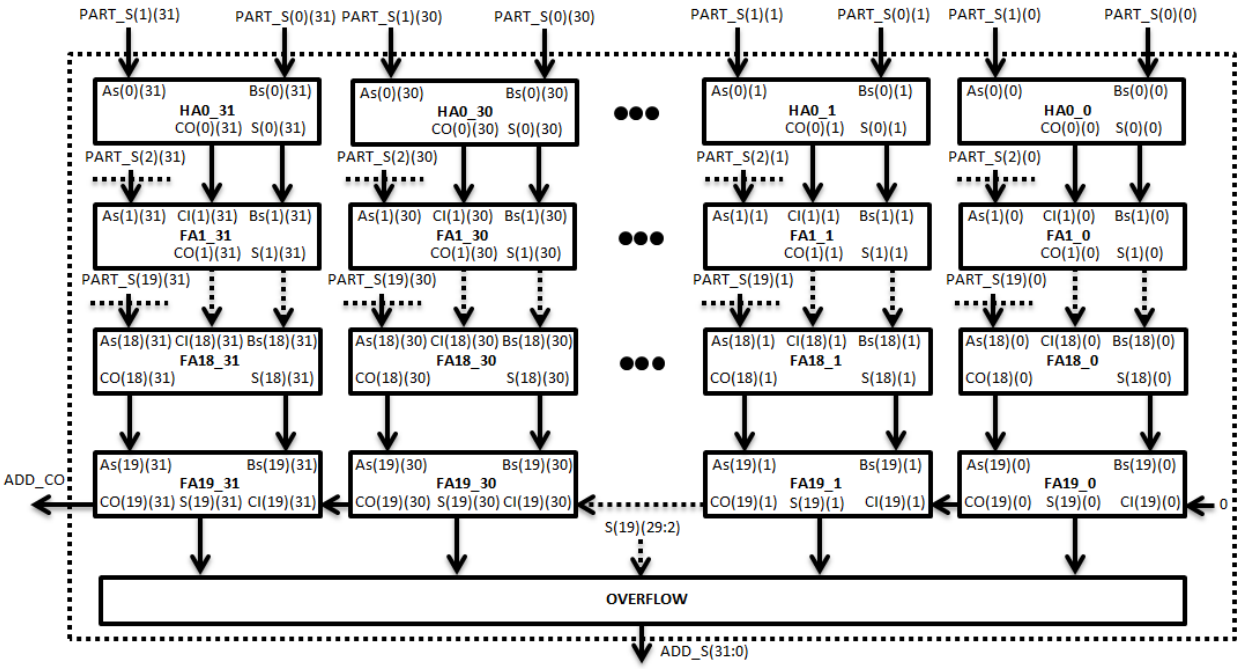


Figure 27. Carry-Save Accumulator Schematic

The CSA accumulator's architecture resembles closely to the shift-add multiplier. Half adders, labeled as HA0_0 to HA0_31 in Figure 27, compose the highest level of the adder hierarchy, while full adders make up the all other levels of the adder. For this architecture, VHDL code using entirely concurrent statements proved sufficient.

Operation

Because of their similarities in architecture, the CSA accumulator also operates similarly to the shift-add multiplier. The sum and carry bits from each adder stage are saved and passed to the next lower level in the adder hierarchy. The carry bit is finally propagated through, in ripple-carry fashion, on the last hierarchical stage to calculate the final value of the sum. This architecture allows for faster accumulation of multiple terms because carries from each summation stage do not propagate through the adder until the last stage.

Gate Delay Analysis

The critical path of the full adders that make all but the last stage of the accumulator is from input to sum, or three gate delays per stage. If the maximum number of terms the accumulator may add together is 20, then about 60 gate delays are introduced for the upper stages of the accumulator. Full adders in the last stage create the carry chain and have critical delay paths from carry-in to carry-out, just as in the RC adder. Thirty two full adders compose the carry chain stage, same as the RC adder, increasing propagation time by another 64 gate delays. This is a total of about 124 gate delays for the accumulator configured to sum together 20 terms. This is much faster than using any other adder to accumulate 20 values.

Multiplier Design

Figure 28 shows a blackbox diagram of the multiplier component. The Digilent Nexys2 development board comes equipped with twenty dedicated MULT18X18 multipliers. In addition, two multiplier designs, a simple Shift-Add multiplier and a Modified Booth multiplier, fulfilled the other multiplier components used for the project.

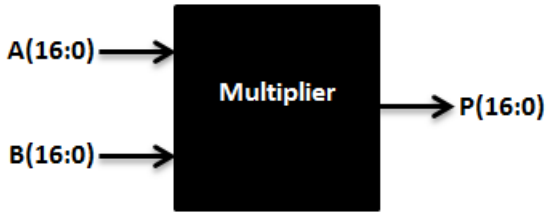


Figure 28.Multiplier Blackbox Diagram

Shift-Add Multiplier

Like the RC adder, the simplistic architecture of the Shift-Add (SA) multiplier makes it compact and easy to design. It applies one of the most basic approaches used for multiplication, the shift-add algorithm. The shift-add multiplier module, shown in Figure 29, calculates the 32-bit product P from the two 16-bit input signals A and B.

Similarly to the CSA adder, half adders, labeled as HA0_0 to HA0_14, compose the top stage of the multiplier hierarchy. All other components used for the multiplier architecture consist of full adders. To implement this architecture in VHDL required only concurrent signal assignments and combinational logic.

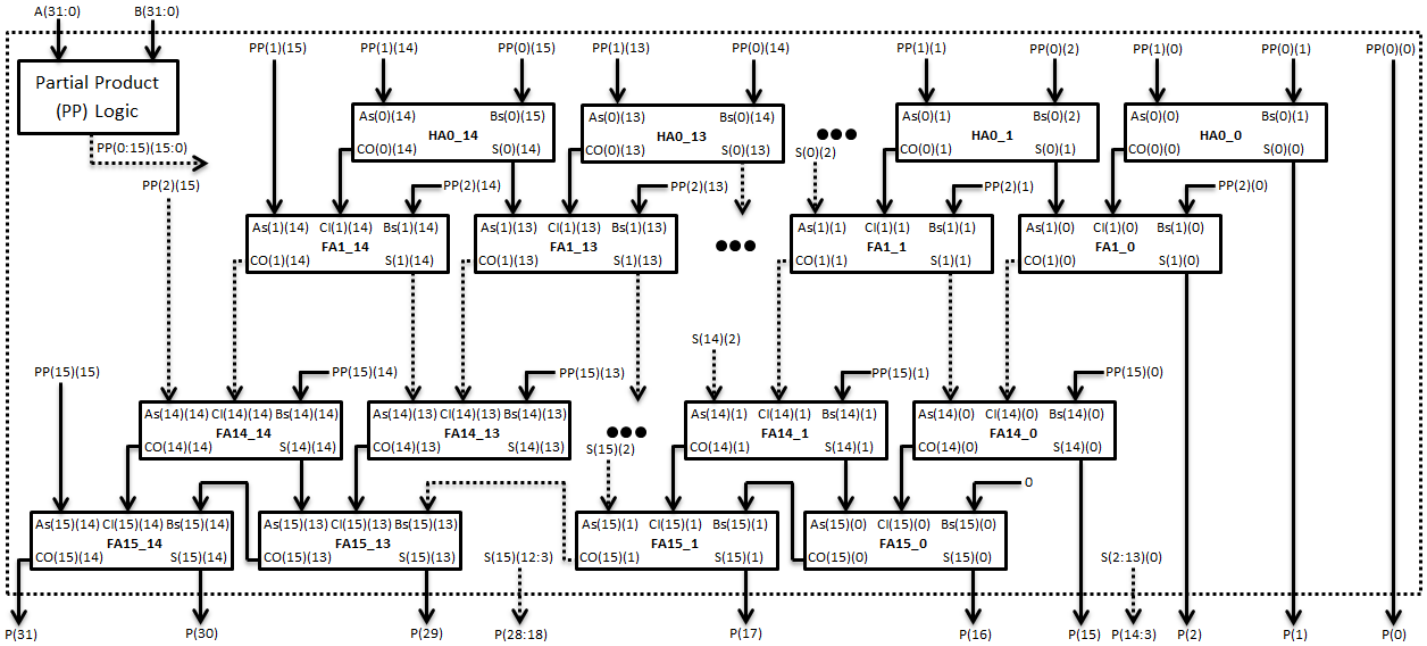


Figure 29. Shift-Add Multiplier Schematic

Operation

The SA multiplier method of multiplication works much like the pencil and paper approach or long multiplication, sometimes called Standard Algorithm. Each bit in the multiplier is multiplied with the multiplicand to produce partial products. These are then shifted one place to the left and accumulated to produce the final product. Figure 30 displays a 6-bit example using the shift-add algorithm.

Multiplicand (41):							1	0	1	0	0	1
Multiplier (29):						x	0	1	1	1	0	1
Partial Product 1							1	0	1	0	0	1
Partial Product 2						0	0	0	0	0	0	
Partial Product 3					1	0	1	0	0	1		
Partial Product 4				1	0	1	0	0	1			
Partial Product 5			1	0	1	0	0	1				
Partial Product 6	+	0	0	0	0	0	0					
Product (1189):	0	1	0	0	1	0	1	0	0	1	0	1

Figure 30. 6-Bit Multiplication Using the Shift-Add Algorithm

The partial product register, PP, contains the partial products determined from the multiplier and multiplicand signals A and B. Like the carry-save accumulator, sums and carries are saved and sent to the next lower tier of adders in the multiplier hierarchy. Finally, as seen in the last tier of the schematic, the carry propagates through in ripple-carry fashion, producing the finished product.

Partial Product Analysis

The SA multiplier is the slowest multiplier used for the project. It is composed primarily of full adders arranged similarly to the CSA accumulator. Because of the similarities in architecture, timing similarities in terms of gate delay also exist.

The SA multiplier does not use complicated algorithms or take advantage of additional logic to lower the number of partial products it uses to calculate its product. Because of this, one partial product is needed for every bit of the multiplicand or multiplier. This means a total of 16 partial products must be accumulated to calculate the 32-bit product.

Modified Booth Multiplier

Figure 31 displays a flowchart for the Modified Booth multiplier. It gets its name from its use of a slightly modified version of Booth's Multiplication Algorithm to calculate the product. The Booth multiplier differs from all other binary arithmetic components used for the project in that VHDL implementation is accomplished using no concurrent statements. One small process, using only combinational logic, makes up the entirety of the VHDL module.

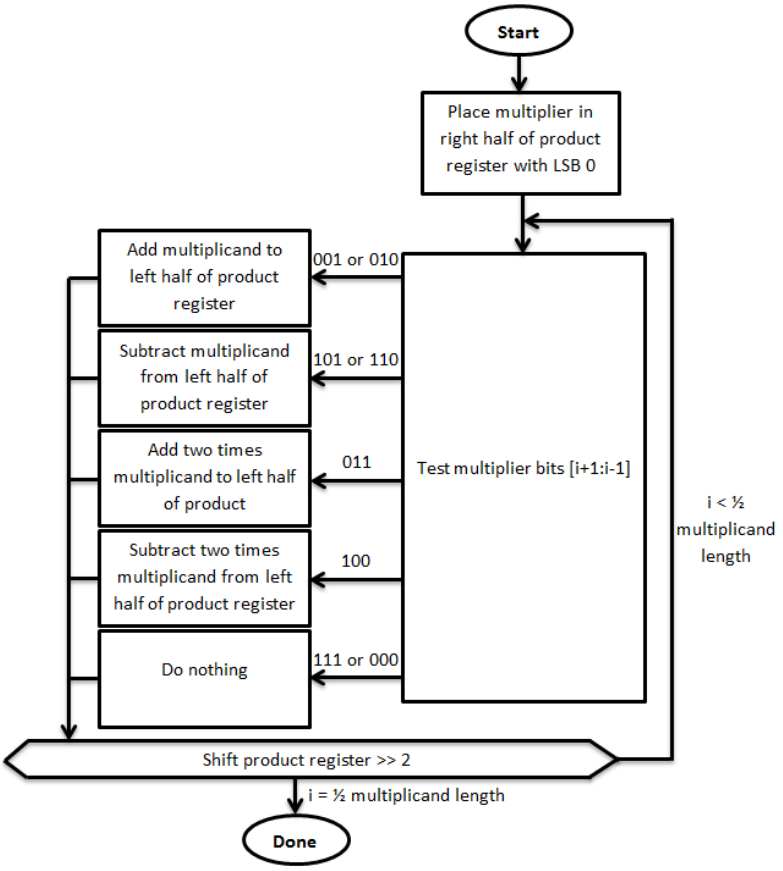


Figure 31. Modified Booth Multiplier Flowchart

Operation

Booth's Multiplication Algorithm improves upon the method of shift-add multiplication by reducing the number of partial product needed to calculate a product. It takes advantage of the fact that any sequence of 1's in a binary number can be broken down into the difference of two binary numbers. For example, the binary value 0b0111 can be broken down into the difference of 0b1000 – 0b0001. The three additions needed for the three partial products required for the value 0b0111, break down into one addition and one subtraction, or two additions. Examining the multiplier 1-bit at a time, plus an extra helper bit on the right, allows for the generation partial products based on the detection of sequences of 1's.

While Booth's Algorithm reduces partial products needed to multiply by sequences of 1's, it actually increases partial products for singleton 1's, which when broken down into the difference of two numbers creates an additional partial product. The modified version of Booth's Algorithm circumvents this issue by looking at the multiplier 2-bits at a time, plus an extra bit on the right, in order to appropriately detect and handle singleton 1's.

Table 3. Generated Partial Products for Modified Booth Algorithm

3-Bit Sequence	Partial Product
000 or 111	None
001 or 010	Multiplicand
101 or 110	-Multiplicand
011	2*Multiplicand
100	-2*Multiplicand

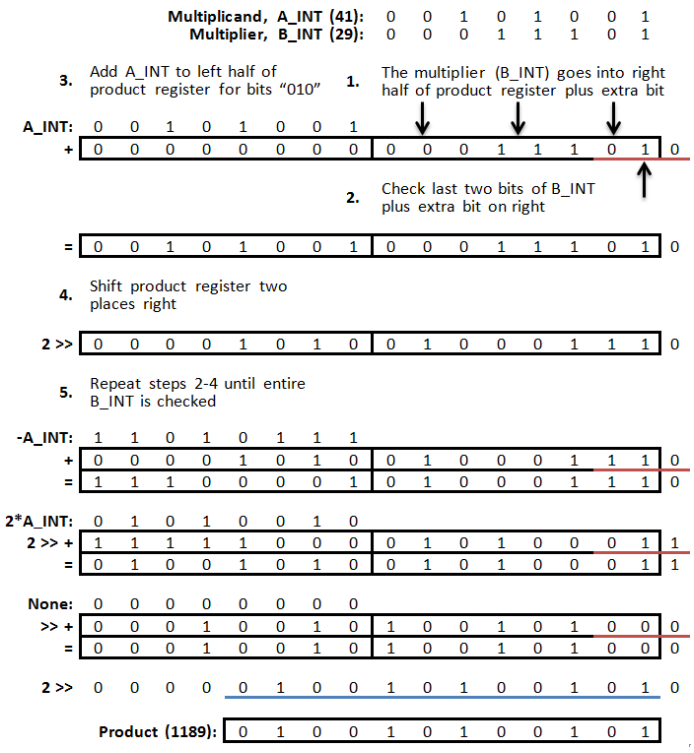


Figure 32. 6-Bit Multiplication Using the Modified Booth's Multiplication Algorithm

Table 3 shows the partial products generated for 3-bit sequences. With the detection of singleton 1's, now only one partial product, the addition of the multiplicand, is generated. The other binary sequences generate partial products of two times the multiplicand, the negative of the multiplicand, or the negative of two times the multiplicand as seen in the table. Figure 32 provides an example of 6-bit multiplication using the Modified Booth's Multiplication Algorithm.

As seen in the example, the multiplier and multiplicand are padded with two 0's, one to account for the generation of negative partial products and another to keep their bit sizes even. After checking the appropriate 2-bits of the multiplier, plus the extra bit on the right, the corresponding partial product is added to the left portion of the product register. The product register is then shifted two places right before the multiplier bits are checked again. These steps iterate a number of times equal to half the length of the two input signals with the addition of the padded 0's, four times in the above example, to determine the product.

Partial Product Analysis

The Booth Multiplier takes advantage of the Modified Booth Multiplication Algorithm to reduce the number of partial products needed to calculate the product. As shown in the example of the algorithms operation, it takes about half the number of partial products to calculate the product as the shift-add multiplier. This implies that the Booth multiplier calculates the product in about half the time of the shift-add multiplier.

MULT18x18 Multiplier Design

The Digilent Nexys2 development board contains hardware with twenty embedded MULT18x18 multipliers and the Xilinx design tools allow for their easy instantiation and use. The Xilinx documentation for the embedded multiplier comes with instantiation templates, inference examples, and detailed instructions for their implementation. Using the provided information, a VHDL module containing one single concurrent statement using the * operator calculates the desired product using the MULT18x18 multiplier. The MULT18x18 documentation does not give detailed information related to the architecture or algorithm used by this multiplier.

VHDL Implementation

This section provides information related to the VHDL implementation of all modules. RTL and Technology schematics generated by the Xilinx tools are discussed for various modules. Behavioral and post PAR simulations for many modules are also displayed and analyzed. Unless otherwise stated, all modules were optimized for speed during synthesis and no constraints were set. All VHDL code can be found in Appendix C and all testbenches used for module verification are contained in Appendix D.

Sample Rate & SPI Control Implementation

Because of their dependence on each other, *SAMPLE_CTRL.vhd* and *CONVERTER_CTRL.vhd* were simulated and tested together. Figure 33 shows a behavioral simulation using the testbench found in Appendix D. These results verify toggling action of the ADC chip select signal CS and the DAC chip select signal SYNC. As shown in the results, CS is brought high on the rising edge of the sampling clock signal SAMP_CLK. The SAMP_DONE flag cues the completion of sample acquisition, toggling SYNC to begin output through the DAC.

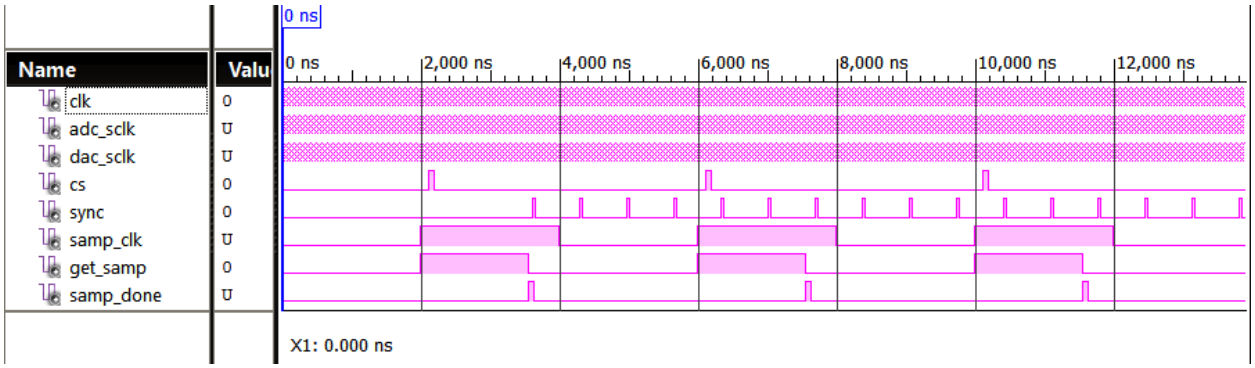


Figure 33. *SAMPLE_CTRL.vhd* Behavioral Simulation – Verification of CS and SYNC

Figure 34 displays a zoomed in version of the same simulation results. This one verifies the amount of time in between CS and SYNC toggles. For proper ADC and DAC operation, at least 16 falling edges of their respective clock signals must pass in between CS and SYNC toggles. As verified in the results, there are many more than 16 ADC_SCLK falling edges in between CS toggles for the ADC and there are exactly 16 DAC_SCLK falling edges between SYNC toggles.

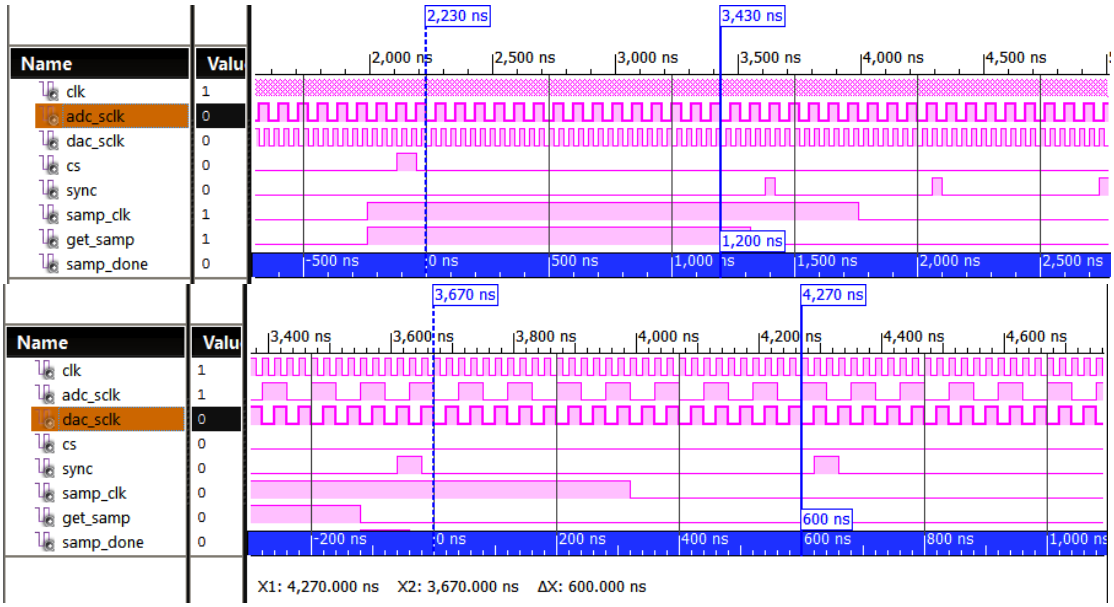


Figure 34. *SAMPLE_CTRL.vhd* Behavioral Simulation – Verification of CS and SYNC Toggling Rates

Adder Implementation

Figure 35 displays a behavioral simulation of the RC adder module. The results display sums from 3 different test cases. Case 1 shows addition of 2 negative numbers, case 3 shows correction by the overflow circuit, and in case 3 most the adders that make up the RC adder architecture generate carries. The adder gives the correct result for each case. Behavioral simulations performed for each adder module produced identical results.

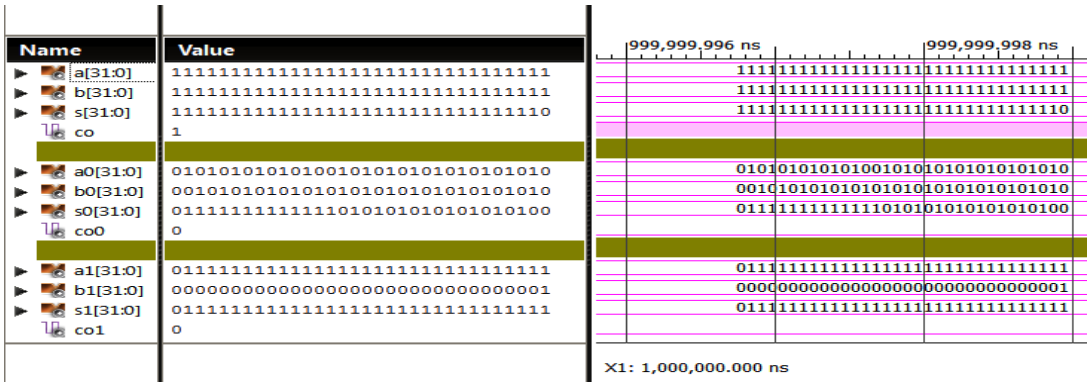


Figure 35. ***RC_ADDER_32BIT.vhd*** **Behavioral Simulation**

Figure 36 shows a behavioral simulation for the CSA accumulator module. This particular test case accumulates 10 values that have all been set to 1 so the sum can be easily checked by inspection. The accumulator calculates the correct sum of 10 as seen in the figure.

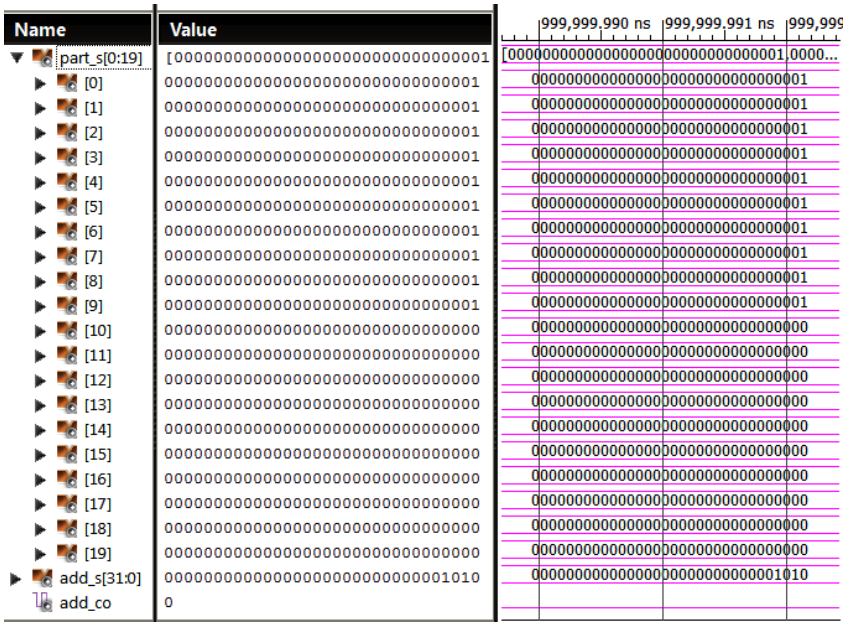


Figure 36. ***CSA_ACCUM_32BIT.vhd*** **Behavioral Simulation**

Figure 37 below displays a post PAR timing simulation for the RC adder without the overflow circuit using maximum values for A and B to generate as many carries as possible. Results show that the RC adder has a propagation delay of 12.292 ns.

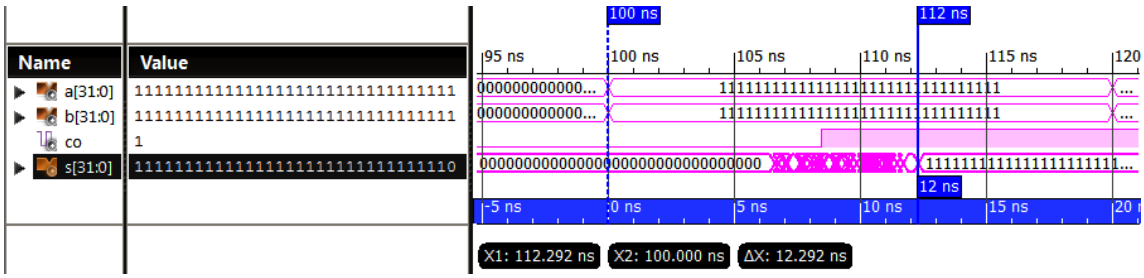


Figure 37. *RC_ADDER_32BIT.vhd* Timing Simulation – No *OVERFLOW.vhd*

Figure 38 shows a timing simulation of the RC adder performed with the overflow circuit and the same values for inputs A and B. The results show an addition in propagation delay of 1.614 ns, yielding a total propagation delay of 13.906 ns for the adder.

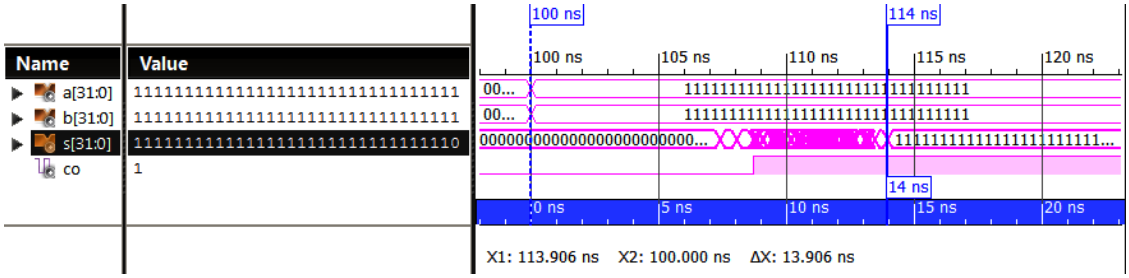


Figure 38. *RC_ADDER_32BIT.vhd* Timing Simulation – With *OVERFLOW.vhd*

Similar results came from post PAR simulations of the other adder modules. Table 4 shows propagation delays for all adder modules with and without the overflow circuit. As expected from gate delay analysis of its architecture, the CSE adder has the fastest propagation delay, shown as 12.593 ns with the overflow correction and 12.593 ns without overflow correction. The propagation delay for the overflow circuit is also the fastest when used with the CSE adder.

Table 4. Summary of Propagation Delays for Adder Modules

Adder	Prop delay with overflow correction (ns)	Prop delay with no overflow correction (ns)	Prop delay of overflow circuit (ns)
Ripple-Carry	13.906	12.292	1.614
Carry-Lookahead	13.856	11.818	2.038
Carry-Select	12.593	11.447	1.146

A timing simulation for the CSA accumulator could not be performed. Trying to execute the testbench for the accumulator led to error messages from Xilinx stating the design has too many bonded comps of type "IBUF" found to fit the target device. The Xilinx tools attempt to take the input and output registers of the accumulator module and connect them to the Spartan 3E pins. There are only 232 of these pins so the design is considered overmapped. Implementation of a higher level module, or wrapper, could be used to circumvent this issue but all attempts to add a wrapper module had no effect.

The propagation delays of the adder modules aren't as different as their architectures would suggest. From studying the Technology schematics generated by the Xilinx tools, the design is heavily optimized during the synthesis and mapping stages. Propagating signals, like the carry chain formed by the RC adder architecture, found in the design are optimized to what appears to be carry-lookahead logic. Changing the synthesis option to optimize by area instead of speed seemed to have no effect on the way adder modules were optimized. Figure 39 shows an example of one LUT in the carry chain belonging to the RC adder.

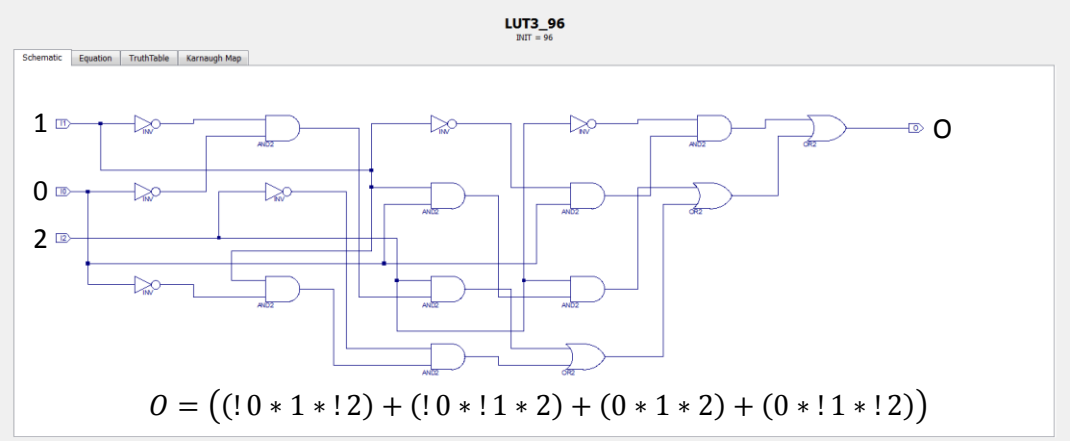


Figure 39. LUT from Ripple-Carry Adder Carry Chain

Table 5 summarizes Nexys2 resource utilization information extracted from the Xilinx Project Summary. Each module is placed in a separate project without overflow correction to produce these statistics. The results do not seem to give an accurate representation of in use Nexys 2 resources, especially for the CSA accumulator. This is most likely caused by the same issue that would not allow a post PAR timing simulation of the accumulator to run properly. The implementation of a "wrapper" module may help achieve more accurate results.

Table 5. Summary of Adder Module Nexys2 Resource Utilization

Module	Slices (%)	LUTs (%)
Ripple-Carry Adder	36 (0%)	63 (0%)
Carry-Lookahead Adder	51 (1%)	92 (0%)
Carry-Select Adder	54 (1%)	100 (1%)
Carry-Save Accumulator	689 (14%)	1199 (12%)

Multiplier Implementation

Figure 40 shows a behavioral simulation for the Booth multiplier. The correct output is calculated for each test case as seen in the results. Test case 1 shows a product calculated from inputs set to their maximum values. Case 2 tests all the possible partial products generated using the Modified Booth's Algorithm. Case 3 gives a test case for less complicated inputs, where the value of the product may be verified easily by inspection.

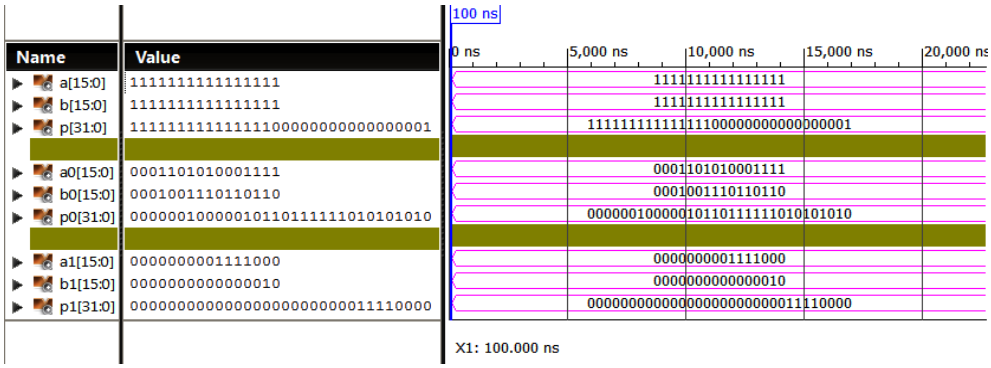


Figure 40. ***BOOTH_MULT_16BIT.vhd*** **Behavioral Simulation**

Figure 41 displays a post PAR simulation for the SA multiplier. Inputs are chosen as maximum for their bit sizes as shown in the simulation. The results give the propagation delay of the SA multiplier as 29.066 ns.

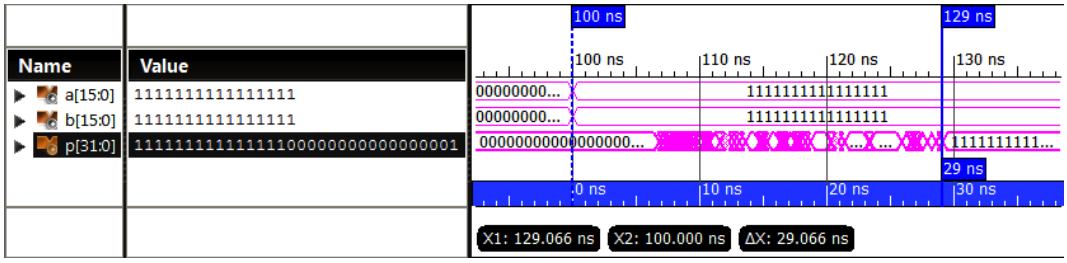


Figure 41. ***SA_MULT_16BIT.vhd*** **Timing Simulation**

A timing simulation for the Booth multiplier could not be performed on the Lite version of ISim software used for project simulations. Figure 42 shows timing simulation results for the Nexys2 dedicated MULT18x18 multiplier. According to the results, it is much faster than the SA multiplier with a propagation delay of only 14.197 ns. This is almost the same speed as the adder modules, making it considerably faster in comparison to the SA multiplier.

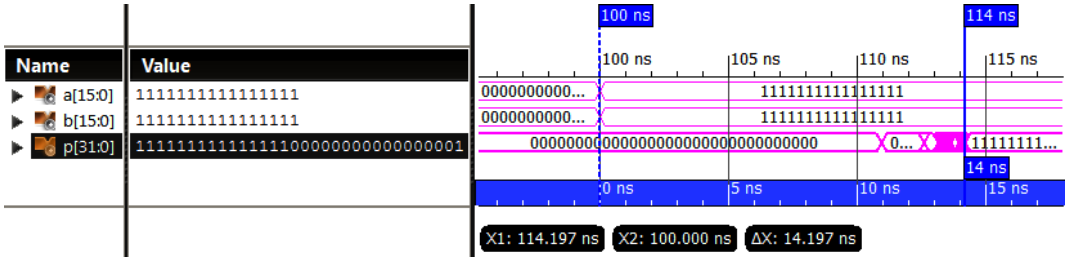


Figure 42. ***MULT18X18.vhd*** **Timing Simulation**

The RTL schematic created for the Booth multiplier show the presence of many adder units generated by Xilinx optimization. These extra adders doubtlessly have an effect on the size of the post PAR model and the capability of the Lite ISim software to perform the timing simulation. Altering the process that handles execution of the Modified Booth Algorithm in the multiplier module by replacing the loop with a series of equivalent sequential statements seemed to have no effect on reducing the redundant adders. Additionally, changing the Synthesis Properties option of Optimization Goal to optimize by area or speed also did not help.

Table 6 summarizes device utilization statists retrieved from the Xilinx Project Summary for the multiplier modules when placed alone in separate Xilinx projects. The redundant adders created in the Booth multiplier use up a lot of Nexys2 resources as evidenced by the 13% of slices and LUTs utilized as shown in the table. However, this value is not a good representation of actual resources used as explained in the following section of this report.

Table 6. Summary of Multiplier Module Nexys2 Resource Utilization

Module	Slices (%)	LUTs (%)
Shift-Add Multiplier	254 (5%)	495 (5%)
Booth Multiplier	629 (13%)	1236 (13%)
MULT18x18 Multiplier	0 (0%)	0 (0%)

Table 6 suggests and interesting characteristic about the dedicated MULT18x18 multiplier. Using this multiplier requires no other additional resources from the Nexys2. This, compiled with the fact that it has the shortest propagation delay of all the multipliers used in the project, makes it an ideal choice for implementing digital filters in the Nexys2.

System Implementation

Figure 43 shows a behavioral simulation for *FILTER.vhd*, the entire DSP system module. ADC input to the system is created within a process contained in the testbench. Simulations performed in this manner provided information used to debug the filter realization modules *NORMAL.vhd* and *CASCADE.vhd*. These particular results implement the *NORMAL.vhd* module while performing parallel calculations. Because of the complexity of the simulation, only the value for the calculated output is shown with the current input sample.

Figure 43. *FILTER.vhd* Behavioral Simulation – *NORMAL.vhd* I/O for 6 Samples

Table 7, created with Microsoft Excel, shows correct values for the filter implemented for this simulation. It shows the first 6 samples used as stimuli and the resulting output. The difference equation used to produce these results is also contained in Table 7. The values match with those shown Figure 43.

Table 7. *NORMAL.vhd* Excel Created Output Results for First 6 Samples

Difference EQ:	$y[n] = 1x[n] + 2x[n-1] + 3x[n-2] + 4x[n-3] + 1y[n-1] + 2y[n-2]$						
Signal:	x[n]	x[n-1]	x[n-2]	x[n-3]	y[n-1]	y[n-2]	y[n]
	1	0	0	0	0	0	1
	2	1	0	0	1	0	5
	3	2	1	0	5	1	17
	4	3	2	1	17	5	47
	5	4	3	2	47	17	111
	6	5	4	3	111	47	245

Another behavioral simulation, results shown in Figure 44, displays results using the *CASCADE.vhd* module with 2 filter stages performing serial calculations. This time various negative coefficients were specified to test calculations with negative values. Table 8 contains the difference equations for each filter stage used for this particular simulation. Figure 44 shows the current sample *x*n, w_i from stage A of the filter, y_i for stage A of the filter, and the resulting output for the first 4 samples used as stimuli.

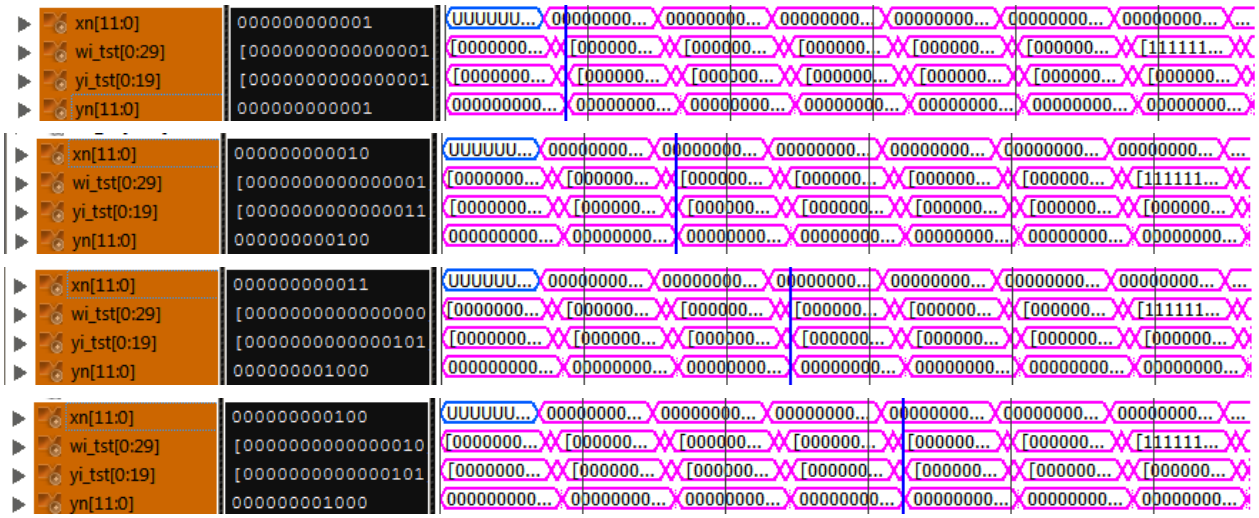


Figure 44. *FILTER.vhd* Behavioral Simulation – *CASCADE.vhd* I/O for 4 Samples

Table 8 contains values calculated at each filter stage to verify behavioral simulation results. Input *x*[n], intermediary signal w_i, delayed w_i signals, and the output y_i is shown for each filter stage. Behavioral simulation results match with tabulated values showing that values are calculated correctly at every stage of the filter.

Table 8. *CASCADE.vhd* Excel Created Output Results for 2 Filter Stages

Difference EQ:	$y[n] = 1x[n] + 2x[n-1] + 3x[n-2] + 4x[n-3] - 1y[n-1] - 2y[n-2]$				
Stage A:	**x[n]**	**w_A[n-1]**	**w_A[n-1]**	**w_A[n-2]**	**y_A[n]**
	1	1	0	0	1
	2	1	1	0	3
	3	0	1	1	5
	4	2	0	1	5
Stage B:	**x_B[n]**	**w_B[n]**	**w_B[n-1]**	**w_B[n-2]**	**y[n]**
	1	1	0	0	1
	2	2	1	0	4
	5	1	2	1	8
	5	0	1	2	8

Table 9 summarizes Nexys2 resource utilization when implemented using *NORMAL.vhd* configured for serial calculations. From the table results, the choice of which adder module to use has little effect on the amount of Nexys2 resources required for implementation. The data contained in Table 9 shows inconsistencies with results gathered in Table 5, which shows that more slices and LUTs are required to implement the CLA adder than the RC adder, as expected.

Table 9 shows that when implemented in the system the RC adder and SA multiplier use 1242 slices and 2084 LUTs respectively, and the CLA adder and SA multiplier use 1239 slices and 2070 LUTs respectively. By this data the RC adder uses more Nexys2 resources than the CLA adder, conflicting with results in Table 5. However, as expected the least amount of device resources are required to implement the dedicated MULT18x18 multiplier.

Table 9. Summary of Nexys2 Resource Utilization for Complete System – *NORMAL.vhd* & Serial Calculations

Adder	Multiplier	Slices (%)	LUTs (%)
	SA	1242 (26%)	2084 (22%)
RC	Booth	1505 (32%)	2615 (28%)
	MULT18x18	991 (21%)	1529 (16%)
	SA	1239 (26%)	2070 (22%)
CLA	Booth	1514 (32%)	2638 (28%)
	MULT18x18	1017 (21%)	1566 (16%)
	SA	1244 (26%)	2080 (22%)
CSE	Booth	1522 (32%)	2649 (28%)
	MULT18x18	1007 (21%)	1542 (16%)

Table 10 provides system information when implemented using the *NORMAL.vhd* module configured for parallel calculations. When configured for this implementation, a length 9 filter generates 10 multipliers and a length 19 filter generates 20 multipliers and the CSA accumulator is the only adder module used for accumulation.

The amount of device resources used per multiplier when implemented in the entire system may be calculated from the difference in resources between the two filter lengths. For example, the Booth multiplier uses (2410 – 1370) / 10 = 104 slices and (3869 – 2215)/10 = 165.4 LUTs, much less than suggested by the data contained in Table 6 of 629 slices and 1236 LUTs. Similar inconsistencies exist with the other multiplier modules. A similar analysis is not possible for the adder modules without significant modifications to many of the system modules.

Table 10. Summary of Nexys2 Resource Utilization for Complete System – *NORMAL.vhd* & Parallel Calculations

	Length 9 Filter		Length 19 Filter	
Multiplier	Slices (%)	LUTs (%)	Slices (%)	LUTs (%)
Shift-Add	916 (19%)	1468 (15%)	1057 (22%)	1780 (19%)
Booth	1370 (29%)	2215 (23%)	2410 (51%)	3869 (41%)
MULT18x18	935 (20%)	1473 (15%)	1065 (22%)	1767 (18%)

Another inconsistency is apparent from the Table 10 data. In most cases, Nexys2 resource utilization is actually greater when using the MULT18x18 dedicated multiplier than it is for the SA multiplier. Only the LUTs used for a length 19 filter are less for the MULT18x18 than for the SA multiplier.

Nexys2 Implementation

This section of the report provides information about the implementation of the DSP system in the Nexys2. Several test cases analyze the behavior of the system under different conditions. The maximum allowable filter length for any test filter is 19. A filter of this size could feasibly incorporate every MULT18x18 multiplier in the Nexys2 when configured for normal direct form I filter realization and parallel calculations, 1 for each term of the filter. Because of the limited number of MULT18x18 multipliers in the Nexys2, the maximum number of multipliers allowed for any given filter is 20.

To verify timing data for binary arithmetic components gathered from post PAR simulation results required a simple test. This is accomplished with a small process that gave the adder or multiplier under examination inputs on the rising edge of the 50 Mhz system clock and made the attempt to display the respective sum or product on the Nexys2 LEDs at the beginning of the next clock cycle. This test shows that all arithmetic components, except the CSA accumulator, can calculate their respective sums and products in one clock period of the 50 MHz system clock. Conducting the same test on the accumulator showed that 2 clock periods were sufficient for it to accumulate the maximum allowable number of 20 filter terms.

Normal Direct Form I Implementation

Testing the implementation of *NORMAL.vhd* employed 2 test cases. These were selected to check proper operation for length 19 filters and to verify proper calculations while using negative filter coefficients, both while executing serial and parallel calculations. Frequency responses for each case created with MATLAB gave results for comparison against oscilloscope plots created from Nexys2 implementation. Filter coefficients for all test cases were determined with a magnitude of 1 for zeros and a magnitude of .95 for poles and all test filters have unity gain.

Test Case 1

Test case 1 implements a length 19 weighted moving average filter. Table 11 shows all relevant information regarding test case 1.

Table 11. Filter Specifications for *NORMAL.vhd* Test Case 1

Realization Structure	Normal Direct Form I
Adder	Ripple-Carry
Multiplier	Shift-Add
Scaling	16-Bit
Filter Length	19
Sampling Freq	44.1 kHz
Freq of First Zero	4.009 kHz
Ak	[0, … , 0]
Bk	[297, 595, 893, 1191, 1489, 1787, 2085, 2383, 2680, 2978, 2978, 2680, 2383, 2085, 1787, 1489, 1191, 893, 595, 297]

The frequency response of this filter created with MATLAB, displayed in Figure 45, shows that the first zero of the weighted moving average filter occurs at a digital frequency of about .1 cycles per sample. For a 44.1 kHz sampling rate, this is an analog frequency of about 4 kHz.

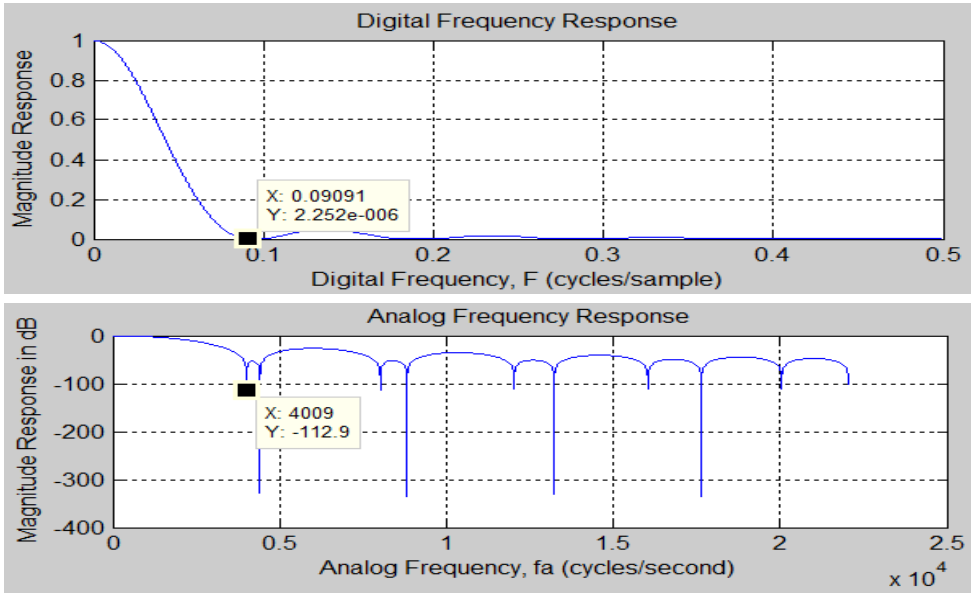


Figure 45. *NORMAL.vhd* Test Case 1 Frequency Response Plot from MATLAB

Figure 46 contains the frequency response plot created from the oscilloscope data. It matches closely with the one generated by MATLAB, with the first zero appearing at around 4 kHz. Identical results were achieved for both serial and parallel calculation implementations.

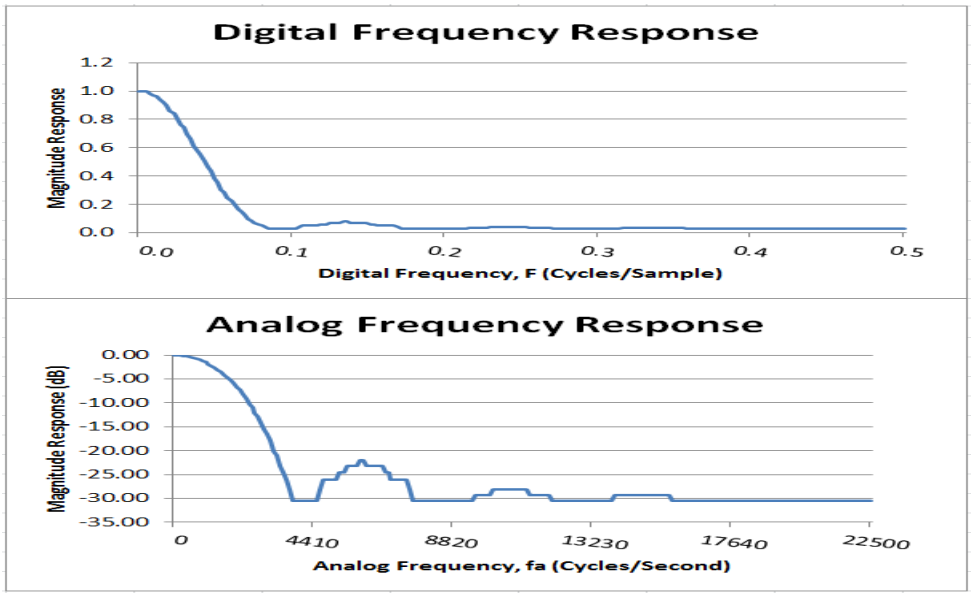


Figure 46. *NORMAL.vhd* Test Case 1 Frequency Response Plot from Experimental Data

Figure 47 displays a scope capture of the first zero. Displayed in the figure, the input to the system is displayed on channel 2 with amplitude of 2.05 V and a frequency of 4.006 kHz. Channel 1 shows the resulting output with amplitude of only 60 mV.

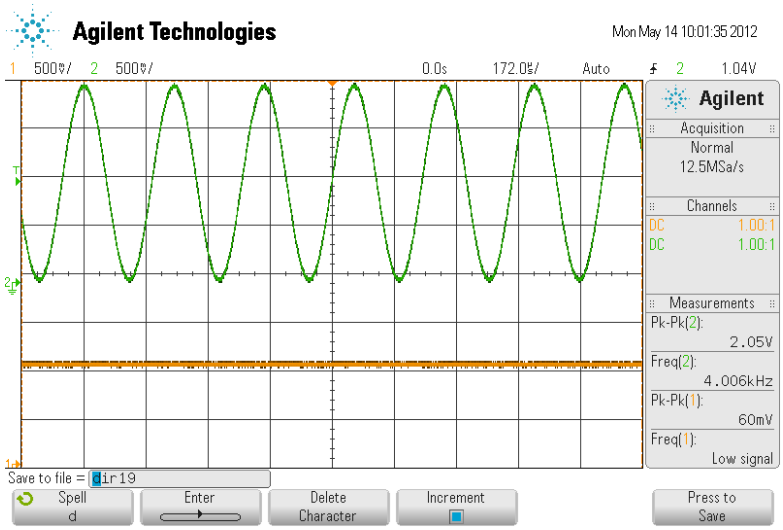


Figure 47. *NORMAL.vhd* Test Case 1 Scope Capture of First Zero

Test Case 2

This case is designed to test calculations using negative numbers. Specifications for the 2nd order IIR test filter are shown in Table 12.

Table 12. Filter Specifications for *NORMAL.vhd* Test Case 2

Realization Structure	Normal Direct Form I
Adder	Carry-Select
Multiplier	Booth
Scaling	9-Bit
Filter Length	4
Sampling Freq	44.1 kHz
Freq of Zeros	15 kHz
Ak	[-260, -231, 0, … , 0]
Bk	[243, 261, 243, 0, … , 0]

Figure 48 shows a frequency response plot created with MATLAB for the filter specified in Table 12. This plot shows the zero for the filter at digital frequency of about .3401. This corresponds to an analog frequency of about 15 kHz, as desired.

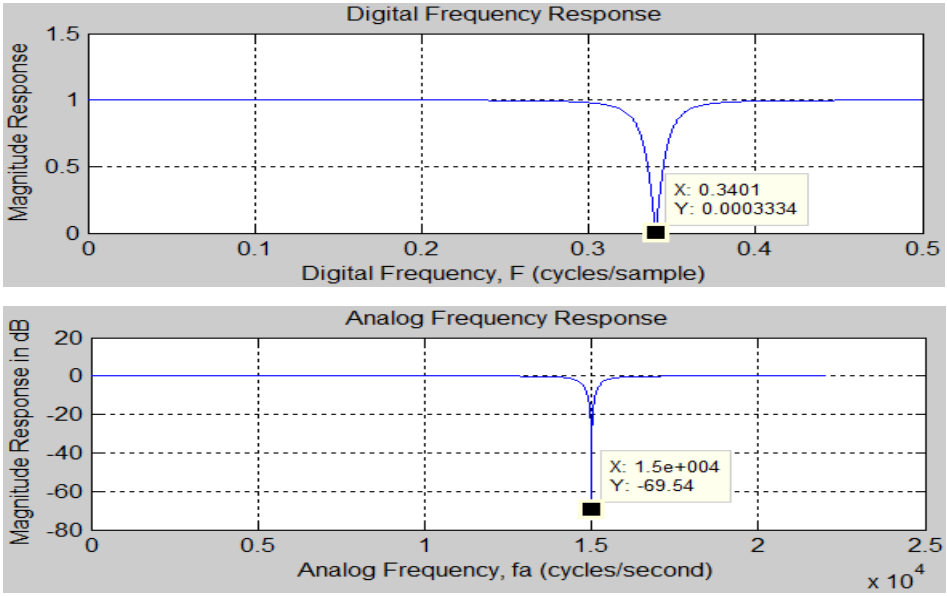


Figure 48. ***NORMAL.vhd*** **Test Case 2 Frequency Response Plot from MATLAB**

Figure 49 contains a frequency response plot generated with experimental data from the oscilloscope. It is very similar to the MATLAB generated plot for this test case. The parallel calculation implementation for this test case gave identical results.

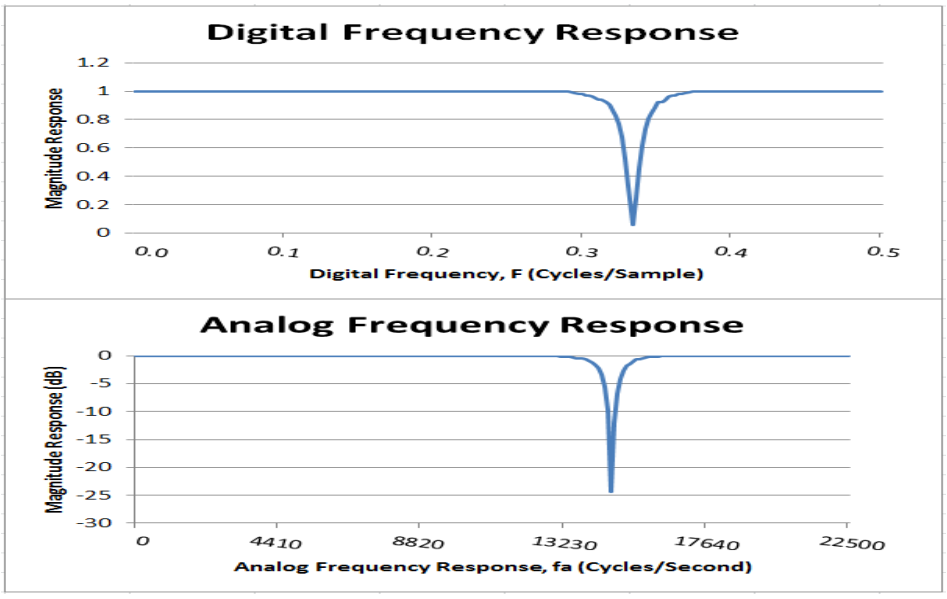


Figure 49. ***NORMAL.vhd*** **Test Case 2 Frequency Response Plot from Experimental Data**

Figure 50 shows a scope capture of the first zero, occurring at about 15 kHz. Channel 2 contains the input signal, seen with amplitude of 2.09 V and a frequency of 15 kHz. The output on channel 1 shows amplitude of 120 mV, a value very close to 0 V.

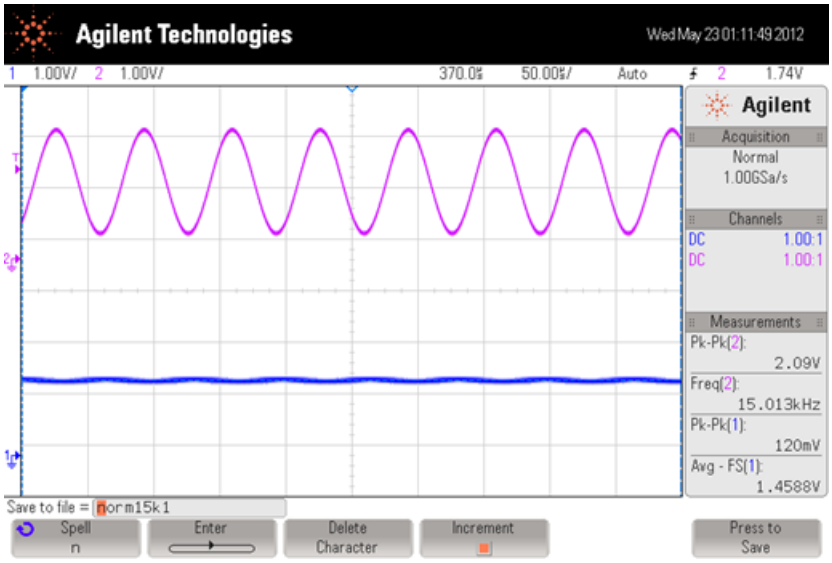


Figure 50. *NORMAL.vhd* Test Case 2 Scope Capture of Zero

Cascade Direct Form II Implementation

Complications arose during the Nexys2 implementation of *CASCADE.vhd*. The parallel calculation implementation would not produce an output and only a single stage filter would produce output during serial calculation implementation. More details about these problems are contained in this section of the report. Any filters used in test cases for this module were designed with unity gain and filter coefficients created from zero magnitudes of 1 and pole magnitudes of .95.

Test Case 1

Table 13 shows all relevant filter information regarding test case 1 for Nexys2 implementation of *CASCADE.vhd*. This case is designed to test basic operation of the module. The filter is composed of only one 2nd order filter stage with a zero at 10 kHz as specified in the table.

Table 13. Filter Specifications for *CASCADE.vhd* Test Case 1

Realization Structure	Cascade Direct Form II
Adder	Carry-Save
Multiplier	MULT18x18
Scaling	13-Bit
Filter Order	2
Sampling Freq	44.1 kHz
Freq of Zero	10 kHz
Aki	[1132, -3695, 0, … , 0]
Bki_S	[3894, -1133, 3894, 0, … , 0]

Figure 51 shows a frequency response plot created with MATLAB for the filter used for test case 1. This plot shows the zero for the small filter at digital frequency of about .2268. This corresponds to an analog frequency of about 10 kHz.

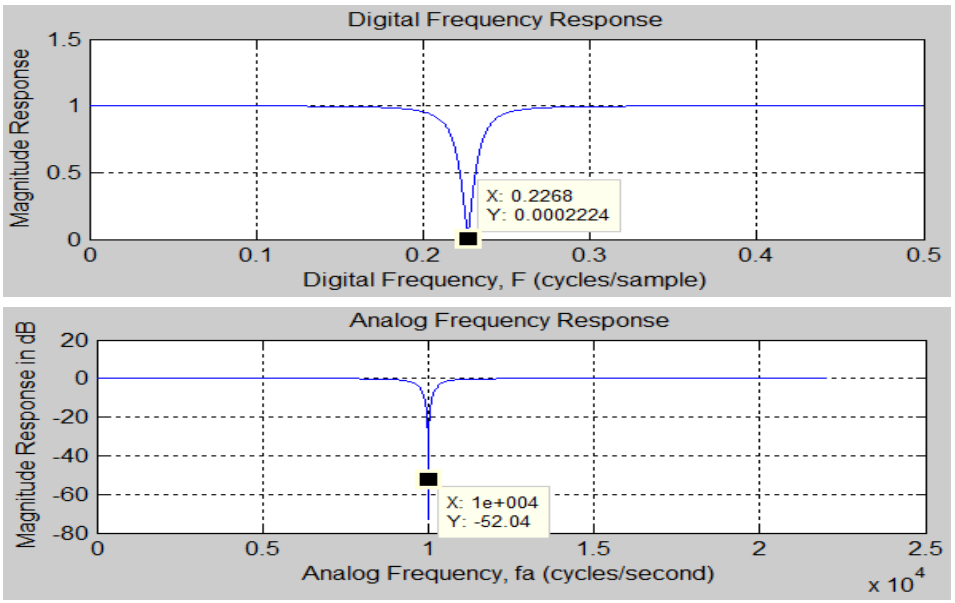


Figure 51. *CASCADE.vhd* Test Case 1 Frequency Response Plot from MATLAB

The frequency response plot shown in Figure 52 is produced from experimental data gathered form the oscilloscope. It closely resembles the plot created using MATLAB, with the location of the zero shown at around 10 kHz.

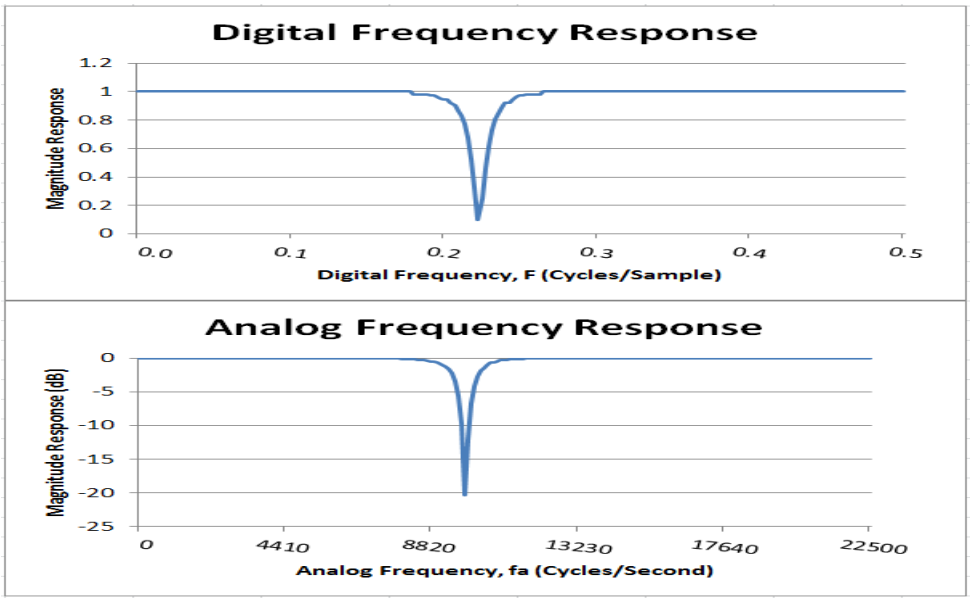


Figure 52. *CASCADE.vhd* Test Case 1 Frequency Response Plot from Experimental Data

Figure 53 displays a scope capture of the zero of the filter used for this test case. The input signal on channel 2 has amplitude of 2.09 V with a frequency of about 10 kHz. The output, shown on channel 1 has amplitude of 80 mV.

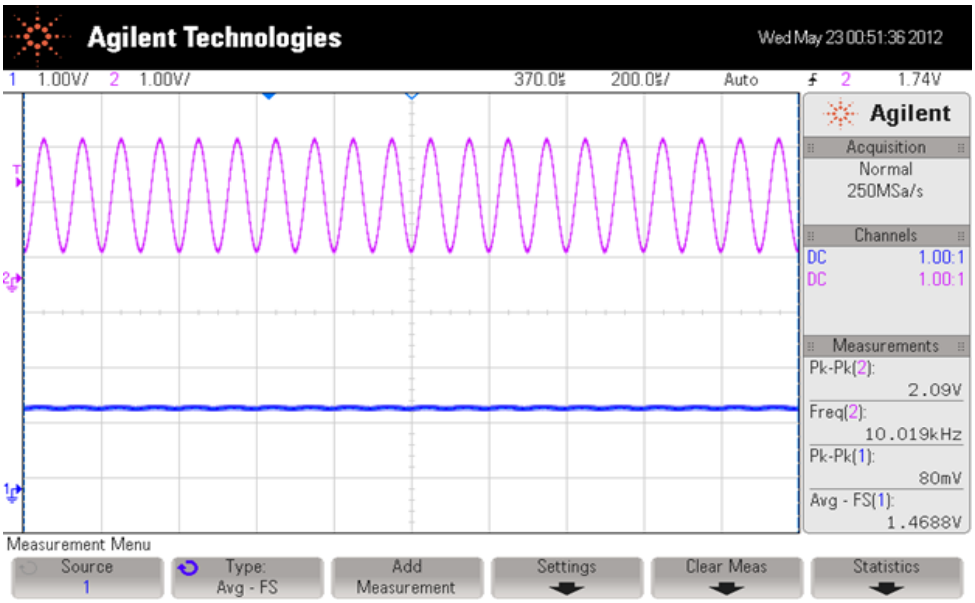


Figure 53. *CASCADE.vhd* Test Case 1 Scope Capture of Zero

Test Case 2

Test case 2 is designed to test multi-stage filter implementation for this module. Table 14 shows specifications for the filter, which contains 3 filter stages with zeros at 2 kHz, 9 kHz, and 19 kHz.

Table 14. Filter Information for *CASCADE.vhd* Test Case 2

Realization Structure	Normal Direct Form I
Adder	Booth
Multiplier	Carry-Select
Scaling	13-Bit
Filter Order	6
Sampling Freq	44.1 kHz
Freq of Zeros	2 kHz, 9 kHz, 19 kHz
Aki	[7466, -3695, 2213, -3695, -7057, -3695, 0, … , 0]
Bki_S	[3893, -7471, 3893, 3894, -2215, 3894, 3893, 7062, 3893, 0, … , 0]

Figure 54 shows a digital frequency response plot created with MATLAB for the 3 stage filter specified in Table 14. Zeros occur at digital frequencies of .04536, .2041, and .4308, corresponding to analog frequencies of 2 kHz, 9 kHz, and 19 kHz respectively.

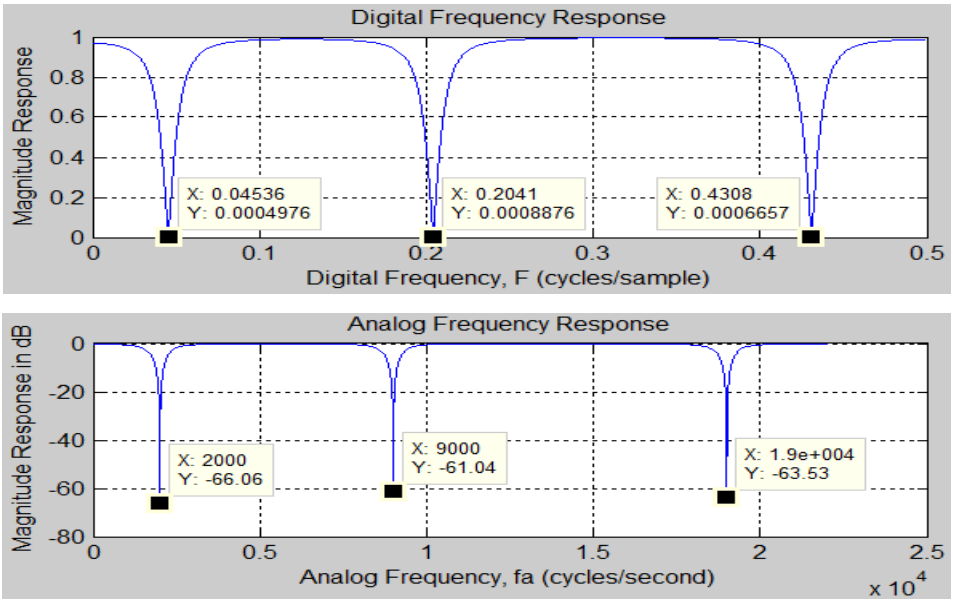


Figure 54. *CASCADE.vhd* Test Case 2 Frequency Response Plot from MATLAB

When implemented in the Nexys2, the filter specified for this case produces a white noise output. During the investigation to resolve this issue the scope capture shown Figure 55 was created from a single stage filter designed to have a zero at 2 kHz.

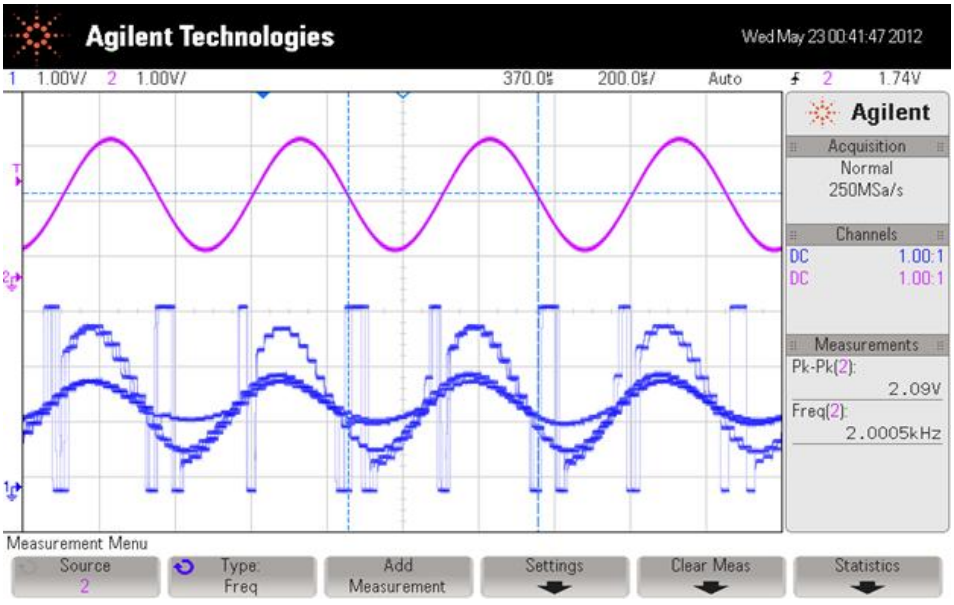


Figure 55. *CASCADE.vhd* Test Case 2 Scope Capture – Evidence of Overflow Error

The figure shows the input signal on channel 2 with amplitude of 2.09 V at a frequency of 2 kHz. This frequency is the location of the zero specified by the filter design. It can be determined from the scope capture that with this filter specification some samples are saturated to the upper and lower boundaries of the DAC output voltage range, leading to the conclusion that there is an over flow error in the *CASCADE.vhd* module VHDL code. While this error is not present for test case 1, this same error must be the cause of the white noise generated for the 3 stage filter specified for test case 2.

The calculations for the direct form II filter realization are much more complex that those used for direct form I. Because of its structure, the intermediary signal w_i must be descaled and checked for overflow, and the same must occur when calculating the intermediary stage output signal y_i. For the 3 stage filter specified for test case 2, this means that descaling occurs 6 times and overflow is checked and corrected another 6 times before producing the final output. This leaves a lot of room for error.

As an attempt to simplify calculations and have a working VHDL module that could easily implement multi-stage 2nd order IIR filters, a third module using a direct form I realization was experimented with. The *CASC_DFI.vhd* module uses a slightly modified version of the *NORMAL.vhd* VHDL code. The oscilloscope and signal generator were not available to test this module so testing was performed with a digital multimeter and output provided from a computer using a freeware program called ToneGen. With these tools, one filter stage was determined functional but all attempts to implement a multi-stage filter failed. Because of time constraints, neither the *CASCADE.vhd* nor *CASC_DFI.vhd* modules could be fully completed.

Audio Implementation

Two free programs helped facilitate testing of the audio interfaces used for the project. A program called ToneGen generates a simple sign wave output at frequencies ranging from 100 Hz to 15 kHz through the computer's soundcard. Another program called Soundcard Scope monitors the computer microphone input and displays the signal like an oscilloscope. Figure 56 shows the ToneGen GUI configured to output a 500 Hz signal.

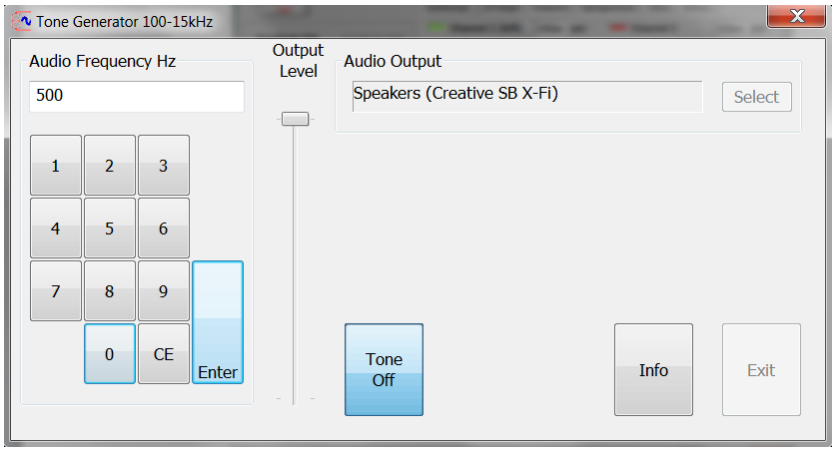


Figure 56. ToneGen GUI Configured for 500 Hz Output

Figure 57 shows the Soundcard Scope GUI while monitoring the computer microphone jack. A 10 µF capacitor couples DAC output to the microphone jack to remove the DC offset from the signal. As seen in the figure, the signal is not very smooth and is oddly shaped. This imperfection is heard when listening to the microphone jack as a moderate amount of noise although the filtered tone can still be heard.

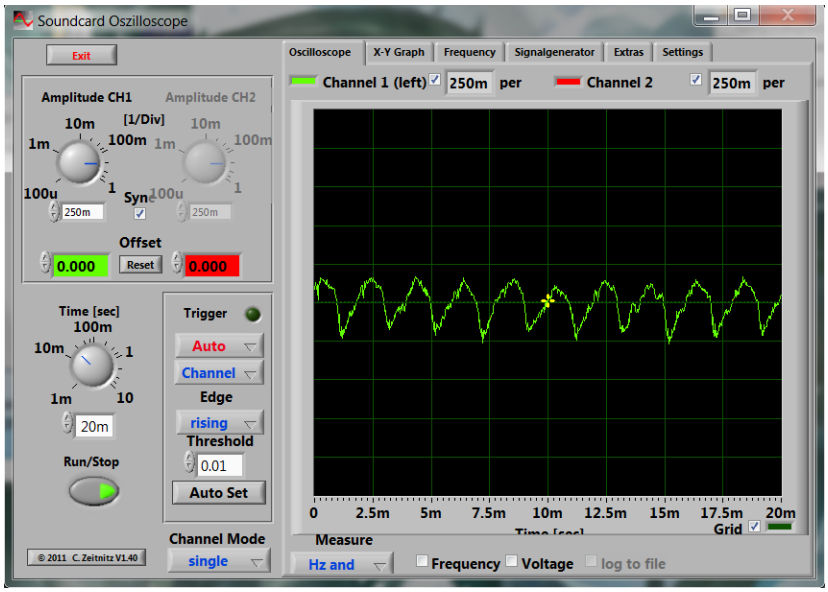


Figure 57. Soundcard Scope Displaying Filtered Output from DAC at 500 Hz

Figure 58 shows the Soundcard Scope GUI displaying the zero of the test filter used to create the audio interface experiment. It occurs at 5 kHz. When listening to the microphone port the tone is barely audible but there is still a considerable amount of noise.

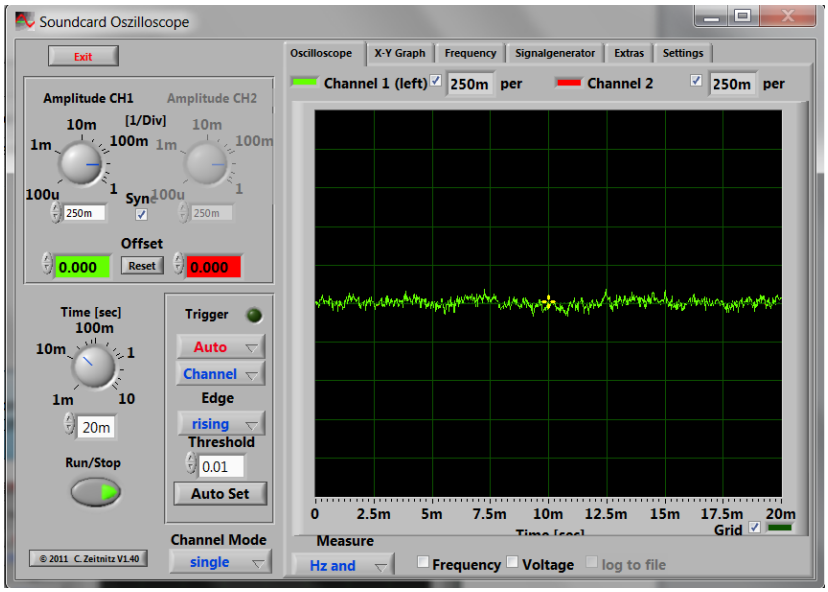


Figure 58. Soundcard Scope Displaying Filter Zero at 5000 Hz

The roughness and odd shape of the signal that reaches the microphone jack might be introduced by the AC coupling capacitor. A different device, like another level shifter, might provide a more stable signal to the speaker or microphone jack. Because of time constraints, other methods to remove the DC offset from the output signal could not be tested.

DSP Performance Analysis

This section provides an analysis of the maximum achievable sampling rate for a unity gain length 4 moving average filter and the maximum filter length for a sampling rate of 44.1 kHz. A behavioral simulation for each case allows for the estimating of these limitations.

Figure 59 shows a behavioral simulation for the estimation of maximum sampling rate of the complete system using the *NORMAL.vhd* module and serial calculations. As seen in the figure, sample acquisition begins on the rising edge of SAMP_CLK and filter calculations begin short after, after the assertion of SAMP_DONE. The way the *NORMAL.vhd* FSM is constructed, once filter calculations begin they do no stop until complete, even if the ADC begins a new sample.

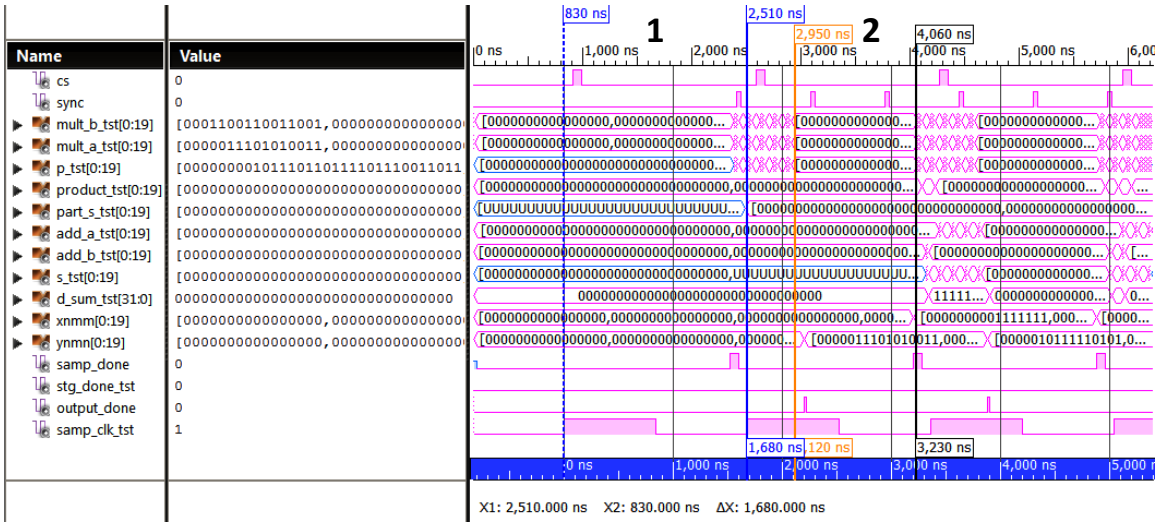


Figure 59. Estimated Maximum Sampling Rate for *NORMAL.vhd* Serial Calculation Implementation

The information contained in Figure 59 suggests that with this realization the maximum sampling rate is not limited by filter length by instead by the time it takes the A DC to take a sample. If the rising edge of SAMP_CLK happens to come before the assertion of the SAMP_DONE flag then the ADC will wait until the next SAMP_CLK rising edge to take the next sample. Essentially, this is equivalent to slowing the sampling rate, changing where the zeros appear in the frequency response. From the simulation results shown in Figure 59 in cursor interval 1, a maximum sampling rate of about 525 kHz still allows for correct output results.

Figure 59 leads to another important conclusion. Because of sampling rate dependence on ADC sample acquisition time, the filter length can be increased beyond 4 for the maximum sampling rate of 525 kHz. The gap between multiplication sequences, shown in cursor interval 2, provides evidence of this. Using a parallel calculation implementation would take even less time to calculate an output. This implies that this sampling rate is the maximum for any implementation designed for this project.

The results gathered from Figure 59 are impossible to verify in the Nexys2 with the ToneGen application. For the proposed maximum sampling rate of 525 kHz, the first zero for the length 4 moving average filter appears somewhere around 119 kHz. The ToneGen application is limited by the computer sound card and can achieve a maximum frequency of only 15 kHz.

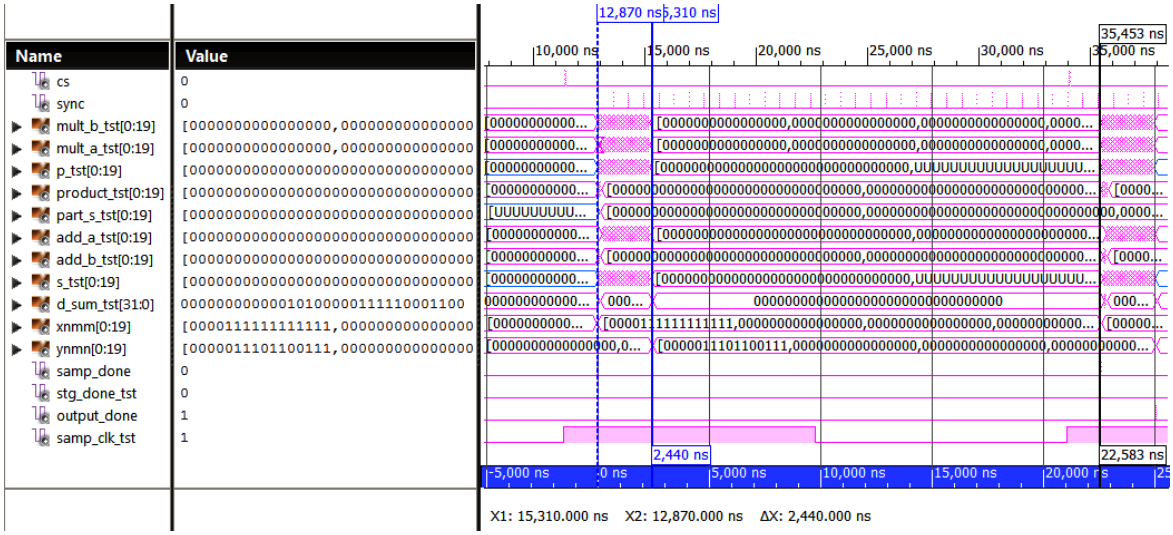


Figure 60. Estimated Maximum Filter Length for *NORMAL.vhd* with 44.1 kHz Sampling Frequency

Figure 60 shows a behavioral simulation for *NORMAL.vhd* using serial calculations for a length 19 weighted moving average filter. The results display that it takes about 2,440 ns to finish the calculation sequence. The time in between calculation sequences is shown as 35,453 ns – 12,870 ns = 22,583 ns. This implies that a filter length of about 22,583 / 2,440 * 19 ≈ 175 could be achieved for this filter realization. These results could not be verified in the Nexys2 without numerous modifications to the VHDL code to change register sizes, indexing, and loop iterations throughout project modules.

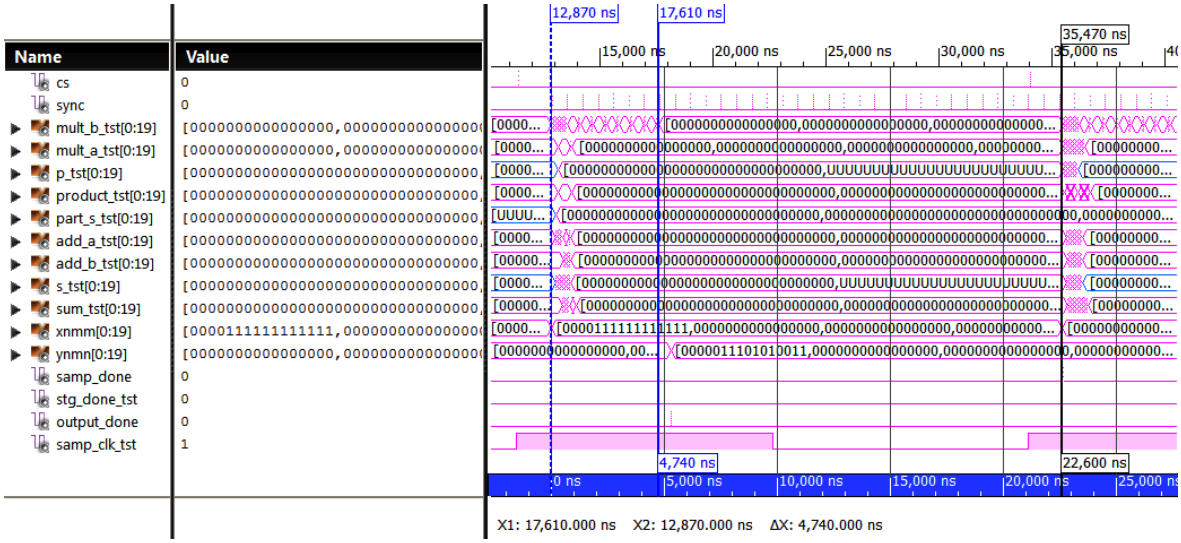


Figure 61. Estimated Maximum Filter Stages for *CASCADE.vhd* with 44.1 kHz Sampling Frequency

Figure 61 shows a behavioral simulation for *CASCADE.vhd* using a serial calculation implementation and 9 filter stages. The time it takes to complete the calculation sequence is 4,740 ns and the time in between calculation sequences is 22,600 ns as displayed in the results. With this information the number of filter stages is estimated as about (22,600 / 4,720) * 9 ≈ 43 stages. These results were not verified in the Nexys2 because of the numerous modifications to VHDL modules required to implement a filter of this size.

Arithmetic Component Analysis

This section of the report summarizes information regarding the analysis of arithmetic components designed for the project. It contains comparisons for delay and device resource utilization, and a brief explanation of power analysis.

Timing Analysis

Table 15 contains a summary of analysis performed on arithmetic component architecture. It contains gate delay estimations acquired from the adders' schematics and partial product analysis acquired from multiplier algorithms. It also summarizes propagation delay times gathered from timing simulations.

Table 15. Summary of Arithmetic Component Delay Information

Component	Gate Delay	Partial Products (16-Bit Inputs)	Prop Delay (ns) (No Overflow Correction)	Prop Delay (ns) (Overflow Correction)
Ripple-Carry Adder	64	–	12.292	13.906
Carry-Lookahead Adder	28	–	11.818	13.856
Carry-Select Adder	22	–	11.447	12.593
Shift-Add Multiplier	–	16	29.066	–
Modified Booth Multiplier	–	9	–	–
MULT18x18 Multiplier	–	–	14.197	–

As expected from gate delay analysis, the RC adder has the slowest architecture, with a propagation delay of 12.292 ns without overflow correction according to post PAR simulation results. From gate delay analysis, the CSE gate delay estimation is the lowest with only 22 gate delays to calculate the sum. Post PAR simulation results give the propagation delay without overflow correction as 11.447 ns, the fastest time of the adders, coinciding with the gate delay analysis result.

Unfortunately, a post PAR simulation for the Booth Multiplier could not be performed as explained in the Design Specifications section of this report. From looking at partial product analysis results, it should have a propagation delay of about half the SA multiplier's propagation delay of 29.066 ns. As expected, the MULT18x18 dedicated multiplier has the fastest propagation of the multipliers simulated, listed as 14.197 ns.

Comments

According to post PAR results, all components except for the SA multiplier have a propagation delay that is less than 1 clock period of the 50 MHz system clock. This implies that they can calculate their respective sums and products in only one clock cycle. The SA multiplier would take 2 clock cycles according to these results although the hardware test performed in the Nexys2 Implementation section verifies that its product is calculated in only one clock cycle.

This means that the propagation delay of arithmetic components designed for this project has no effect on filter performance characteristics like maximum sampling rate and maximum filter length. All will impose the same limitations on filter specifications. This helps to simplify the choice of which arithmetic components to use for a given filter realization. Because arithmetic component speed is not a factor, only component size needs consideration.

Resource Analysis

Table 16 gives a summary of resource usage statistics for the project's arithmetic components. It consolidates information contained in the Design Specification section of this report, showing the numbers of slices and LUTs used for implementation.

Table 16. Nexys2 Resource Utilization for All Arithmetic Components

Component	Slices (%)	LUTs (%)
Ripple-Carry Adder	36 (0%)	63 (0%)
Carry-Lookahead Adder	51 (1%)	92 (0%)
Carry-Select Adder	54 (1%)	100 (1%)
Carry-Save Accumulator	689 (14%)	1199 (12%)
Shift-Add Multiplier	254 (5%)	495 (5%)
Modified Booth Multiplier	629 (13%)	1236 (13%)
MULT18x18 Multiplier	0 (0%)	0 (0%)

These Nexys2 resource statistics are expected based on arithmetic architecture analysis. The simpler architectures, like the RC adder and the SA multiplier, use less Nexys2 resources than their more complicated counterparts. The faster components, like the CSE adder and the Booth multiplier, use the most system resources.

Comments

To acquire this data, each component was placed in a Xilinx project by itself. The slices and LUTs given for the adder modules seem to be the most accurate. For example, gate delay for the RC adder is about 64. If 1 LUT is needed for each gate, then it would take about 64 LUTS, or 32 slices, to implement the RC adder in the Nexys2. These values are very close to the ones given in Table 16.

The slices and LUTs shown for the CSA accumulator and the Booth multiplier seem less accurate. It is possible to implement 20 Booth multipliers in a filter realization for use in parallel calculations as shown in Table 10. When implemented with the entire DSP system, the total slices used, including the 20 Booth multipliers is only 2410. This is much less than the utilized slices suggested by the data in Table 16. The CSA accumulator appears to share this same problem when considering the same logic.

A “wrapper” module was designed and implemented as an attempt to remedy this issue. The wrapper module concept is supposed to hide unconnected signals and registers coming from the test module so they are not mapped to Nexys2 resources during the Xilinx tools mapping process. Every attempt to design a wrapper module that would accomplish this task proved unsuccessful.

Power Analysis

The attempt to perform power analysis on the system using Xilinx XPower Analyzer yielded little information. Power estimations are performed with the software using 3 different files as explained in the Xilinx ISE help file in the XPower Analyzer Files section. These files include the following:

- **Physical Constraints File (PCF)** – Contains physical constraints created by the Xilinx MAP process and physical constraints translated from the User Constraint File (UCF) by the MAP program.

- **Settings File** – A file in XML format that contains current settings like default toggling rates, voltages, and ambient temperature. It may be generated form an XPower Estimator spreadsheet.

- **Simulation Activity File (SAIF or VCD)** – Simulation tools, like ISim, may generate SAIF of VCD files from post PAR simulations. SAIF and VCD files contain the following information:
    - **VCD (Value Change Dump)** – Contains a series of time-ordered value changes for the signals in the simulation model.
    - **SAIF (Switching Activity Interchange Format)** – A smaller version of the VCD file created by omitting timing information. Only contains signal switching information.

Both the PCF and either the SAIF or VCD files were used in attempts to perform power analysis on the binary arithmetic components without conclusive results. Each attempt yielded identical information, where the XPower software did not appear to receive any signal switching information from the files. Project time constraints did not allow for further investigation to resolve the issues.

Conclusions

Binary arithmetic components have expected characteristics, with the larger, more complex components using more system resources and having faster propagation times. However, architecture has little effect on DSP performance when used in the filter realization structures created for Nexys2 implementation. This is despite the fact that each has a different architecture and a different propagation delay. This is because each can calculate its respective output in less than 20 ns, the period of the 50 Mhz Nexys2 system clock. If a system with a faster clock is used for implementation, component architecture would have more drastic effects on performance.

Not all filter realization structures used in the final design functioned properly. The cascade direct form II module, *CASCADE.vhd*, would produce no output for parallel calculation implementation or for multi-stage filters using serial calculation implementation. This is most likely due to an overflow error in the VHDL code for this module. The normal direct form II module, *NORMAL.vhd*, seemed to work excellently in all cases.

The audio conversion interface used for the Nexys2 could also use some improvements. The level shifter circuit placed on the input did a good job of providing a clean input signal with an offset of 1.5 V. The output capacitor was able to remove this offset before the signal is sent to the listening device, but introduces a small amount of noise. A better solution may be another level shifter to drop the voltage back to balance around 0 V.

Estimations for DSP performance limitations show the Nexys2 has capabilities to implement complex digital filters. The maximum filter length for serial calculation implementations of 175, the maximum number of filter stages of 43, and the maximum sampling rate of 525 kHz could be improved by some minor optimizations to the filter realization modules' VHDL code. These modifications could not be accomplished due to project time constraints.

The Xilinx software optimizes any module selected for implementation, in some undetermined way, to be more efficient. This is evidenced by the similarities in propagation delay between adder modules despite their considerable contrasts in architecture. This simplifies decisions regarding arithmetic component selection for a particular filter design. The carry-chains included in simpler architectures, like the RC adder, are replaced by Xilinx optimization with more complex logic, making them more desirable for implementation due to the ease of their design.

Overall, the Nexys2 provides an excellent hardware resource for students to examine many aspects of digital filter realization. The complete system shows an overview of how all parts of a DSP system work together to accomplish filtering operations. Using this project, students may study a large amount of information regarding how binary arithmetic components are organized and used in digital filters, as well as how their architecture influences their performance, design, and hardware implementation.

References

Using Embedded Multipliers in Spartan-3 FPGAs. (2003, May 13). Retrieved February 22, 2012, from Xilinx Support: http://www.xilinx.com/support/documentation/application_notes/xapp467.pdf

Non-Inverting Op-Amp Level Shifter. (2004). Retrieved April 19, 2012, from Daycounter, Inc: http://www.daycounter.com/Circuits/OpAmp-Level-Shifter/OpAmp-Level-Shifter.phtml

VHDL Code: Booth Multiplier. (2010, December 18). Retrieved February 5, 2010, from Codz Home: http://codzhome.blogspot.com/2010/12/vhdl-code-booth-multiplier.html

Carry Save Adder Implementation. (2011, December 11). Retrieved September 20, 2011, from ECEN 6263 Advanced VLSI Design: http://lgjohn.okstate.edu/5263/lectures/csa.pdf

Carry-Select Adder. (2011, October 31). Retrieved January 17, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Carry-select_adder

Historical DSP Applications. (2011). Retrieved May 2, 2012, from Signalogic: http://www.signalogic.com/index.pl?page=dsp_app

Serial Peripheral Interface Bus. (2011, December 10). Retrieved January 20, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

Adder (Electronics). (2012, April 4). Retrieved April 10, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Adder_(electronics)

Booth's Multiplication Algorithm. (2012, May 10). Retrieved May 28, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Booth's_multiplication_algorithm

Carry-Lookahead Adder. (2012, April 25). Retrieved May 29, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Carry-lookahead_adder

Carry-Save Adder. (2012, May 13). Retrieved May 28, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Carry-save_adder

Kogge-Stone Adder. (2012, January 27). Retrieved January 7, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Kogge%E2%80%93Stone_adder

Multiplication Algorithm. (2012, May 25). Retrieved May 28, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Multiplication_algorithm

Andraka Consulting Group, I. (2007, March 16). *Multiplication in FPGAs.* Retrieved April 7, 2012, from Andraka Consulting Group, Inc: http://www.andraka.com/multipli.htm

Compton, K. (n.d.). *Carry Lookahead Adders.* Retrieved January 13, 2012, from University of Wisconsin Madison Computer Science Home Page: http://pages.cs.wisc.edu/~jsong/CS352/Readings/CLAs.pdf

Da Huang, N. A. (n.d.). *Modified Booth Encoding Radix-4 8-Bit Multiplier.* Retrieved February 6, 2012, from Duke University Electrical & Computer Engineering: http://people.ee.duke.edu/~jmorizio/ece261/F08/projects/MULT.pdf

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

Hardware Algorithms for Arithmetic Modules. (n.d.). Retrieved January 9, 2012, from Tohoku University Computer Structures Laboratory: http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html#ppa_wlc

Knagge, G. (2010, July 27). *ASIC Design for Signal Processing.* Retrieved February 5, 2012, from GeaffKnagge.com: http://www.geoffknagge.com/fyp/booth.shtml

Loh, P. (2005, February 2). *Carry-Save Addition.* Retrieved January 13, 2012, from Engineering Texas A & M University: http://www.ece.tamu.edu/~sshakkot/courses/ecen248/csa-notes.pdf

Morgan, M. (2003, Fall). *ECE 8053 Introduction to Computer Arithmetic.* Retrieved September 5, 2011, from Mississippi State University Department of Electrical and Computer Engineering: http://www.ece.msstate.edu/courses/ece8053/presentations/07f03-carryskip.ppt

Morris Mano, C. R. (2008). *Carry-Lookahead Adders.* Retrieved January 14, 2011, from Logic & Computer Design Fundamentals: http://writphotec.com/mano4/Supplements/Carrylookahead_supp4.pdf

Parallel Adders. (n.d.). Retrieved September 13, 2012, from http://users.encs.concordia.ca/~asim/COEN_6501/Lecture_Notes/Lecture_2_Slides.pdf

Serial Peripheral Interface (SPI) for FPGA. (n.d.). Retrieved January 20, 2012, from Electronics Bus: http://electronicsbus.com/serial-peripheral-interface-spi-design-fpga-vhdl-verilog/

Shift-and-Add Multiplication. (n.d.). Retrieved April 3, 2012, from Structure of Computer Systems: http://users.utcluj.ro/~baruch/book_ssce/SSCE-Shift-Mult.pdf

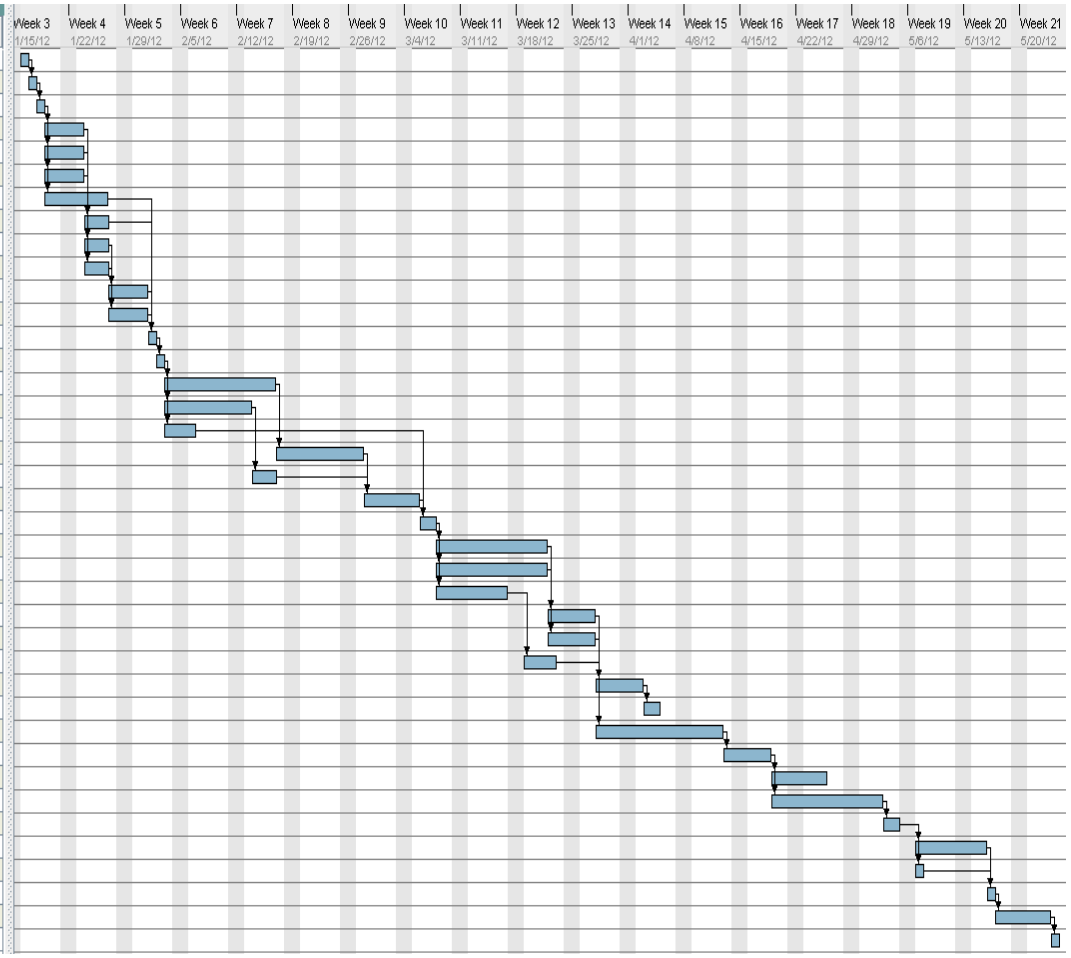
Sorin, D. J. (2009). *Booth's Algorithm.* Retrieved February 7, 2012, from ECE 152 Computer Architecture: http://people.ee.duke.edu/~sorin/prior-courses/ece152-spring2009/lectures/3.3-arith.pdf

Appendix A: Project Planning

This appendix contains the proposed project budget, Gantt chart, and senior project analysis.

Project Gantt Chart

Name	Begin date	End date
Project Proposal	1/16/12	1/16/12
Compose Preliminary Project Specification	1/17/12	1/17/12
Preliminary Project Specification	1/18/12	1/18/12
Background Technology Research	1/19/12	1/23/12
Investigate Solution Alternatives	1/19/12	1/23/12
Partition System Info	1/19/12	1/23/12
System Verification Test Plan	1/19/12	1/26/12
Draft Introduction Section of Report	1/24/12	1/26/12
Select Design Approaches	1/24/12	1/26/12
Develop Subsystem Specifications	1/24/12	1/26/12
Verify Solution Alternative Selections	1/27/12	1/31/12
Draft Specifications Section of Report	1/27/12	1/31/12
Concept Design & Subsystem Verification	2/1/12	2/1/12
Preliminary Design Review	2/2/12	2/2/12
Select Binary Arithmetic Components	2/3/12	2/16/12
Order PMod Units	2/3/12	2/13/12
Subsystem Design Verification Test	2/3/12	2/6/12
Define VHDL Modules	2/17/12	2/27/12
Verfy PMod Functionality	2/14/12	2/16/12
Design Verification	2/28/12	3/5/12
Critical Design Review	3/6/12	3/7/12
Develop VHDL Adder Modules	3/8/12	3/21/12
Develop VHDL Multiplier Modules	3/8/12	3/21/12
Develop VHDL SPI Interface Module	3/8/12	3/16/12
Verify Adder Operation	3/22/12	3/27/12
Verify Multiplier Operation	3/22/12	3/27/12
Verify SPI Interface Module	3/19/12	3/22/12
Arithmetic Component Analysis	3/28/12	4/2/12
Draft Component Analysis Section of Report	4/3/12	4/4/12
Develop VHDL Filter Realization Modules	3/28/12	4/12/12
Verify Filter Realization Modules	4/13/12	4/18/12
Draft Design Section of Report	4/19/12	4/25/12
System Implementation, Debug, & Revision	4/19/12	5/2/12
System Demonstration	5/3/12	5/4/12
Draft Remaining Report Sections	5/7/12	5/15/12
Complete "Analysis of Senior Project Design"	5/7/12	5/7/12
Submit Complete Report Draft	5/16/12	5/16/12
Draft Report Review	5/17/12	5/23/12
Final Project Submission	5/24/12	5/24/12



Proposed Project Budget

Item	Quantity	Cost/Item ($)	Combined Item Cost ($)
Digilent Nexys2 FPGA	1	99.00	99.00
PmodDA2	1	29.99	29.99
PmodAD1	1	34.99	34.99
PmodCON4	2	9.99	19.98
LM324 OP AMP	1	1.05	1.05
10 µF Capacitor	1	0.70	1.40
100 kΩ Resistor	2	0.09	0.18
47 kΩ Resistor	2	0.15	0.30
Total			**186.88**

Students should already have a DIgilent Nexys2 FPGA board, as well as both the ADC and DAC Pmod units, from the digital design series of Cal Poly classes. In this case, the only needed items to complete the project are the PmodCON4 RCA jacks, capacitor, and the resistors for a total of $22.91.

Senior Project Analysis

This appendix provides a project analysis for several project characteristics.

Summary of Functional Requirements

This project implements a modifiable digital filter in the Nexys2 Development Board. This is accomplished using numerous VHDL modules including filter realization structures, ADC and DAC SPI interfaces, a sampling timing interface, and several binary arithmetic components to handle filter calculations. It can filter signals provided with the signal generator or from an audio source, filtering in real time at a variable sampling rate.

Primary Constraints

One of the main difficulties associated with implementing the project is VHDL. A lot of time has passed since taking the digital design series of classes at Cal Poly and remembering how to code in VHDL proved very challenging. After overcoming this hurdle, the next issues arose from the different filter realization modules, which are extremely complex and hard to debug. This is mainly because providing stimuli to the test bench that would simulate the ADC register and an incoming sine wave was extremely difficult and tedious and, ultimately, was not accomplished. Debugging the filter realization modules was done with simpler stimuli that provided various samples, not representing a sine wave, which simulated ADC input to the Nexys2.

Economic

The project budget did not change much from start to finish. The only additional components added were the capacitor, the 4 resistors, and the OP AMP for an additional total of $2.93 above the original estimation. These components were used to create the level shifter circuit for the audio interface.

The time for project completion was underestimated during initial project planning stages. Originally, about 10 weeks were proposed to complete the project. It took almost twice as long to complete, as shown in the Gantt chart in Appendix A.

Environmental

There is not much environmental impact associated with the project. The purpose was not to create something to be manufactured so there is no waste or pollution added from manufacturing. Most of the components used for the project should already be owned by Cal Poly EE/CPE students so there is not any additional waste introduced to the environment from components at the end of their life span. The electricity used to power and operate the Nexys2, computer, and test equipment is one of the few ways the project impacts the environment.

Manufacturability

The project was not designed to be manufactured. If manufacturing the design is desired, the circuit created by the Xilinx tools can be sent to various manufacturers for packaging.

Sustainability

The project was not designed to be sustainable. Sustainable resources are not used to power the Nexys2, computer, or various test equipment used to verify its operation. A power analysis of the system was attempted but no results could be extracted. Power usage could be improved upon, however. This could be accomplished by making more efficient VHDL modules that would perform DSP operations in fewer clock cycles or perform filter calculations using less logic.

Development

During the project I learned about many different things concerning VHDL implementation and the Xilinx ISE tools. The most prominent thing I learned was about the creation of what Xilinx defines as a gated clock. This is a flip-flop that receives its enable input form a gate rather than a clock. A gated clock creates a delay for the enable input, which in turn delays flip flop output, causing many timing problems during Nexys2 implementation.

I also learned about some of the hazards of using 2 process state machines. Careful consideration must be applied when assigning signals using 2 process state machines. If a signal needs be synchronous and it is assigned in the combinatorial process of the state machine then it most likely will generate a gated clock as described above. Gated clock are dangerous because they do not produce logic errors and may not be noticed errors occur during hardware implementation.

Appendix B: Project Hardware & Software Information

This appendix contains information for all hardware and software used to create and test the project.

Table of Hardware Used for Project Development & Testing

Device Name	Purpose
Digilent Nexys2 Spartan-3E	Development board used for VHDL implementation and testing
Digilent Pmod-DA2	Digital to analog converter module
Digilent Pmod-AD1	Analog to digital converter
Digilent Pmod-CON4	RCA audio jacks module
Agilent InfiniiVision MSO-X 3012A	Oscilloscope
Agilent E3630A	Triple output DC power supply

Table of Software Used for Project Development & Testing

Program Name	Purpose
Microsoft Windows XP Professional ver. 2000 SP3	Cal Poly computer operating system
Microsoft Windows 7 Ultimate SP1	Home computer operating system
Microsoft Word 2010	Report development
Microsoft Excel 2010	Frequency response plot development
Xilinx ISE 13.2 ver. 0.61xd	VHDL system used for module design and Nexys2 implementation
MATLAB R2011 ver. 7.13.0.564	Used to create filter frequency response plots and generate filter coefficients
Digilent Adept ver. 2.9.4	Facilitates programming the Nexys2 FPGA board
ToneGen Audio Frequency Generator	Creates test signals for testing the project at home
Soundcard Scope ver. 1.40	Used to verify audio implementation via pc microphone jack

Appendix C: VHDL Module Code

This appendix contains VHDL code for all modules used for the project.

FILTER.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      Filter
-- Module Name:      FILTER - behavioral
-- Project Name:     senior_project
-- Description:      FILTER implements the entire DSP system.
----------------------------------------------------------------------------------
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 FILTER is
   port( CLK      :  in    std_logic;
         RST      :  in    std_logic;
         EN       :  in    std_logic;
         MISO     :  in    std_logic;
         ADC_SCLK :  out   std_logic;
         DAC_SCLK :  out   std_logic;
         CS       :  out   std_logic;
         SYNC     :  out   std_logic;
         MOSI     :  out   std_logic );
end FILTER;

architecture behavioral of FILTER is
-- registers for Ak and Bk coefficients
signal Bs   : vect_16x30 := (others => (others =>'0'));
signal As   : vect_16x20 := (others => (others =>'0'));
-- multiplier inputs
signal MULT_B  : vect_16x20 := (others => (others => '0'));
signal MULT_A  : vect_16x20 := (others => (others =>  '0'));
-- multiplier products
signal P : vect_32x20 := (others => (others => '0'));
-- product register
signal PRODUCT : vect_32x20 := (others => (others => '0'));
-- accumulator input
signal PART_S  : vect_32x20 := (others => (others => '0'));
-- adder inputs
signal ADD_A   : vect_32x20 := (others => (others => '0'));
signal ADD_B   : vect_32x20 := (others => (others => '0'));
-- adder sums
signal S : vect_32x20 := (others => (others => '0'));
-- current input sample
signal Xn   :  std_logic_vector(11 downto 0) := (others => '0');
-- current output
signal Yn   :  std_logic_vector(11 downto 0) := (others => '0');
-- clock signal for debounce circuit
signal DBC_CLK :  std_logic := '0';
-- debounced reset button
signal RST_DB  :  std_logic := '0';
-- debounced enable button
signal EN_DB   :  std_logic := '0';
```

```
-- registers used for debouncing buttons
signal RST_DB_REG :  std_logic_vector(7 downto 0) := (others => '0');
signal EN_DB_REG  :  std_logic_vector(7 downto 0) := (others => '0');
-- register for time delay signals
signal XnmM : vect_16x20 := (others => (others => '0'));
signal YnmN : vect_16x20 := (others => (others => '0'));
-- signals sample acquisition
signal SAMP_DONE  :  std_logic := '0';
-- notifies output is calculated
signal OUTPUT_DONE   :  std_logic := '0';
-- coefficient used for cascade filters
signal C_VECT  :  std_logic_vector(15 downto 0) := (others => '0');

begin
   -- normal direct form I filter realization
   norm  : if STRUCTURE = 0 generate
      n_filter : NORMAL port map( CLK, RST_DB, EN_DB, SAMP_DONE, XnmM, YnmN, As,
                                  Bs, C_VECT, P, S, MULT_B, MULT_A, ADD_A, ADD_B,
                                 PART_S, Yn, OUTPUT_DONE );
   end generate norm;

   -- cascade direct form II filter realization
   casc  : if STRUCTURE = 1 generate
      c_filter : CASCADE port map( CLK, RST_DB, EN_DB, SAMP_DONE, XnmM, YnmN, As,
                                   Bs, C_VECT, P, S, MULT_B, MULT_A, ADD_A, ADD_B,
                                   PART_S, Yn, OUTPUT_DONE );
   end generate casc;

   -- cascade direct form I filter realization
   casc_dir : if STRUCTURE = 2 generate
      nc_filter   : CASC_DFI port map( CLK, RST_DB, EN_DB, SAMP_DONE, XnmM, YnmN,
                                       As, Bs, C_VECT, P, S, MULT_B, MULT_A,
                                       ADD_A, ADD_B, PART_S, Yn, OUTPUT_DONE );
   end generate casc_dir;

   -- binary arithmetic component generation
   bb       : DSP_BB port map( MULT_A, MULT_B, ADD_A, ADD_B, PART_S, P, S, As,
                               Bs, C_VECT );

   -- sampling rate control
   sample   : SAMPLE_CTRL port map( CLK, RST_DB, EN_DB, MISO, Yn, OUTPUT_DONE,
                                    ADC_SCLK, DAC_SCLK, CS, SYNC, MOSI,
                                    Xn, XnmM, YnmN, SAMP_DONE );

   -- debounce circuit clock
   debounce_clk   :  CLK_DIV port map( 1000, CLK, DBC_CLK );

   -- debounce process for button inputs
   debounce_proc : process ( CLK, DBC_CLK ) is
   begin
      if rising_edge(DBC_CLK) then
         RST_DB_REG(7 downto 1)  <= RST_DB_REG(6 downto 0);
         RST_DB_REG(0)  <= RST;

         EN_DB_REG(7 downto 1)   <= EN_DB_REG(6 downto 0);
         EN_DB_REG(0)   <= EN;
      end if;

      if rising_edge(CLK) then
         if EN_DB_REG = "11111111" then
            EN_DB <= '1';
         elsif EN_DB_REG = "00000000" then
            EN_DB <= '0';
         end if;
```

```
         if RST_DB_REG = "11111111" then
            RST_DB   <= '1';
         elsif RST_DB_REG = "00000000" then
            RST_DB   <= '0';
         end if;
      end if;
   end process debounce_proc;
end behavioral;
```

NORMAL.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      Normal Direct Form I FIlter Realization
-- Module Name:      NORMAL - behavioral
-- Project Name:     senior_project
-- Description:      Normal Direct Form I FIlter Realization implements a
--                   digital filter using this filter realization.
----------------------------------------------------------------------------------
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 NORMAL is
   port( CLK         : in  std_logic;
         RST         : in  std_logic;
         EN          : in  std_logic;
         SAMP_DONE   : in  std_logic;
         XnmM        : in  vect_16x20;
         YnmN        : in  vect_16x20;
         As          : in  vect_16x20;
         Bs          : in  vect_16x30;
         C_VECT      : in  std_logic_vector(15 downto 0);
         P           : in vect_32x20;
         S           : in vect_32x20;
         MULT_B      : out vect_16x20;
         MULT_A      : out vect_16x20;
         ADD_A : out vect_32x20;
         ADD_B : out vect_32x20;
         PART_S   : out vect_32x20;
         Yn          : out std_logic_vector(11 downto 0);
         OUTPUT_DONE : out std_logic );
end NORMAL;

architecture behavioral of NORMAL is
-- FSM state declarations
type state_type is (IDLE, WAIT4Xn, MULT, MULT_DUM, GET_P, P_DUM, ACCUM,
                    A_DUM, GET_S, OUT_DUM1, OUT_DUM2, SET_OUTPUT);
signal PS, NS  :  state_type;
-- signals to keep track of multiplying and accumulating filter terms
signal CNT_TERMS     : integer range 0 to 20 := 0;
signal MULT_A_TERMS  : integer range 0 to 20 := 0;
signal MULT_B_TERMS  : integer range 0 to 20 := 0;
-- flag asserted when accumulation is complete
```

```
signal ACCUM_DONE    : std_logic := '0';
-- product register
signal PRODUCT : vect_32x20 := (others => (others => '0'));
-- two signals to add during serial accumulation
signal ADD_A_BUF  : vect_32x20 := (others => (others => '0'));
signal ADD_B_BUF  : vect_32x20 := (others => (others => '0'));
-- sum register
signal SUM     : std_logic_vector(31 downto 0) := (others => '0');
-- register to temporarily hold Yn output
signal Yn_BUF  : std_logic_vector(11 downto 0) := (others => '0');
-- flag to clear registers
signal CLR_REGS   : std_logic := '1';
-- flag to set multiplier inputs
signal LOAD_MULT  : std_logic := '0';
-- flag to load product register
signal LOAD_P  : std_logic := '0';
-- flag to set adder inputs
signal LOAD_ADD   : std_logic := '0';
-- flag to load sum register
signal LOAD_S  : std_logic := '0';
-- flag set when output has been calculated
signal GOT_OUTPUT : std_logic := '0';
-- applies offset of about 1.5 V
constant OFFSET   : std_logic_vector(31 downto 0)
                    := "00000000000000000000011101010011";
-- offset for scaled integers
signal SCALED_OFFSET : std_logic_vector(31 downto 0) := (others => '0');

begin
   -- set scaled offset
   SCALED_OFFSET(11 + SCALE-1 downto SCALE-1)
      <= OFFSET(11 downto 0) when SCALE > 1;
   SCALED_OFFSET(SCALE - 2 downto 0)   <= (others => '0') when SCALE > 1;
   -- assign output Yn to temporary Yn register
   Yn       <= Yn_BUF;
   -- set adder inputs
   ADD_A <= ADD_A_BUF;
   ADD_B <= ADD_B_BUF;

--=============================================================================
-- This process is part of the FSM that controls DSP operation. It synchronizes
-- all outputs from the FSM including the loading of registers and inputs to
-- the various components used.
--=============================================================================
   sync_regs   : process( CLK ) is
   variable TMP_OUT  : std_logic_vector(31 downto 0) := (others => '0');
   begin
      if rising_edge(CLK) then
         -- initialize signals and registers
         if CLR_REGS = '1' then
            MULT_A   <= (others => (others => '0'));
            MULT_B   <= (others => (others => '0'));
            PRODUCT  <= (others => (others => '0'));
            ADD_A_BUF   <= (others => (others => '0'));
            ADD_B_BUF   <= (others => (others => '0'));
            SUM   <= (others => '0');
            Yn_BUF   <= (others => '0');
            CNT_TERMS   <= 0;
            OUTPUT_DONE <= '0';
         end if;

-------------------------------------------------------------------------------
         -- load multiplier inputs
         if LOAD_MULT = '1' then
```

```
-- for serial operation
if SorP = 0 then
   -- give Bk terms to multiplier inputs
   if MULT_B_TERMS < M then
      -- adjust multiplier inputs to only receive positive values
      if XnmM(MULT_B_TERMS) > OFFSET then
         MULT_A(0)   <= XnmM(MULT_B_TERMS) - OFFSET(15 downto 0);
      elsif XnmM(MULT_B_TERMS) <= OFFSET then
         MULT_A(0)   <= OFFSET(15 downto 0) - XnmM(MULT_B_TERMS);
      end if;
      if Bs(MULT_B_TERMS)(15) = '0' then
         MULT_B(0)   <= Bs(MULT_B_TERMS);
      elsif Bs(MULT_B_TERMS)(15) = '1' then
         MULT_B(0)   <= not(Bs(MULT_B_TERMS)) + 1;
      end if;
      -- increment Bk term counter
      MULT_B_TERMS   <= MULT_B_TERMS + 1;
   -- give Ak terms to multiplier inputs
   elsif MULT_A_TERMS < N then
      -- adjust multiplier inputs to only receive positive values
      if YnmN(MULT_A_TERMS) > OFFSET then
         MULT_A(0)   <= YnmN(MULT_A_TERMS) - OFFSET(15 downto 0);
      elsif YnmN(MULT_A_TERMS) <= OFFSET then
         MULT_A(0)   <= OFFSET(15 downto 0) - YnmN(MULT_A_TERMS);
      end if;
      if As(MULT_A_TERMS)(15) = '0' then
         MULT_B(0)   <= As(MULT_A_TERMS);
      elsif As(MULT_A_TERMS)(15) = '1' then
         MULT_B(0)   <= not(As(MULT_A_TERMS)) + 1;
      end if;
      -- increment Ak term counter
      MULT_A_TERMS   <= MULT_A_TERMS + 1;
   end if;
   -- increment filter term counter
   CNT_TERMS   <= CNT_TERMS + 1;
-- for parallel calculations
elsif SorP = 1 then
   -- assign all multiplier inputs
   for i in 0 to F_LENGTH loop
      -- assign the Bk terms to multipliers
      if i < M then
         -- adjust multiplier inputs to only positive values
         -- and remove offset from input sample
         if XnmM(i) > OFFSET then
            MULT_A(i)   <= XnmM(i) - OFFSET(15 downto 0);
         elsif XnmM(i) <= OFFSET then
            MULT_A(i)   <= OFFSET(15 downto 0) - XnmM(i);
         end if;
         if Bs(i)(15) = '0' then
            MULT_B(i)   <= Bs(i);
         elsif Bs(i)(15) = '1' then
            MULT_B(i)   <= not(Bs(i)) + 1;
         end if;
      -- assign the Bk terms to multipliers
      elsif i >= M then
         -- adjust multiplier inputs to only positive values
         -- and remove offset from output signal
         if YnmN(i - M) > OFFSET then
            MULT_A(i)   <= YnmN(i - M) - OFFSET(15 downto 0);
         elsif YnmN(i - M) <= OFFSET then
            MULT_A(i)   <= OFFSET(15 downto 0) - YnmN(i - M);
         end if;
         if As(i - M)(15) = '0' then
            MULT_B(i)   <= As(i - M);
```

```
                  elsif As(i - M)(15) = '1' then
                     MULT_B(i)   <= not(As(i - M)) + 1;
                  end if;
               end if;
            end loop;
         end if;
      end if;

-------------------------------------------------------------------------------
      -- load product register after multiplication is complete
      if LOAD_P = '1' then
         -- reset multiplier inputs
         MULT_A   <= (others => (others => '0'));
         MULT_B   <= (others => (others => '0'));
         -- for serial operation
         if SorP = 0 then
            -- adjust product to reflect multiplication with negative Bk
            if CNT_TERMS <= M then
               if XnmM(MULT_B_TERMS - 1) = "0000000000000000" then
                  PRODUCT(0)  <= (others => '0');
               elsif XnmM(MULT_B_TERMS - 1) > OFFSET then
                  if Bs(MULT_B_TERMS - 1)(15) = '0' then
                     PRODUCT(0)  <= P(0);
                  elsif BS(MULT_B_TERMS - 1)(15) = '1' then
                     PRODUCT(0)  <= not(P(0)) + 1;
                  end if;
               elsif XnmM(MULT_B_TERMS - 1) <= OFFSET then
                  if Bs(MULT_B_TERMS - 1)(15) = '0' then
                     PRODUCT(0)  <= not(P(0)) + 1;
                  elsif BS(MULT_B_TERMS - 1)(15) = '1' then
                     PRODUCT(0)  <= P(0);
                  end if;
               end if;
            -- adjust product to reflect multiplication with negative Ak
            elsif CNT_TERMS > M then
               if YnmN(MULT_A_TERMS - 1) = "0000000000000000" then
                  PRODUCT(0)  <= (others => '0');
               elsif YnmN(MULT_A_TERMS - 1) > OFFSET then
                  if As(MULT_A_TERMS - 1)(15) = '0' then
                     PRODUCT(0)  <= P(0);
                  elsif As(MULT_A_TERMS - 1)(15) = '1' then
                     PRODUCT(0)  <= not(P(0)) + 1;
                  end if;
               elsif YnmN(MULT_A_TERMS - 1) <= OFFSET then
                  if As(MULT_A_TERMS - 1)(15) = '0' then
                     PRODUCT(0)  <= not(P(0)) + 1;
                  elsif As(MULT_A_TERMS - 1)(15) = '1' then
                     PRODUCT(0)  <= P(0);
                  end if;
               end if;
            end if;
         -- for parallel operation
         elsif SorP = 1 then
            for i in 0 to F_LENGTH loop
               -- adjust product to reflect multiplication with negative Bk
               if i < M then
                  if XnmM(i) = "0000000000000000" then
                     PRODUCT(i)  <= (others => '0');
                  elsif XnmM(i) > OFFSET then
                     if Bs(i)(15) = '0' then
                        PRODUCT(i)  <= P(i);
                     elsif BS(i)(15) = '1' then
                        PRODUCT(i)  <= not(P(i)) + 1;
                     end if;
```

```
                  elsif XnmM(i) <= OFFSET then
                     if Bs(i)(15) = '0' then
                        PRODUCT(i)  <= not(P(i)) + 1;
                     elsif BS(i)(15) = '1' then
                        PRODUCT(i)  <= P(i);
                     end if;
                  end if;
               -- adjust product to reflect multiplication with negative Ak
               elsif i >= M then
                  if YnmN(i - M) = "0000000000000000" then
                     PRODUCT(i)  <= (others => '0');
                  elsif YnmN(i - M) > OFFSET then
                     if As(i - M)(15) = '0' then
                        PRODUCT(i)  <= P(i);
                     elsif As(i - M)(15) = '1' then
                        PRODUCT(i)  <= not(P(i)) + 1;
                     end if;
                  elsif YnmN(i - M) <= OFFSET then
                     if As(i - M)(15) = '0' then
                        PRODUCT(i)  <= not(P(i)) + 1;
                     elsif As(i - M)(15) = '1' then
                        PRODUCT(i)  <= P(i);
                     end if;
                  end if;
               end if;
            end loop;
         end if;
      end if;

-------------------------------------------------------------------------------
      -- load adder inputs
      if LOAD_ADD = '1' then
         -- for serial operation
         if SorP = 0 then
            -- load current value of accumulated sum
            ADD_A_BUF(0)   <= SUM;
            -- load current term to accululate
            ADD_B_BUF(0)   <= PRODUCT(0);
         -- for parallel operation
         elsif SorP = 1 then
            -- load accumulation register with all appropriate terms
            PART_S(0 to F_LENGTH)   <= PRODUCT(0 to F_LENGTH);
         end if;
      end if;

-------------------------------------------------------------------------------
      -- load sum register
      if LOAD_S = '1' then
         -- reset adder inputs
         ADD_A_BUF   <= (others => (others => '0'));
         ADD_B_BUF   <= (others => (others => '0'));
         PART_S   <= (others => (others => '0'));
         -- for serial operation
         if SorP = 0 then
            -- continue accumulation if not done
            if CNT_TERMS <= F_LENGTH then
               SUM   <= S(0);
            -- account for upper and lower bound saturation of output
            -- when all terms have been accumulated
            elsif CNT_TERMS > F_LENGTH then
               -- reset Ak and Bk term counters
               MULT_B_TERMS   <= 0;
               MULT_A_TERMS   <= 0;
               -- assign output for scaled coefficients
```

```
                if SCALE > 1 then
                   -- add 1.5 V scaled offset to output
                   TMP_OUT  := S(0) + SCALED_OFFSET;
                   if TMP_OUT(31) = '0' then
                      if TMP_OUT = TMP_OUT(11 + SCALE - 1 downto 0) then
                         Yn_BUF   <= TMP_OUT(11 + SCALE - 1 downto SCALE - 1);
                      elsif TMP_OUT > TMP_OUT(11 + SCALE - 1 downto 0) then
                         Yn_BUF   <= (others => '1');
                      end if;
                   elsif TMP_OUT(31) = '1' then
                      Yn_BUF   <= (others => '0');
                   end if;
                -- assign output for unscaled coefficients
                elsif SCALE = 1 then
                   -- add 1.5 V offset to output
                   TMP_OUT  := S(0) + OFFSET;
                   if TMP_OUT(31) = '0' then
                      if TMP_OUT = TMP_OUT(11 downto 0) then
                         Yn_BUF   <= TMP_OUT(11 downto 0);
                      elsif TMP_OUT > TMP_OUT(11 downto 0) then
                         Yn_BUF   <= (others => '1');
                      end if;
                   elsif TMP_OUT(31) = '1' then
                      Yn_BUF   <= (others => '0');
                   end if;
                end if;
             end if;
          -- for parallel operation
          elsif SorP = 1 then
             -- assign output for scaled coefficients
             if SCALE > 1 then
                -- add 1.5 V scaled offset to output
                TMP_OUT  := S(0) + SCALED_OFFSET;
                if TMP_OUT(31) = '0' then
                   if TMP_OUT = TMP_OUT(11 + SCALE - 1 downto 0) then
                      Yn_BUF   <= TMP_OUT(11 + SCALE - 1 downto SCALE - 1);
                   elsif TMP_OUT > TMP_OUT(11 + SCALE - 1 downto 0) then
                      Yn_BUF   <= (others => '1');
                   end if;
                elsif TMP_OUT(31) = '1' then
                   Yn_BUF   <= (others => '0');
                end if;
             -- assign output for unscaled coefficients
             elsif SCALE = 1 then
                -- add 1.5 V offset to output
                TMP_OUT  := S(0) + OFFSET;
                if TMP_OUT(31) = '0' then
                   if TMP_OUT = TMP_OUT(11 downto 0) then
                      Yn_BUF   <= TMP_OUT(11 downto 0);
                   elsif TMP_OUT > TMP_OUT(11 downto 0) then
                      Yn_BUF   <= (others => '1');
                   end if;
                elsif TMP_OUT(31) = '1' then
                   Yn_BUF   <= (others => '0');
                end if;
             end if;
          end if;
       end if;

-------------------------------------------------------------------------------
       -- set flag to signal output has been calculated
       if GOT_OUTPUT = '1' then
          OUTPUT_DONE <= '1';
       end if;
```

```
      end if;
   end process sync_regs;

--=============================================================================
-- This process synchronizes state changes for the FSM.
--=============================================================================
   sync_proc   : process( NS, CLK, RST ) is
   begin
      -- asynchronous reset
      if RST = '1' then
         PS <= IDLE;
      elsif rising_edge(CLK) then
         PS <= NS;
      end if;
   end process sync_proc;

--=============================================================================
-- This process contains the combinatorial logic used for state changes of the
-- FSM and the various other operations carried out during DSP calculations.
--=============================================================================
   comb_proc   : process( PS, EN, SAMP_DONE, CNT_TERMS ) is
   begin
      case PS is
      -- state, idle/initialization
      when IDLE =>
         -- set flag to initialize registers and signals
         CLR_REGS <= '1';
         -- initialize FSM flags
         LOAD_MULT   <= '0';
         LOAD_P   <= '0';
         LOAD_ADD <= '0';
         LOAD_S   <= '0';
         GOT_OUTPUT  <= '0';
         -- change state to wait for a sample when system enable is received
         if EN = '1' then
            NS <= WAIT4Xn;
         elsif EN = '1' then
            NS <= IDLE;
         end if;

      -- state, wait for sample from ADC
      when WAIT4Xn =>
         -- clear registers and signals
         CLR_REGS <= '1';
         -- assign remaining flags of FSM
         LOAD_MULT   <= '0';
         LOAD_P   <= '0';
         LOAD_ADD <= '0';
         LOAD_S   <= '0';
         GOT_OUTPUT  <= '0';
         -- cange to the multiply state after sample acquisition
         if SAMP_DONE = '0' then
            NS <= WAIT4Xn;
         elsif SAMP_DONE = '1' then
            NS <= MULT;
         end if;

      -- state, multiply filter terms
      when MULT =>
         -- set flag to load multiplier inputs
         LOAD_MULT   <= '1';
         -- assign remaining flags of FSM
         CLR_REGS <= '0';
         LOAD_P   <= '0';
```

```
   LOAD_ADD <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- go to state that loads product register
   NS <= MULT_DUM;

-- state, dummy for MULT state
when MULT_DUM =>
   LOAD_MULT   <= '0';
   CLR_REGS <= '0';
   LOAD_P   <= '0';
   LOAD_ADD <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   NS <= GET_P;

-- state, load product register
when GET_P =>
   -- set flag to load product register
   LOAD_P   <= '1';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_ADD <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- go to state that accumulates filter terms
   NS <= P_DUM;

-- state, dummy for GET_P
when P_DUM =>
   -- set flag to load product register
   LOAD_P   <= '0';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_ADD <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- go to state that accumulates filter terms
   NS <= ACCUM;

-- state, accumulate filter terms
when ACCUM =>
   -- set flag to load adder inputs
   LOAD_ADD <= '1';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- for serial operation go to the state that loads the
   -- accumulation register
   if SorP = 0 then
      NS <= GET_S;
   -- for parallel operation go to the dummy state needed for
   -- calculations made with the 32-bit Carry-Save accumulator
   elsif SorP = 1 then
      NS <= A_DUM;
   else
      NS <= ACCUM;
   end if;
```

```
-- state, dummy state needed to complete calculations
-- accomplished by the 32-bit Carry-Save accumulator
when A_DUM =>
   -- set flag to load adder inputs
   LOAD_ADD <= '0';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- go to the state that loads the accumulation register
   NS <= GET_S;

-- state, loads accumulation register
when GET_S =>
   -- set flag that loads the accumulation register
   LOAD_S   <= '1';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_ADD <= '0';
   GOT_OUTPUT  <= '0';
   -- for serial operation
   if SorP = 0 then
      -- verfy all terms have been accumulated
      if CNT_TERMS <= F_LENGTH then
         -- go to state to multiply next term if terms remain
         NS <= MULT;
      elsif CNT_TERMS > F_LENGTH then
         -- go to state that signals output has been calculated
         NS <= OUT_DUM1;
      end if;
   -- for parallel operation
   elsif SorP = 1 then
      -- go to state that signals output has been calculated
      NS <= OUT_DUM1;
   end if;

-- state, first dummy for output
when OUT_DUM1 =>
   LOAD_ADD <= '0';
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   NS <= OUT_DUM2;

-- state, second dummy for output
when OUT_DUM2 =>
   LOAD_ADD <= '0';
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   NS <= SET_OUTPUT;

-- state, signals that the final output value has been set
when SET_OUTPUT =>
   -- flag to sync completion flag, OUTPUT_DONE
   GOT_OUTPUT  <= '1';
```

```
         -- assign remaining flags of FSM
         CLR_REGS <= '0';
         LOAD_MULT   <= '0';
         LOAD_P   <= '0';
         LOAD_ADD <= '0';
         LOAD_S   <= '0';
         -- go to state to wait for next sample from ADC
         NS <= WAIT4Xn;

      when others =>
         NS <= IDLE;

      end case;
   end process comb_proc;
end behavioral;
```

CASCADE.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      Cascade Direct Form II FIlter Realization
-- Module Name:      CASCADE - behavioral
-- Project Name:     senior_project
-- Description:      Cascade Direct Form II FIlter Realization implements a
--                   digital filter using this filter realization.
----------------------------------------------------------------------------------
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 CASCADE is
   port( CLK         : in  std_logic;
         RST         : in  std_logic;
         EN          : in  std_logic;
         SAMP_DONE   : in  std_logic;
         XnmM        : in  vect_16x20;
         YnmN        : in  vect_16x20;
         As          : in  vect_16x20;
         Bs          : in  vect_16x30;
         C_VECT      : in  std_logic_vector(15 downto 0);
         P           : in vect_32x20;
         S           : in vect_32x20;
         MULT_B      : out vect_16x20;
         MULT_A      : out vect_16x20;
         ADD_A : out vect_32x20;
         ADD_B : out vect_32x20;
         PART_S   : out vect_32x20;
         Yn          : out std_logic_vector(11 downto 0);
         OUTPUT_DONE : out std_logic );
end CASCADE;

architecture behavioral of CASCADE is
type state_type is (IDLE, WAIT4Xn, MULT, MULT_DUM, GET_P, P_DUM, ACCUM,
                    A_DUM, GET_S, OUT_DUM1, OUT_DUM2, SET_OUTPUT);
signal PS, NS  :  state_type;
-- signals to keep track of multiplying and accumulating filter terms
```

```
signal CNT_TERMS     : integer range 0 to 6 := 0;
signal MULT_A_TERMS  : integer range 0 to 2 := 0;
signal MULT_B_TERMS  : integer range 0 to 3 := 0;
signal NUM_A_TERMS   : integer range 0 to 20 := 0;
signal NUM_B_TERMS   : integer range 0 to 30 := 0;
-- signal to keep track of filter stages
signal CNT_STAGE     : integer range 0 to 3 := 0;
-- flag asserted when accumulation is complete
signal ACCUM_DONE : std_logic := '0';
-- signals that hold multipliers and multiplicands
signal MULT_A_BUF : vect_16x20 := (others => (others => '0'));
signal MULT_B_BUF : vect_16x20 := (others => (others => '0'));
-- product register
signal PRODUCT : vect_32x20 := (others => (others => '0'));
-- buffers for adder inputs
signal ADD_A_BUF  : vect_32x20 := (others => (others => '0'));
signal ADD_B_BUF  : vect_32x20 := (others => (others => '0'));
-- sum register
signal SUM  : vect_32x20 := (others => (others => '0'));
-- register to temporarily hold Yn output
signal Yn_BUF  : std_logic_vector(11 downto 0) := (others => '0');
-- current intermediary signal for cascading filter stages
signal Wn   : vect_16x20 := (others => (others => '0'));
signal Yi   : vect_16x20 := (others => (others => '0'));
-- flag to signal Wn has been calculated
signal Wn_DONE : std_logic := '0';
-- array to hold shifted intermediary signals for cascading filter stages
signal Wi   : vect_16x30 := (others => ( others => '0'));
-- flag to clear registers
signal CLR_REGS   : std_logic := '1';
-- flag to set multiplier inputs
signal LOAD_MULT  : std_logic := '0';
-- flag to load product register
signal LOAD_P  : std_logic := '0';
-- flag to set adder inputs
signal LOAD_ADD   : std_logic := '0';
-- flag to load sum register
signal LOAD_S  : std_logic := '0';
-- flag set when output has been calculated
signal GOT_OUTPUT : std_logic := '0';
signal AB_TOG  : std_logic := '0';
-- used to accumulate terms for parallel operation
signal ACCUM_TOG  : std_logic := '0';
-- flag to accumulate output terms for parallel operation
signal ACCUM_OUTPUT  : std_logic := '0';
-- applies offset of about 1.5 V
constant OFFSET   : std_logic_vector(31 downto 0)
                    := "00000000000000000000011101010011";
-- offset for scaled integers
signal SCALED_OFFSET : std_logic_vector(31 downto 0) := (others => '0');

begin
   -- set scaled offset
   SCALED_OFFSET(11 + SCALE-1 downto SCALE-1)
      <= OFFSET(11 downto 0) when SCALE > 1;
   SCALED_OFFSET(SCALE - 2 downto 0)   <= (others => '0') when SCALE > 1;

   -- assign adder and multiplier inputs from buffers
   MULT_A   <= MULT_A_BUF;
   MULT_B   <= MULT_B_BUF;
   ADD_A <= ADD_A_BUF;
   ADD_B <= ADD_B_BUF;

   -- assign output Yn to from buffer
```

```
   Yn       <= Yn_BUF;

   -- determine number of Ak and Bk terms from filter stages
   NUM_A_TERMS <= F_STAGES * 2;
   NUM_B_TERMS <= F_STAGES * 3;

--=============================================================================
-- This process is a SIPO shift register for Wn.
--=============================================================================
   shift_w  : process( Wn_DONE ) is
   begin
      if rising_edge(Wn_DONE) then
         if SorP = 0 then
            if CNT_STAGE = 0 then
               Wi(1 to 2)  <= Wi(0 to 1);
               Wi(0) <= Wn(0);
            elsif CNT_STAGE = 1 then
               Wi(4 to 5)  <= Wi(3 to 4);
               Wi(3) <= Wn(0);
            elsif CNT_STAGE = 2 then
               Wi(7 to 8)  <= Wi(6 to 7);
               Wi(6) <= Wn(0);
            elsif CNT_STAGE = 3 then
               Wi(10 to 11)   <= Wi(9 to 10);
               Wi(9) <= Wn(0);
            elsif CNT_STAGE = 4 then
               Wi(13 to 14)   <= Wi(12 to 13);
               Wi(12)   <= Wn(0);
            elsif CNT_STAGE = 5 then
               Wi(16 to 17)   <= Wi(15 to 16);
               Wi(15)   <= Wn(0);
            elsif CNT_STAGE = 6 then
               Wi(19 to 20)   <= Wi(18 to 19);
               Wi(18)   <= Wn(0);
            elsif CNT_STAGE = 7 then
               Wi(22 to 23)   <= Wi(21 to 22);
               Wi(21)   <= Wn(0);
            elsif CNT_STAGE = 8 then
               Wi(25 to 26)   <= Wi(24 to 25);
               Wi(24)   <= Wn(0);
            end if;
         elsif SorP = 1 then
            for i in 0 to F_STAGES - 1 loop
               Wi(i * 3 + 1 to i * 3 + 2) <= Wi(i * 3 to i * 3 + 1);
               Wi(i * 3)   <= Wn(i);
            end loop;
         end if;
      end if;
   end process shift_w;

--=============================================================================
-- This process is part of the FSM that controls DSP operation. It synchronizes
-- all outputs from the FSM including the loading of registers and inputs to
-- the various components used.
--=============================================================================
   sync_regs   : process( CLK ) is
   variable TMP_OUT  : std_logic_vector(31 downto 0)  := (others => '0');
   begin
      if rising_edge(CLK) then
         -- initialize signals and registers
         if CLR_REGS = '1' then
            MULT_A_BUF  <= (others => (others => '0'));
            MULT_B_BUF  <= (others => (others => '0'));
            PRODUCT  <= (others => (others => '0'));
```

```
            ADD_A_BUF   <= (others => (others => '0'));
            ADD_B_BUF   <= (others => (others => '0'));
            SUM   <= (others => (others => '0'));
            Yn_BUF   <= (others => '0');
            Yi <= (others => (others => '0'));
            Wn <= (others => (others => '0'));
            AB_TOG   <= '0';
            ACCUM_TOG   <= '0';
            ACCUM_OUTPUT   <= '0';
            CNT_TERMS   <= 0;
            OUTPUT_DONE <= '0';
            Wn_DONE  <= '0';
            CNT_STAGE   <= 0;
            MULT_A_TERMS   <= 0;
            MULT_B_TERMS   <= 0;
         end if;

-------------------------------------------------------------------------------
         -- load multiplier inputs
         if LOAD_MULT = '1' then
            -- for serial operation
            if SorP = 0 then
               -- check if multiplying Ak or Bk terms
               if AB_TOG = '0' then
                  -- give Bk terms to multiplier inputs
                  if MULT_A_TERMS < NUM_A_TERMS then
                     -- adjust multiplier inputs to only receive positive values
                     if Wi(MULT_A_TERMS + CNT_STAGE)(15) = '1' then
                        MULT_A_BUF(0)  <= not(Wi(MULT_A_TERMS + CNT_STAGE)) + 1;
                     elsif Wi(MULT_A_TERMS + CNT_STAGE)(15) = '0' then
                        MULT_A_BUF(0)  <= Wi(MULT_A_TERMS + CNT_STAGE);
                     end if;
                     if As(MULT_A_TERMS)(15) = '1' then
                        MULT_B_BUF(0)  <= not(As(MULT_A_TERMS)) + 1;
                     elsif Bs(MULT_A_TERMS)(15) = '0' then
                        MULT_B_BUF(0)  <= As(MULT_A_TERMS);
                     end if;
                     -- increment Bk term counter
                     MULT_A_TERMS   <= MULT_A_TERMS + 1;
                  end if;
               elsif AB_TOG = '1' then
                  -- give Ak terms to multiplier inputs
                  if MULT_B_TERMS < NUM_B_TERMS then
                     -- adjust multiplier inputs to only receive positive values
                     if Wi(MULT_B_TERMS)(15) = '1' then
                        MULT_A_BUF(0)  <= not(Wi(MULT_B_TERMS)) + 1;
                     elsif Wi(MULT_B_TERMS)(15) = '0' then
                        MULT_A_BUF(0)  <= Wi(MULT_B_TERMS);
                     end if;
                     if Bs(MULT_B_TERMS)(15) = '1' then
                        MULT_B_BUF(0)  <= not(Bs(MULT_B_TERMS)) + 1;
                     elsif Bs(MULT_B_TERMS)(15) = '0' then
                        MULT_B_BUF(0)  <= Bs(MULT_B_TERMS);
                     end if;
                     -- increment Ak term counter
                     MULT_B_TERMS   <= MULT_B_TERMS + 1;
                  end if;
               end if;
               -- increment filter term counter
               CNT_TERMS   <= CNT_TERMS + 1;
            -- for parallel implementation
            elsif SorP = 1 then
               -- assign multiplier input for all filter stages
               for i in 0 to F_STAGES - 1 loop
```

```
                for j in 0 to 1 loop
                   -- check to see if assigning Ak or Bk coefficients
                   if AB_TOG = '0' then
                      -- adjust multiplier inputs to only positive values
                      if Wi(j + 3 * i)(15) = '1' then
                         MULT_A_BUF(j + 2 * i)   <= not(Wi(j + 3 * i)) + 1;
                      elsif Wi(j + 3 * i)(15) = '0' then
                         MULT_A_BUF(j + 2 * i)   <= Wi(j + 3 * i);
                      end if;
                      if As(j + 2 * i)(15) = '1' then
                         MULT_B_BUF(j + 2 * i)   <= not(As(j + 2 * i)) + 1;
                      elsif As(j + 2 * i)(15) = '0' then
                         MULT_B_BUF(j + 2 * i)   <= As(j + 2 * i);
                      end if;
                   elsif AB_TOG = '1' then
                      -- adjust multiplier inputs to only positive values
                      if Wi(j + 3 * i)(15) = '1' then
                         MULT_A_BUF(j + 2 * i)   <= not(Wi(j + 3 * i)) + 1;
                      elsif Wi(j + 3 * i)(15) = '0' then
                         MULT_A_BUF(j + 2 * i)   <= Wi(j + 3 * i);
                      end if;
                      if Bs(j + 2 * i)(15) = '1' then
                         MULT_B_BUF(j + 2 * i)   <= not(Bs(j + 2 * i)) + 1;
                      elsif Bs(j + 2 * i)(15) = '0' then
                         MULT_B_BUF(j + 2 * i)   <= Bs(j + 2 * i);
                      end if;
                      -- assign multiplier inputs to calculate Cx[n]
                      if (i = F_STAGES - 1) and (j = 1) then
                         if XnmM(0) > OFFSET then
                            MULT_A_BUF(j + 2 * i + 1)
                               <= XnmM(0) - OFFSET(15 downto 0);
                         elsif XnmM(0) <= OFFSET then
                            MULT_A_BUF(j + 2 * i + 1)
                               <= OFFSET(15 downto 0) - XnmM(0);
                         end if;
                         if C_VECT(15) = '1' then
                            MULT_B_BUF(j + 2 * i + 1)  <= not(C_VECT) + 1;
                         elsif C_VECT(15) = '0' then
                            MULT_B_BUF(j + 2 * i + 1)  <= C_VECT;
                         end if;
                      end if;
                   end if;
                end loop;
             end loop;
          end if;
       end if;

-------------------------------------------------------------------------------
       -- load product register after multiplication is complete
       if LOAD_P = '1' then
          -- reset multiplier inputs
          MULT_A_BUF  <= (others => (others => '0'));
          MULT_B_BUF  <= (others => (others => '0'));
          -- for serial operation
          if SorP = 0 then
             -- calculating Wi
             if CNT_TERMS < 3 then
                -- Adjust product to be positive or negative depending on
                -- multiplier input
                if (As(MULT_A_TERMS - 1)(15) xor
                    Wi(MULT_A_TERMS + CNT_STAGE - 1)(15)) = '1' then
                   PRODUCT(CNT_TERMS - 1)  <= not(P(0)) + 1;
                elsif ((As(MULT_A_TERMS - 1)(15) and
                        Wi(MULT_A_TERMS + CNT_STAGE - 1)(15)) xor
```

```
                    (As(MULT_A_TERMS - 1)(15) nor
                     Wi(MULT_A_TERMS + CNT_STAGE - 1)(15))) = '1' then
                  PRODUCT(CNT_TERMS - 1)  <= P(0);
               end if;
            -- calculating Yi
            elsif CNT_TERMS >= 3 then
               -- Adjust product to be positive or negative depending on
               -- multiplier input
               if (Bs(MULT_B_TERMS - 1)(15) xor
                   Wi(MULT_B_TERMS - 1)(15)) = '1' then
                  PRODUCT(CNT_TERMS - 1)  <= not(P(0)) + 1;
               elsif ((Bs(MULT_B_TERMS - 1)(15) and
                       Wi(MULT_B_TERMS - 1)(15)) xor
                      (Bs(MULT_B_TERMS - 1)(15) nor
                       Wi(MULT_B_TERMS - 1)(15))) = '1' then
                  PRODUCT(CNT_TERMS - 1)  <= P(0);
               end if;
            end if;
         -- for parallel operation
         elsif SorP = 1 then
            -- load
            for i in 0 to F_STAGES - 1 loop
               for j in 0 to 1 loop
                  -- adjust product for the calculation of Wi
                  if AB_TOG = '0' then
                     if (As(j + 2 * i)(15) xor Wi(j + 3 * i + 1)(15))
                        = '1' then
                        PRODUCT(j + 2 * i)   <= not(P(j + 2 * i)) + 1;
                     elsif ((As(j + 2 * i)(15) and Wi(j + 3* i + 1)(15)) xor
                            (As(j + 2 * i)(15) nor Wi(j + 3 * i + 1)(15)))
                              = '1' then
                        PRODUCT(j + 2 * i)   <= P(j + 2 * i);
                     end if;
                  -- adjust product for the calculation of Yi
                  elsif AB_TOG = '1' then
                     if (Bs(j + 2 * i)(15) xor Wi(j + 3 * i)(15)) = '1' then
                        PRODUCT(j + 2 * i)   <= not(P(j + 2 * i)) + 1;
                     elsif ((Bs(j + 2 * i)(15) and Wi(j + 3 * i)(15)) xor
                            (Bs(j + 2 * i)(15) nor Wi(j + 3 * i)(15))) = '1' then
                        PRODUCT(j + 2 * i)   <= P(j + 2 * i);
                     end if;
                     -- adjust product for calculating Cx[n]
                     if (i = F_STAGES - 1) and (j = 1) then
                        if XnmM(0) > OFFSET then
                           if MULT_B_BUF(j + 2 * i + 1)(15) = '0' then
                              PRODUCT(j + 2 * i + 1)  <= P(j + 2 * i + 1);
                           elsif MULT_B_BUF(j + 2 * i + 1)(15) = '1' then
                              PRODUCT(j + 2 * i + 1)  <= not(P(j + 2 * i + 1)) + 1;
                           end if;
                        elsif XnmM(0) <= OFFSET then
                           if MULT_B_BUF(j + 2 * i + 1)(15) = '0' then
                              PRODUCT(j + 2 * i + 1)  <= not(P(j + 2 * i + 1)) + 1;
                           elsif MULT_B_BUF(j + 2 * i + 1)(15) = '1' then
                              PRODUCT(j + 2 * i + 1)  <= P(j + 2 * i + 1);
                           end if;
                        end if;
                     end if;
                  end if;
               end loop;
            end loop;
         end if;
      end if;

--------------------------------------------------------------------------------
```

```
         -- load adder inputs
         if LOAD_ADD = '1' then
            -- for serial operation
            if SorP = 0 then
               -- acumulate terms for Wi
               if CNT_TERMS = 2 then
                  if ACCUM_TOG = '0' then
                     ADD_A_BUF(0)   <= PRODUCT(0);
                     ADD_B_BUF(0)   <= PRODUCT(1);
                  elsif ACCUM_TOG = '1' then
                     if CNT_STAGE = 0 then
                        -- remove offset from input sample
                        ADD_A_BUF(0)
                           <= ("0000000000000000" & XnmM(0)) - OFFSET;
                     elsif CNT_STAGE > 0 then
                        ADD_A_BUF(0)(15 downto 0)  <= Yi(0);
                        if Yi(0)(15) = '0' then
                           ADD_A_BUF(0)(31 downto 16) <= (others => '0');
                        elsif Yi(0)(15) = '1' then
                           ADD_A_BUF(0)(31 downto 16) <= (others => '1');
                        end if;
                     end if;
                     ADD_B_BUF(0)   <= SUM(0);
                  end if;
               -- accumulate terms for Yi
               elsif CNT_TERMS > 2 then
                  if ACCUM_TOG = '0' then
                     ADD_A_BUF(0)   <= PRODUCT(2);
                     ADD_B_BUF(0)   <= PRODUCT(3);
                  elsif ACCUM_TOG = '1' then
                     ADD_A_BUF(0)   <= PRODUCT(4);
                     ADD_B_BUF(0)   <= SUM(0);
                  end if;
               end if;
            -- for parallel operation
            elsif SorP = 1 then
               for i in 0 to F_STAGES - 1 loop
                  -- accumulate terms for Wi
                  if ACCUM_OUTPUT = '0' then
                     if AB_TOG = '0' then
                        if ACCUM_TOG = '0' then
                           ADD_A_BUF(i)   <= PRODUCT(i + i);
                           ADD_B_BUF(i) <= PRODUCT(i + i + 1);
                        elsif ACCUM_TOG = '1' then
                           -- remove offset from input sample
                           ADD_A_BUF(i)(15 downto 0)
                              <= ("0000000000000000" & XnmM(0)) - OFFSET;
                           ADD_B_BUF(i)   <= SUM(i);
                        end if;
                     -- accumulate terms
                     elsif AB_TOG = '1' then
                        ADD_A_BUF(i)   <= PRODUCT(i + i);
                        ADD_B_BUF(i) <= PRODUCT(i + i + 1);
                     end if;
                  elsif ACCUM_OUTPUT = '1' then
                     PART_S(i)(15 downto 0)  <= Yi(i);
                     PART_S(F_STAGES)  <= PRODUCT(F_STAGES * 2);
                  end if;
               end loop;
            end if;
         end if;

-------------------------------------------------------------------------------
         -- load sum register
```

```
if LOAD_S = '1' then
   -- reset adder inputs
   ADD_A_BUF   <= (others => (others => '0'));
   ADD_B_BUF   <= (others => (others => '0'));
   PART_S   <= (others => (others => '0'));
   -- for serial operation
   if SorP = 0 then
      -- accumulating Wi terms
      if CNT_TERMS = 2 then
         -- accumulating Ak terms of Wi
         if ACCUM_TOG = '0' then
            -- change to add stage input to Wi
            ACCUM_TOG   <= '1';
            -- for integer scaling
            if SCALE > 1 then
               -- check if current acumulated value for Wi is positive
               if S(0)(31) = '0' then
                  -- descale sum before placing in register
                  if S(0) = S(0)(14 + SCALE downto 0) then
                     SUM(0)(15 downto 0)
                        <= S(0)(15 + SCALE - 1 downto SCALE - 1);
                     SUM(0)(15)  <= '0';
                  -- adjust sum for upper saturation
                  elsif S(0) > S(0)(14 + SCALE downto 0) then
                     SUM(0)(14 downto 0)  <= (others => '1');
                     SUM(0)(31 downto 15) <= (others => '0');
                  end if;
               -- check if current acumulated value for Wi is negative
               elsif S(0)(31) = '1' then
                  -- descale sum before placing in register
                  if to_integer(unsigned(S(0)(31 downto 15 + SCALE) + 1))
                     = 0 then
                     SUM(0)(15 downto 0)
                        <= S(0)(15 + SCALE - 1 downto SCALE - 1);
                     SUM(0)(31 downto 15 + SCALE)  <= (others => '1');
                  -- adjust sum for lover saturation
                  elsif to_integer(unsigned(S(0)(31 downto 15 + SCALE) + 1))
                     /= 0 then
                     SUM(0)(14 downto 0)  <= (others => '0');
                     SUM(0)(31 downto 15) <= (others => '1');
                  end if;
               end if;
            -- for no integer scaling
            elsif SCALE = 1 then
               -- check if current sum is positive
               if S(0)(31) = '0' then
                  -- store sum in register if good value
                  if S(0) = S(0)(14 downto 0) then
                     SUM(0)(15 downto 0)  <= S(0)(15 downto 0);
                  -- adjust sum for upper saturation
                  elsif S(0) > S(0)(14 downto 0) then
                     SUM(0)(14 downto 0)  <= (others => '1');
                     SUM(0)(31 downto 15) <= (others => '0');
                  end if;
               -- check if current sum is negative
               elsif S(0)(31) = '1' then
                  -- store sum in register if good value
                  if to_integer(unsigned(S(0)(31 downto 15) + 1))
                     = 0 then
                     SUM(0)(15 downto 0)  <= S(0)(15 downto 0);
                  -- adjust sum for lower saturation
                  elsif to_integer(unsigned(S(0)(31 downto 15) + 1))
                     /= 0 then
                     SUM(0)(14 downto 0)  <= (others => '0');
```

```
                SUM(0)(31 downto 15) <= (others => '1');
             end if;
          end if;
       end if;
    -- done calculating Wi
    elsif ACCUM_TOG = '1' then
       -- toggle acumulation toggle signals for next accumulation
       ACCUM_TOG   <= '0';
       AB_TOG   <= '1';
       -- set Wn_DONE flag to notify conpletion of Wi calculation
       Wn_DONE  <= '1';
       -- check if current sum is positive
       if S(0)(15) = '0' then
          -- set Wn if good sum value
          if ((ADD_A_BUF(0)(15) nor ADD_B_BUF(0)(15)) xor
              (ADD_A_BUF(0)(15) xor ADD_B_BUF(0)(15))) = '1' then
             Wn(0) <= S(0)(15 downto 0);
          -- adjust Wn for upper saturation
          elsif (ADD_A_BUF(0)(15) and ADD_B_BUF(0)(15)) = '1' then
             Wn(0)(14 downto 0)   <= (others => '0');
             Wn(0)(15)   <= '1';
          end if;
       -- check if current sum is negative
       elsif S(0)(15) = '1' then
          -- set Wn for good sum value
          if ((ADD_A_BUF(0)(15) and ADD_B_BUF(0)(15)) xor
              (ADD_A_BUF(0)(15) xor ADD_B_BUF(0)(15))) = '1' then
             Wn(0) <= S(0)(15 downto 0);
          -- adjust Wn for lower saturation
          elsif (ADD_A_BUF(0)(15) nor ADD_B_BUF(0)(15)) = '1' then
             Wn(0)(14 downto 0)   <= (others => '1');
             Wn(0)(15)   <= '0';
          end if;
       end if;
    end if;
 -- calculating Yi
 elsif CNT_TERMS = 5 then
    -- still calculating Yi
    if ACCUM_TOG = '0' then
       -- toggle accumulation signal for next term
       ACCUM_TOG   <= '1';
       -- place sum in register
       SUM(0)   <= S(0);
    -- done calculating Yi
    elsif ACCUM_TOG = '1' then
       -- toggle accumulation signals for next term
       ACCUM_TOG   <= '0';
       AB_TOG   <= '0';
       -- reset Wn_DONE flag for next stage
       Wn_DONE  <= '0';
       CNT_TERMS   <= 0;
       -- increment stage counter
       CNT_STAGE   <= CNT_STAGE + 1;
       -- check if there are more stages to calculate
       if CNT_STAGE < F_STAGES - 1 then
          -- for scaled integers
          if SCALE > 1 then
             -- check if current sum is positive
             if S(0)(31) = '0' then
                -- set Yi if sum is good
                if S(0) = S(0)(14 + SCALE downto 0) then
                   Yi(0) <= S(0)(15 + SCALE - 1 downto SCALE - 1);
                   Yi(0)(15)   <= '0';
                -- adjust Yi to upper saturation
```

```
            elsif S(0) > S(0)(14 + SCALE downto 0) then
               Yi(0)(14 downto 0)   <= (others => '1');
               Yi(0)(15)   <= '0';
            end if;
         -- check if current sum is negative
         elsif S(0)(31) = '1' then
            -- set Yi if sum is good
            if to_integer(unsigned(S(0)(31 downto 15 + SCALE) + 1))
               = 0 then
               Yi(0) <= S(0)(15 + SCALE - 1 downto SCALE - 1);
               Yi(0)(15)   <= '1';
            -- adjust Yi to lower saturation
            elsif to_integer(unsigned(S(0)(31 downto 15 + SCALE) + 1))
               /= 0 then
               Yi(0)(14 downto 0)   <= (others => '0');
               Yi(0)(15)   <= '1';
            end if;
         end if;
      -- for no scaling
      elsif SCALE = 1 then
         -- check if current sum is positive
         if S(0)(31) = '0' then
            -- set Yi if sum is good
            if S(0) = S(0)(14 downto 0) then
               Yi(0) <= S(0)(15 downto 0);
               Yi(0)(15)   <= '0';
            -- adjust Yi to upper saturation
            elsif S(0) > S(0)(14 downto 0) then
               Yi(0)(14 downto 0)   <= (others => '1');
               Yi(0)(15)   <= '0';
            end if;
         -- check if current sum is negative
         elsif S(0)(31) = '1' then
            -- set Yi if sum is good
            if to_integer(unsigned(S(0)(31 downto 15) + 1))
               = 0 then
               Yi(0) <= S(0)(15 downto 0);
               Yi(0)(15)   <= '1';
            -- adjust Yi to lower saturation
            elsif to_integer(unsigned(S(0)(31 downto 15) + 1))
               /= 0 then
               Yi(0)(14 downto 0)   <= (others => '0');
               Yi(0)(15)   <= '1';
            end if;
         end if;
      end if;
   -- all filter stages are calculated
   elsif CNT_STAGE = F_STAGES - 1 then
      -- for integer scaling
      if SCALE > 1 then
         -- add scaling to temp output for comparison
         TMP_OUT  := S(0) + SCALED_OFFSET;
         -- set output for good value
         if TMP_OUT = TMP_OUT(11 + SCALE - 1 downto 0) then
            Yn_BUF   <= TMP_OUT(11 + SCALE - 1 downto SCALE - 1);
         elsif TMP_OUT > TMP_OUT(11 + SCALE - 1 downto 0) then
            -- adjust output for upper saturation
            if TMP_OUT(31) = '0' then
               Yn_BUF   <= (others => '1');
            -- adjust output for lower saturation
            elsif TMP_OUT(31) = '1' then
               Yn_BUF   <= (others => '0');
            end if;
         end if;
```

```
            elsif SCALE = 1 then
               -- add scaling to temp output for comparison
               TMP_OUT  := S(0) + OFFSET;
               -- set output for good value
               if TMP_OUT = TMP_OUT(11 downto 0) then
                  Yn_BUF   <= TMP_OUT(11 downto 0);
               elsif TMP_OUT > TMP_OUT(11 downto 0) then
                  if TMP_OUT(31) = '0' then
                     -- adjust output for upper saturation
                     Yn_BUF   <= (others => '1');
                  elsif TMP_OUT(31) = '1' then
                     -- adjust output for lower saturation
                     Yn_BUF   <= (others => '0');
                  end if;
               end if;
            end if;
         end if;
      end if;
   end if;
-- for parallel operation
elsif SorP = 1 then
   -- still calculating Wi
   if ACCUM_OUTPUT = '0' then
      -- accumulating Ak terms
      if AB_TOG = '0' then
         if ACCUM_TOG = '0' then
            -- reset accumulation toggle for next term
            ACCUM_TOG   <= '1';
            for i in 0 to F_STAGES - 1 loop
               -- for scaled integers
               if SCALE > 1 then
                  -- check if current sum is positive
                  if S(i)(31) = '0' then
                     -- place sum in register if good value
                     if S(i) = S(i)(14 + SCALE downto 0) then
                        SUM(i)(15 downto 0)
                           <= S(i)(15 + SCALE - 1 downto SCALE - 1);
                        SUM(i)(15)  <= '0';
                     -- adjust sum for upper saturation
                     elsif S(i) > S(i)(14 + SCALE downto 0) then
                        SUM(i)(14 downto 0)  <= (others => '1');
                        SUM(i)(31 downto 15) <= (others => '0');
                     end if;

                  elsif S(i)(31) = '1' then
                     -- place descaled sum in register if good value
                     if to_integer(unsigned(S(i)(31 downto 15 + SCALE) + 1))
                        = 0 then
                        SUM(i)(15 downto 0)
                           <= S(i)(15 + SCALE - 1 downto SCALE - 1);
                        SUM(i)(31 downto 15 + SCALE)  <= (others => '1');
                     -- adjust sum for lower saturation
                     elsif to_integer(unsigned(S(i)(31 downto 15 + SCALE) + 1))
                        /= 0 then
                        SUM(i)(14 downto 0)  <= (others => '0');
                        SUM(i)(31 downto 15) <= (others => '1');
                     end if;
                  end if;
               -- no integer scaling
               elsif SCALE = 1 then
                  -- check if current sum is positive
                  if S(i)(31) = '0' then
                     -- place sum in register if good value
                     if S(i) = S(i)(14 downto 0) then
```

```
                  SUM(i)(15 downto 0)  <= S(i)(15 downto 0);
                  SUM(i)(15)  <= '0';
               -- adjust sum for upper saturation
               elsif S(i) > S(i)(14 downto 0) then
                  SUM(i)(14 downto 0)  <= (others => '1');
                  SUM(i)(31 downto 15) <= (others => '0');
               end if;
            -- check if current sum is negative
            elsif S(i)(31) = '1' then
               if to_integer(unsigned(S(i)(31 downto 15) + 1)) = 0 then
                  SUM(i)(15 downto 0)  <= S(i)(15 downto 0);
                  SUM(i)(31 downto 15) <= (others => '1');
               -- adjust sum for llower saturation
               elsif to_integer(unsigned(S(i)(31 downto 15) + 1)) /= 0 then
                  SUM(i)(14 downto 0)  <= (others => '0');
                  SUM(i)(31 downto 15) <= (others => '1');
               end if;
            end if;
         end if;
      end loop;
   -- done calculating Wn
   elsif ACCUM_TOG = '1' then
      -- reset toggle signals for next accumulation
      ACCUM_TOG   <= '0';
      AB_TOG   <= '1';
      -- set Wn_DONE flag to show Wi calculation is complete
      Wn_DONE  <= '1';
      -- store Wi for each stage in register
      for i in 0 to F_STAGES - 1 loop
         -- check if sum is positive
         if S(i)(15) = '0' then
            -- good Wi value
            if ((ADD_A_BUF(i)(15) nor ADD_B_BUF(i)(15)) xor
                (ADD_A_BUF(i)(15) xor ADD_B_BUF(i)(15))) = '1' then
               Wn(i) <= S(i)(15 downto 0);
            -- adjust Wi for upper saturation
            elsif (ADD_A_BUF(i)(15) and ADD_B_BUF(i)(15)) = '1' then
               Wn(i)(14 downto 0)   <= (others => '0');
               Wn(i)(15)   <= '1';
            end if;
         -- check if sum is negative
         elsif S(i)(15) = '1' then
            -- good Wi value
            if ((ADD_A_BUF(i)(15) and ADD_B_BUF(i)(15)) xor
                (ADD_A_BUF(i)(15) xor ADD_B_BUF(i)(15))) = '1' then
               Wn(i) <= S(i)(15 downto 0);
            -- adjust Wi for lower saturation
            elsif (ADD_A_BUF(i)(15) nor ADD_B_BUF(i)(15)) = '1' then
               Wn(i)(14 downto 0)   <= (others => '1');
               Wn(i)(15)   <= '0';
            end if;
         end if;
      end loop;
   end if;
-- calculating Yi
elsif AB_TOG = '1' then
   -- reset toggle signals for next accumulation
   AB_TOG   <= '0';
   ACCUM_OUTPUT   <= '1';
   -- reset Wn_DONE flag for next calculation sequence
   Wn_DONE  <= '0';
   -- store Yi in register
   for i in 0 to F_STAGES - 1 loop
      -- for integer scaling
```

```
            if SCALE > 1 then
               -- if current sum is positive
               if S(i)(31) = '0' then
                  -- place Yi in register for good value
                  if S(i) = S(i)(14 + SCALE downto 0) then
                     Yi(i) <= S(i)(15 + SCALE - 1 downto SCALE - 1);
                     Yi(i)(15)   <= '0';
                  -- adjust Yi for upper saturation
                  elsif S(i) > S(i)(14 + SCALE downto 0) then
                     Yi(i)(14 downto 0)   <= (others => '1');
                     Yi(i)(15)   <= '0';
                  end if;
               -- check if current sum is negative
               elsif S(i)(31) = '1' then
                  -- place Yi in register if good value
                  if to_integer(unsigned(S(i)(31 downto 15 + SCALE) + 1))
                     = 0 then
                     Yi(i) <= S(i)(15 + SCALE - 1 downto SCALE - 1);
                     Yi(i)(15)   <= '1';
                  -- adjust sum for lower saturation
                  elsif to_integer(unsigned(S(i)(31 downto 15 + SCALE) + 1))
                     /= 0 then
                     Yi(i)(14 downto 0)   <= (others => '0');
                     Yi(i)(15)   <= '1';
                  end if;
               end if;
            -- for no integer scaling
            elsif SCALE = 1 then
               -- check if current sum value is positive
               if S(i)(31) = '0' then
                  -- place Yi in register if good value
                  if S(i) = S(i)(14 downto 0) then
                     Yi(i) <= S(i)(15 downto 0);
                     Yi(i)(15)   <= '0';
                  -- adjust Yi for upper saturation
                  elsif S(i) > S(i)(14 downto 0) then
                     Yi(i)(14 downto 0)   <= (others => '1');
                     Yi(i)(15)   <= '0';
                  end if;
               -- check if current sum value is positive
               elsif S(i)(31) = '1' then
                  -- place Yi in register if good value
                  if to_integer(unsigned(S(i)(31 downto 15) + 1)) = 0 then
                     Yi(i) <= S(i)(15 downto 0);
                     Yi(i)(15)   <= '1';
                  -- adjust Yi for lower saturation
                  elsif to_integer(unsigned(S(i)(31 downto 15) + 1)) /= 0 then
                     Yi(i)(14 downto 0)   <= (others => '0');
                     Yi(i)(15)   <= '1';
                  end if;
               end if;
            end if;
         end loop;
      end if;
   -- done calculating output
   elsif ACCUM_OUTPUT = '1' then
      ACCUM_OUTPUT   <= '0';
      -- for integer scaling
      if SCALE > 1 then
         -- apply offset to temp output for comparison
         TMP_OUT  := S(9) + SCALED_OFFSET;
         -- set output if good value
         if TMP_OUT = TMP_OUT(11 + SCALE - 1 downto 0) then
            Yn_BUF   <= TMP_OUT(11 + SCALE - 1 downto SCALE - 1);
```

```
                  elsif TMP_OUT > TMP_OUT(11 + SCALE - 1 downto 0) then
                     -- adjust Yi for upper saturation
                     if TMP_OUT(31) = '0' then
                        Yn_BUF   <= (others => '1');
                     -- adjust Yi for lower saturation
                     elsif TMP_OUT(31) = '1' then
                        Yn_BUF   <= (others => '0');
                     end if;
                  end if;

               -- for no integer scaling
               elsif SCALE = 1 then
                  -- apply offset to temp output for comparison
                  TMP_OUT  := S(9) + OFFSET;
                  -- set output if good value
                  if TMP_OUT = TMP_OUT(11 downto 0) then
                     Yn_BUF   <= TMP_OUT(11 downto 0);
                  elsif TMP_OUT > TMP_OUT(11 downto 0) then
                     -- adjust Yi for upper saturation
                     if TMP_OUT(31) = '0' then
                        Yn_BUF   <= (others => '1');
                     elsif TMP_OUT(31) = '1' then
                        -- adjust Yi for lower saturation
                        Yn_BUF   <= (others => '0');
                     end if;
                  end if;
               end if;
            end if;
         end if;

-------------------------------------------------------------------------------
         -- set flag to signal output has been calculated
         if GOT_OUTPUT = '1' then
            OUTPUT_DONE <= '1';
         end if;
      end if;
   end process sync_regs;

--===============================================================================
-- This process synchronizes state changes for the FSM.
--===============================================================================
   sync_proc   : process( NS, CLK, RST ) is
   begin
      -- asynchronous reset
      if RST = '1' then
         PS <= IDLE;
      elsif rising_edge(CLK) then
         PS <= NS;
      end if;
   end process sync_proc;

--===============================================================================
-- This process contains the combinatorial logic used for state changes of the
-- FSM and the various other operations carried out during DSP calculations.
--===============================================================================
   comb_proc   : process( PS, EN, SAMP_DONE, CNT_TERMS ) is
   begin
      case PS is
      -- state, idle/initialization
      when IDLE =>
         -- set flag to initialize registers and signals
         CLR_REGS <= '1';
         -- initialize FSM flags
```

```
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_ADD <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- change state to wait for a sample when system enable is received
   if EN = '1' then
      NS <= WAIT4Xn;
   elsif EN = '1' then
      NS <= IDLE;
   end if;

-- state, wait for sample from ADC
when WAIT4Xn =>
   -- clear registers and signals
   CLR_REGS <= '1';
   -- assign remaining flags of FSM
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_ADD <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- cange to the multiply state after sample acquisition
   if SAMP_DONE = '0' then
      NS <= WAIT4Xn;
   elsif SAMP_DONE = '1' then
      NS <= MULT;
   end if;

-- state, multiply filter terms
when MULT =>
   -- set flag to load multiplier inputs
   LOAD_MULT   <= '1';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_P   <= '0';
   LOAD_ADD <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- go to state that loads product register
   NS <= MULT_DUM;

-- state, dummy for MULT
when MULT_DUM =>
   LOAD_ADD <= '0';
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   NS <= GET_P;

-- state, load product register
when GET_P =>
   -- set flag to load product register
   LOAD_P   <= '1';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_ADD <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   if SorP = 0 then
      -- continue multiplying terms
```

```
      if (CNT_TERMS /= 2) and (CNT_TERMS /= 5) then
         NS <= MULT;
      -- done multiplying terms, begin accumulation
      elsif (CNT_TERMS = 2) or (CNT_TERMS = 5) then
         NS <= P_DUM;
      end if;
   elsif SorP = 1 then
      NS <= P_DUM;
   end if;

-- state, dummy for GET_P
when P_DUM =>
   LOAD_ADD <= '0';
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   NS <= ACCUM;

-- state, accumulate filter terms
when ACCUM =>
   -- set flag to load adder inputs
   LOAD_ADD <= '1';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- for serial operation go to the state that loads the
   -- accumulation register
   if SorP = 0 then
      NS <= A_DUM;
   -- for parallel operation go to the dummy state needed for
   -- calculations made with the 32-bit Carry-Save accumulator
   elsif SorP = 1 then
      NS <= A_DUM;
   end if;

-- state, dummy for ACCUM
when A_DUM =>
   -- set flag to load adder inputs
   LOAD_ADD <= '0';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   -- go to the state that loads the accumulation register
   NS <= GET_S;

-- state, loads accumulation register
when GET_S =>
   -- set flag that loads the accumulation register
   LOAD_S   <= '1';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_ADD <= '0';
   GOT_OUTPUT  <= '0';
   -- for serial operation
```

```
   if SorP = 0 then
      if CNT_TERMS = 2 then
         -- still accumulating Wi
         if ACCUM_TOG = '0' then
            NS <= ACCUM;
         -- done accumulating Wi
         elsif ACCUM_TOG = '1' then
            NS <= MULT;
         end if;
      elsif CNT_TERMS = 5 then
         -- still accumulating Yi
         if ACCUM_TOG = '0' then
            NS <= ACCUM;
         -- done accumulating Yi
         elsif ACCUM_TOG = '1' then
            -- begin multiplications for next stage
            if CNT_STAGE < F_STAGES - 1 then
               NS <= MULT;
            -- output done
            elsif CNT_STAGE = F_STAGES - 1 then
               NS <= OUT_DUM1;
            end if;
         end if;
      end if;
   -- for parallel operation
   elsif SorP = 1 then
      if ACCUM_OUTPUT = '0' then
         if ACCUM_TOG = '0' then
            NS <= ACCUM;
         elsif ACCUM_TOG = '1' then
            NS <= MULT;
         end if;
      elsif ACCUM_OUTPUT = '1' then
         NS <= OUT_DUM1;
      end if;
   end if;

-- state, first dummy for output
when OUT_DUM1 =>
   LOAD_ADD <= '0';
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   NS <= OUT_DUM2;

-- state, second dummy for output
when OUT_DUM2 =>
   LOAD_ADD <= '0';
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
   LOAD_P   <= '0';
   LOAD_S   <= '0';
   GOT_OUTPUT  <= '0';
   NS <= SET_OUTPUT;

-- state, signals that the final output value has been set
when SET_OUTPUT =>
   -- flag to sync completion flag, OUTPUT_DONE
   GOT_OUTPUT  <= '1';
   -- assign remaining flags of FSM
   CLR_REGS <= '0';
   LOAD_MULT   <= '0';
```

```
         LOAD_P   <= '0';
         LOAD_ADD <= '0';
         LOAD_S   <= '0';
         -- go to state to wait for next sample from ADC
         NS <= WAIT4Xn;

      when others =>
         NS <= IDLE;

      end case;
   end process comb_proc;
end behavioral;
```

DSP_BB.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      DSP Hardware Generator
-- Module Name:      DSP_BB - behavioral
-- Project Name:     senior_project
-- Description:      DSP Hardware Generator generates and organizes arithmetic
--                   components for filter calculations.
----------------------------------------------------------------------------------
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 DSP_BB is
   port( MULT_A      : in vect_16x20;
         MULT_B      : in vect_16x20;
         ADD_A : in vect_32x20;
         ADD_B : in vect_32x20;
         PART_S   : in vect_32x20;
         P           : out vect_32x20;
         S           : out vect_32x20;
         As    : out vect_16x20;
         Bs    : out vect_16x30;
         C_VECT   : out std_logic_vector(15 downto 0) );
end DSP_BB;

architecture behavioral of DSP_BB is
-- carry-out bit from adder
signal ADD_CO  : std_logic_vector(8 downto 0) := (others => '0');
signal CO   : std_logic := '0';

begin
   -- generate hardware for NORMAL.vhd module
   normal   : if STRUCTURE = 0 generate
      mult_gen : for i in 0 to F_LENGTH generate
         -- generate multipliers for serial operation
         s_mult_gen  : if SorP = 0 generate
            mult0 : if i = 0 generate
               SA_choose   : if MULTIPLIER = 0 generate
                  SA_s_mult   : SA_MULT_16BIT
                  port map( MULT_A(i), MULT_B(i), P(i) );
               end generate SA_choose;
```

```
         booth_choose   : if MULTIPLIER = 1 generate
            booth_s_mult   : BOOTH_MULT_16BIT
            port map( MULT_A(i), MULT_B(i), P(i) );
         end generate booth_choose;

         MULT18X18_choose  : if MULTIPLIER = 2 generate
            MULT18X18_s_mult  : MULT18X18
            port map( MULT_A(i), MULT_B(i), P(i) );
         end generate MULT18X18_choose;
      end generate mult0;
   end generate s_mult_gen;

   -- generate multipliers parallel operation
   p_mult_gen  : if SorP = 1 generate
      SA_choose   : if MULTIPLIER = 0 generate
         SA_p_mult   : SA_MULT_16BIT
         port map( MULT_A(i), MULT_B(i), P(i) );
      end generate SA_choose;

      booth_choose   : if MULTIPLIER = 1 generate
         booth_p_mult   : BOOTH_MULT_16BIT
         port map( MULT_A(i), MULT_B(i), P(i) );
      end generate booth_choose;

      MULT18X18_choose  : if MULTIPLIER = 2 generate
         MULT18X18_p_mult  : MULT18X18
         port map( MULT_A(i), MULT_B(i), P(i) );
      end generate MULT18X18_choose;
   end generate p_mult_gen;
end generate mult_gen;

-- generate adder for serial operation
s_add_gen   : if SorP = 0 generate
   RC_add_gen  : if ADDER = 0 generate
      RC_add   : RC_ADDER_32BIT
      port map( ADD_A(0), ADD_B(0), CO, S(0) );
   end generate RC_add_gen;

   CSe_add_gen : if ADDER = 1 generate
      CSe_add  : CSe_ADDER_32BIT
      port map( ADD_A(0), ADD_B(0), CO, S(0) );
   end generate CSe_add_gen;

   CLa_add_gen : if ADDER = 2 generate
      CLa_add  : CLa_ADDER_32BIT
      port map( ADD_A(0), ADD_B(0), CO, S(0) );
   end generate CLa_add_gen;
end generate s_add_gen;

-- generate accumulator for parallel operation
p_add_gen   : if SorP = 1 generate
   CSa_accum   : CSa_ACCUM_32BIT
   port map( PART_S, CO, S(0) );
end generate p_add_gen;

-- change integer filter coefficients to vectors
init_coeffs_n  : for i in 0 to 19 generate
   init_Bs_n   : if i < M generate
      Bs(i) <= std_logic_vector(to_signed(Bk(i), Bs(i)'length));
   end generate init_Bs_n;
   zero_Bs_n   : if i >= M generate
      Bs(i) <= (others => '0');
   end generate zero_Bs_n;
```

```
         init_As_n   : if i < N generate
            As(i) <= std_logic_vector(to_signed(Ak(i), As(i)'length));
         end generate init_As_n;
         zero_As_n   : if i >= N generate
            As(i) <= (others => '0');
         end generate zero_As_n;
      end generate init_coeffs_n;
   end generate normal;

-------------------------------------------------------------------------------
   -- generate hardware for CASCADE.vhd module
   cascade  : if STRUCTURE = 1 generate
      -- generate multipliers for serial operation
      mult_gen : for i in 0 to F_STAGES * 2 generate
         s_mult_gen  : if SorP = 0 generate
            mult0 : if i = 0 generate
               SA_choose   : if MULTIPLIER = 0 generate
                  SA_s_mult   : SA_MULT_16BIT
                  port map( MULT_A(i), MULT_B(i), P(i) );
               end generate SA_choose;

               booth_choose   : if MULTIPLIER = 1 generate
                  booth_s_mult   : BOOTH_MULT_16BIT
                  port map( MULT_A(i), MULT_B(i), P(i) );
               end generate booth_choose;

               MULT18X18_choose  : if MULTIPLIER = 2 generate
                  MULT18X18_s_mult  : MULT18X18
                  port map( MULT_A(i), MULT_B(i), P(i) );
               end generate MULT18X18_choose;
            end generate mult0;
         end generate s_mult_gen;

         -- generate multipliers for parallel operation
         p_mult_gen  : if SorP = 1 generate
            SA_choose   : if MULTIPLIER = 0 generate
               SA_p_mult   : SA_MULT_16BIT
               port map( MULT_A(i), MULT_B(i), P(i) );
            end generate SA_choose;

            booth_choose   : if MULTIPLIER = 1 generate
               booth_p_mult   : BOOTH_MULT_16BIT
               port map( MULT_A(i), MULT_B(i), P(i) );
            end generate booth_choose;

            MULT18X18_choose  : if MULTIPLIER = 2 generate
               MULT18X18_p_mult  : MULT18X18
               port map( MULT_A(i), MULT_B(i), P(i) );
            end generate MULT18X18_choose;
         end generate p_mult_gen;
      end generate mult_gen;

      -- instantiations for adders used during serial and parallel operation
      add_gen  : for i in 0 to F_STAGES - 1 generate
         RC_choose   : if ADDER = 0 generate
            RC_serial   : if SorP = 0 generate
               RC_add0  : if i = 0 generate
                  RC_s_add : RC_ADDER_32BIT
                  port map( ADD_A(i), ADD_B(i), ADD_CO(i), S(i) );
               end generate RC_add0;
            end generate RC_serial;

            RC_parallel : if SorP = 1 generate
               RC_p_add : RC_ADDER_32BIT
```

```
         port map( ADD_A(i), ADD_B(i), ADD_CO(i), S(i) );
      end generate RC_parallel;
   end generate RC_choose;

   CSe_choose  : if ADDER = 1 generate
      CSe_serial  : if SorP = 0 generate
         CSe_add0 : if i = 0 generate
            CSe_s_add   : CSe_ADDER_32BIT
            port map( ADD_A(i), ADD_B(i), ADD_CO(i), S(i) );
         end generate CSe_add0;
      end generate CSe_serial;

      CSe_parallel   : if SorP = 1 generate
         CSe_p_add   : CSe_ADDER_32BIT
         port map( ADD_A(i), ADD_B(i), ADD_CO(i), S(i) );
      end generate CSe_parallel;
   end generate CSe_choose;

   CLa_choose  : if ADDER = 2 generate
      CLa_serial  : if SorP = 0 generate
         CLa_add0 : if i = 0 generate
            CLa_s_add   : CLa_ADDER_32BIT
            port map( ADD_A(i), ADD_B(i), ADD_CO(i), S(i) );
         end generate CLa_add0;
      end generate CLa_serial;

      CLa_parallel   : if SorP = 1 generate
         CLa_p_add   : CLa_ADDER_32BIT
         port map( ADD_A(i), ADD_B(i), ADD_CO(i), S(i) );
      end generate CLa_parallel;
   end generate CLa_choose;
end generate add_gen;

-- instantiation for 32-bit Carry-Save accumulator used
-- during parallel operation
p_accum_gen : if SorP = 1 generate
   CSa_accum   : CSa_ACCUM_32BIT
   port map( PART_S, CO, S(9) );
end generate p_accum_gen;

-- convert of Ak and Bk terms from integers to vectors
s_coeffs : if SorP = 0 generate
   init_s_coeffs  : for i in 0 to F_STAGES * 5 - 1 generate
      init_Bs_S   : if i < F_STAGES * 3 generate
         Bs(i) <= std_logic_vector(to_signed(Bki_S(i), Bs(i)'length));
      end generate init_Bs_S;

      init_As  : if i >= F_STAGES * 3 generate
         As(i - F_STAGES * 3) <= std_logic_vector(
                                 to_signed(Aki(i - F_STAGES * 3),
                                 As(i - F_STAGES * 3)'length));
      end generate init_As;
   end generate init_s_coeffs;
end generate s_coeffs;

p_coeffs : if SorP = 1 generate
   init_p_coeffs  : for i in 0 to F_STAGES * 4 - 1 generate
      init_Bs_P   : if i < F_STAGES * 2 generate
         Bs(i) <= std_logic_vector(to_signed(Bki_P(i), Bs(i)'length));
      end generate init_Bs_P;

      init_As  : if i >= F_STAGES * 2 generate
         As(i - F_STAGES * 2) <= std_logic_vector(
                                 to_signed(Aki(i - F_STAGES * 2),
```

```
                                    As(i - F_STAGES * 2)'length));
          end generate init_As;
       end generate init_p_coeffs;
    end generate p_coeffs;

    C_VECT   <= std_logic_vector(to_signed(C, C_VECT'length));
 end generate cascade;

-------------------------------------------------------------------------------
-- hardware generation for CASC_DFI.vhd module
-- casc_gen : if STRUCTURE = 2 generate
--    mult_gen : for i in 0 to F_LENGTH generate
--       s_mult_gen  : if SorP = 0 generate
--          mult0 : if i = 0 generate
--             SA_choose   : if MULTIPLIER = 0 generate
--                SA_s_mult   : SA_MULT_16BIT
--                port map( MULT_A(i), MULT_B(i), P(i) );
--             end generate SA_choose;
--
--             booth_choose   : if MULTIPLIER = 1 generate
--                booth_s_mult   : BOOTH_MULT_16BIT
--                port map( MULT_A(i), MULT_B(i), P(i) );
--             end generate booth_choose;
--
--             MULT18X18_choose  : if MULTIPLIER = 2 generate
--                MULT18X18_s_mult  : MULT18X18
--                port map( MULT_A(i), MULT_B(i), P(i) );
--             end generate MULT18X18_choose;
--          end generate mult0;
--       end generate s_mult_gen;
--
--       p_mult_gen  : if SorP = 1 generate
--          SA_choose   : if MULTIPLIER = 0 generate
--             SA_p_mult   : SA_MULT_16BIT
--             port map( MULT_A(i), MULT_B(i), P(i) );
--          end generate SA_choose;
--
--          booth_choose   : if MULTIPLIER = 1 generate
--             booth_p_mult   : BOOTH_MULT_16BIT
--             port map( MULT_A(i), MULT_B(i), P(i) );
--          end generate booth_choose;
--
--          MULT18X18_choose  : if MULTIPLIER = 2 generate
--             MULT18X18_p_mult  : MULT18X18
--             port map( MULT_A(i), MULT_B(i), P(i) );
--          end generate MULT18X18_choose;
--       end generate p_mult_gen;
--    end generate mult_gen;
--
--    s_add_gen   : if SorP = 0 generate
--       RC_add_gen  : if ADDER = 0 generate
--          RC_add   : RC_ADDER_32BIT
--          port map( ADD_A(0), ADD_B(0), CO, S(0) );
--       end generate RC_add_gen;
--
--       CSe_add_gen : if ADDER = 1 generate
--          CSe_add  : CSe_ADDER_32BIT
--          port map( ADD_A(0), ADD_B(0), CO, S(0) );
--       end generate CSe_add_gen;
--
--       CLa_add_gen : if ADDER = 2 generate
--          CLa_add  : CLa_ADDER_32BIT
--          port map( ADD_A(0), ADD_B(0), CO, S(0) );
--       end generate CLa_add_gen;
```

```
--    end generate s_add_gen;
--
--    p_add_gen   : if SorP = 1 generate
--       CSa_accum   : CSa_ACCUM_32BIT
--       port map( PART_S, CO, S(0) );
--    end generate p_add_gen;
--
--    init_coeffs_c  : for i in 0 to F_STAGES * 3 - 1 generate
--       init_Bs_c   : if i < F_STAGES * 3 generate
--          Bs(i) <= std_logic_vector(to_signed(Bki(i), Bs(i)'length));
--       end generate init_Bs_c;
--       zero_Bs_c   : if i >= F_STAGES * 3 generate
--          Bs(i) <= (others => '0');
--       end generate zero_Bs_c;
--       init_As_c   : if i < F_STAGES * 2 generate
--          As(i) <= std_logic_vector(to_signed(Aki(i), As(i)'length));
--       end generate init_As_c;
--       zero_As_c   : if i >= F_STAGES * 2 generate
--          As(i) <= (others => '0');
--       end generate zero_As_c;
--    end generate init_coeffs_c;
-- end generate casc_gen;
end behavioral;
```

SAMPLE_CTRL.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      Sample Control
-- Module Name:      SAMPLE_CTRL - behavioral
-- Project Name:     senior_project
-- Description:      Sample Control module controls sampling timing for the
--                   ADC and DAC.
----------------------------------------------------------------------------------
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 SAMPLE_CTRL is
   port( CLK         : in  std_logic;
         RST         : in  std_logic;
         EN          : in  std_logic;
         MISO        : in  std_logic;
         Yn          : in  std_logic_vector(11 downto 0);
         OUTPUT_DONE : in std_logic;
         ADC_SCLK    : out std_logic;
         DAC_SCLK    : out std_logic;
         CS          : out std_logic;
         SYNC        : out std_logic;
         MOSI        : out std_logic;
         Xn          : out std_logic_vector(11 downto 0);
         XnmM        : out vect_16x20;
         YnmN        : out vect_16x20;
         SAMP_DONE   : out std_logic );
end SAMPLE_CTRL;
```

```
architecture behavioral of SAMPLE_CTRL is
-- FSM states declarations
type state_type is (IDLE, SAMP_START, SAMPLE, HOLD);
signal PS, NS  :  state_type;
-- sampling clock timing
signal SAMP_CLK   : std_logic := '0';
-- flag to start sampling
signal GET_SAMP   : std_logic := '0';
signal NEW_SAMP   : std_logic := '0';
-- buffer for SAMP_DONE output flag
signal SAMP_DONE_INT : std_logic := '0';
-- enable for read sequences
signal RD_EN   : std_Logic := '0';
-- flag to sync RD_EN output
signal RD   : std_Logic := '0';
-- enable for write sequences
signal WRT_EN  : std_Logic := '0';
-- flag to sync WRT_EN output
signal WRT  : std_Logic := '0';
-- flag to initialize sampling sequence
signal FIRST_SAMP : std_logic := '1';
signal SET_FIRST  : std_logic := '0';
-- preset and clear signals for shift registers
signal PRE  : std_logic := '0';
signal CLR  : std_logic := '0';
-- shift register buffers for ADC input and DAC output signals
signal Xn_SFT  : std_logic_vector(15 downto 0) := (others => '0');
signal Yn_SFT  : std_logic_vector(15 downto 0) := (others => '0');
-- ADC input buffer
signal Xn_TMP  : std_logic_vector(11 downto 0) := (others => '0');
-- output shift register buffer
signal YnmN_INT   : vect_16x20 := (others => (others => '0'));

begin
   -- buffer ADC input and DAC output do sent to shift registers
   Xn_SFT   <= "0000" & Xn_TMP;
   Yn_SFT   <= "0000" & Yn;
   -- assign shift register clear signal
   CLR   <= RST;
   -- assign SAMP_DONE output flag
   SAMP_DONE   <= SAMP_DONE_INT;
   -- output input from ADC for filtering
   Xn <= Xn_TMP;
   -- output shifted DAC output values for filtering
   YnmN  <= YnmN_INT;

   -- set timing for sampling
   sample_clk  :  CLK_DIV port map( SAMP_DIV, CLK, SAMP_CLK );

   -- ADC and DAC component control
   convert  :  CONVERTER_CTRL port map( CLK, RST, RD_EN, WRT_EN, MISO,
                                        YnmN_INT(0)(11 downto 0), ADC_SCLK,
                                        DAC_SCLK, CS, SYNC,MOSI, Xn_TMP,
                                        SAMP_DONE_INT );

   -- shift register for ADC inputs
   shift_x  : SIPO_SHR
   port map( SAMP_DONE_INT, CLR, PRE, Xn_SFT, XnmM );

   -- shift register for DAC outputs
   shift_y  : SIPO_SHR
   port map( OUTPUT_DONE, CLR, PRE, Yn_SFT, YnmN_INT );

--------------------------------------------------------------------------------
```

```
-- process to trigger sampling sequence
--------------------------------------------------------------------------------
   samp_trig   : process( CLK, RST ) is
   begin
      -- asynchronous reset
      if RST = '1' then
         GET_SAMP <= '0';
      elsif rising_edge(CLK) then
         if (SAMP_CLK) = '1' then
            if FIRST_SAMP = '0' then
               GET_SAMP <= '1';
            elsif FIRST_SAMP = '1' then
               if EN = '1' then
                  GET_SAMP <= '1';
               end if;
            end if;
         elsif SAMP_CLK = '0' then
            GET_SAMP <= '0';
         end if;
         if (SAMP_CLK and SAMP_DONE_INT) = '1' then
            NEW_SAMP <= '0';
         elsif SAMP_CLK = '0' then
            NEW_SAMP <= '1';
         end if;
      end if;
   end process samp_trig;

--------------------------------------------------------------------------------
-- FSM, sampling sequence control
--------------------------------------------------------------------------------
   -- synchronous process for state changes
   sync_proc   : process( NS, CLK, RST ) is
   begin
      -- asynchronous reset
      if RST = '1' then
         PS <= IDLE;
      elsif rising_edge(CLK) then
         -- sync RD_EN based on RD flag
         if RD = '0' then
            RD_EN <= '0';
         elsif RD = '1' then
            RD_EN <= '1';
         end if;

         -- sync WRT_EN based on WRT flag
         if WRT = '0' then
            WRT_EN   <= '0';
         elsif WRT = '1' then
            WRT_EN   <= '1';
         end if;

         -- set FIRST_SAMP flag to 0 after first sample
         if SAMP_DONE_INT = '1' then
            if FIRST_SAMP = '1' then
               FIRST_SAMP  <= '0';
            end if;
         -- reset FIRST_SAMP flag
         elsif SET_FIRST = '1' then
            FIRST_SAMP  <= '1';
         end if;

         PS <= NS;
      end if;
   end process sync_proc;
```

```
-- combinational process for state logic
comb_proc   : process( PS, EN, GET_SAMP, FIRST_SAMP, SAMP_DONE_INT, NEW_SAMP ) is
begin
   case PS is
   -- state, idle/initialization
   when IDLE =>
      -- initialize read and write enables and FIRST_SAMP flag
      RD <= '0';
      WRT   <= '0';
      SET_FIRST   <= '1';
      if EN = '1' then
         -- go to sampling state on system enable
         NS <= SAMP_START;
      else
         NS <= IDLE;
      end if;

   when SAMP_START =>
      RD <= '0';
      WRT   <= '0';
      SET_FIRST   <= '0';
      -- wait for GET_SAMP flag to begin sampling sequence
      if GET_SAMP = '0' then
         NS <= SAMP_START;
      elsif GET_SAMP = '1' then
         NS <= SAMPLE;
      end if;

   -- state, sampling acquizition takes place in this state
   when SAMPLE =>
      SET_FIRST   <= '0';
      if SAMP_DONE_INT = '0' then
         -- enable reading from ADC on GET_SAMP flag assertion
         RD <= '1';
         NS <= SAMPLE;
      elsif SAMP_DONE_INT = '1' then
         RD <= '0';
         -- go to hold state when sample is acquired
         NS <= HOLD;
      end if;
      if FIRST_SAMP = '0' then
         WRT   <= '1';
      elsif FIRST_SAMP = '1' then
         WRT   <= '0';
      end if;

   -- state, hold sample till next sampling interval
   when HOLD =>
      -- enabe DAC to output on SAMP_DONE flag assertion (GET_SAMP = 0)
      WRT   <= '1';
      -- stop ADC read sequence
      RD <= '0';
      SET_FIRST   <= '0';
      if NEW_SAMP = '1' then
         if GET_SAMP = '1' then
            -- go back for next sample on GET_SAMP flag assertion
            NS <= SAMPLE;
         elsif GET_SAMP = '0' then
            NS <= HOLD;
         end if;
      elsif NEW_SAMP = '0' then
         NS <= HOLD;
      end if;
```

```
      when others =>
         NS <= IDLE;

      end case;
   end process;
end behavioral;
```

CONVERTER_CTRL.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      ADC/DAC Control
-- Module Name:      CONVERTER_CTRL - behavioral
-- Project Name:     senior_project
-- Description:      ADC/DAC Control module controls SPI interfaces to the
--                   ADC and DAC.
----------------------------------------------------------------------------------
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 CONVERTER_CTRL is
   port( CLK         :  in    std_logic;
         RST         :  in    std_logic;
         RD_EN       :  in    std_logic;
         WRT_EN      :  in    std_logic;
         MISO        :  in    std_logic;
         Yn          :  in    std_logic_vector(11 downto 0);
         ADC_SCLK    :  out   std_logic;
         DAC_SCLK    :  out   std_logic;
         CS          :  out   std_logic;
         SYNC        :  out   std_logic;
         MOSI        :  out   std_logic;
         Xn          :  out   std_logic_vector(11 downto 0);
         SAMP_DONE   :  out   std_logic );
end CONVERTER_CTRL;

architecture behavioral of CONVERTER_CTRL is
-- FSM states declarations for ADC
type state_type_ADC is (IDLE_ADC, SET_ADC, RUN_ADC, SET_SAMP);
signal PS_ADC, NS_ADC   :  state_type_ADC;
-- FSM states declarations for DAC
type state_type_DAC is (IDLE_DAC, SET_DAC, RUN_DAC);
signal PS_DAC, NS_DAC   :  state_type_DAC;
-- control bits for DAC
constant CTRL  : std_logic_vector(3 downto 0) := "0000";
-- store input from ADC
signal RD_REG  : std_logic_vector(15 downto 0) := "0000000000000000";
-- store output to send to DAC
signal WRT_REG : std_logic_vector(15 downto 0) := "0000000000000000";
-- temporary siganl to hold clock signal for ADC
signal ADC_SCLK_TMP  : std_logic := '0';
-- temporary siganl to hold clock signal for DAC
signal DAC_SCLK_TMP  : std_logic := '0';
-- integer to keep track of RD_REG data shifts
signal RD_CNT  : integer range 0 to 15 := 0;
-- integer to keep track of WRT_REG data shifts
```

```
signal WRT_CNT : integer range 0 to 15 := 0;
-- flag to enable RD_CNT incrementation and reset
signal RD_CNT_EN : std_logic := '0';
-- flag to enable WRT_CNT incrementation and reset
signal WRT_CNT_EN : std_logic := '0';
-- flag to load WRT_REG with output data for DAC
signal LOAD_Yn : std_logic := '0';
-- flag to set Xn with input data from ADC
signal SET_Xn     : std_logic := '0';
-- flag to set SAMP_DONE output
signal GOT_SAMP   : std_logic := '0';

--------------------------------------------------------------------------------
-- This module contains a total of 6 processes. The first 3 control operation of
-- the ADC and next 3 control DAC operation.
--------------------------------------------------------------------------------
begin
   -- signal assignment for output data to DAC
   MOSI        <= WRT_REG(15);
   -- signal assignment for ADC clock
   ADC_SCLK    <= ADC_SCLK_TMP;
   -- signal assignment for DAC clock
   DAC_SCLK    <= DAC_SCLK_TMP;

   -- ADC clock timing at 12.5 MHz
   ADC_clk  :  CLK_DIV port map( 2, CLK, ADC_SCLK_TMP );
   -- DAC clock timing at 25 MHz
   DAC_clk  :  CLK_DIV port map( 1, CLK, DAC_SCLK_TMP );

--------------------------------------------------------------------------------
-- process to control RD_REG register for ADC data acquisition
--------------------------------------------------------------------------------
   rd_reg_ctrl : process( CLK, ADC_SCLK_TMP, RST ) is
   begin
      -- asynchronous reset
      if RST = '1' then
         -- initialize read register
         RD_REG   <= "0000000000000000";
         Xn <= "000000000000";
      elsif rising_edge(ADC_SCLK_TMP) then
         if SET_Xn = '1' then
            -- set output Xn when ADC register data transfer complete
            Xn <= RD_REG(11 downto 0);
         end if;

         if RD_CNT_EN = '0' then
            -- reset RD_CNT for next read sequence
            RD_CNT <= 0;
         elsif RD_CNT_EN = '1' then
            -- shift RD_REG and increment counter
            RD_REG   <= RD_REG(14 downto 0) & MISO;
            RD_CNT   <= RD_CNT + 1;
         end if;

         -- sets SAMP_DONE flag when output Xn has been set
         if GOT_SAMP = '0' then
            SAMP_DONE <= '0';
         elsif GOT_SAMP = '1' then
            SAMP_DONE   <= '1';
         end if;
      end if;
   end process rd_reg_ctrl;

--------------------------------------------------------------------------------
```

```
-- FSM, control of SPI interface to ADC
-------------------------------------------------------------------------------
   -- synchronous process for state changes
   ADC_sync_proc  : process( NS_ADC, ADC_SCLK_TMP, RST  ) is
   begin
      -- asynchronous reset
      if RST = '1' then
         PS_ADC   <= IDLE_ADC;
      elsif rising_edge(ADC_SCLK_TMP) then
         PS_ADC <= NS_ADC;
      end if;
   end process ADC_sync_proc;

   -- combinational process for state logic
   ADC_comb_proc  : process( PS_ADC, RD_EN, RD_CNT ) is
   begin
      case PS_ADC is
      -- state, idle/initialization
      when IDLE_ADC =>
         -- initialize signals
         CS <= '0';
         RD_CNT_EN   <= '0';
         SET_Xn   <= '0';
         GOT_SAMP <= '0';
         if RD_EN = '1' then
            -- change to state SET_ADC on assertion of RD_EN flag
            NS_ADC   <= SET_ADC;
         else
            NS_ADC   <= IDLE_ADC;
         end if;

      -- state, setup for read sequence
      when SET_ADC =>
         -- bring CS high to start transfer sequence from ADC
         CS <= '1';
         -- set remaining outputs of FSM
         RD_CNT_EN   <= '0';
         GOT_SAMP <= '0';
         SET_Xn   <= '0';
         -- change state to RUN_ADC to start data transfer
         NS_ADC   <= RUN_ADC;

      -- state, transfer data from ADC data register to RD_REG
      when RUN_ADC =>
         -- set CS low to begin data transfer from ADC
         CS    <= '0';
         -- enable RD_CNT_EN to begin RD_REG control
         RD_CNT_EN   <= '1';
         -- set remaining outputs of FSM
         SET_Xn   <= '0';
         GOT_SAMP <= '0';
         if RD_CNT < 15 then
            -- data tranfer incomplete
            NS_ADC <= RUN_ADC;
         elsif RD_CNT = 15 then
            -- set input from RD_REG
            SET_Xn   <= '1';
            -- reset register counter for next read sequence
            RD_CNT_EN   <= '0';
            -- change to idle state to wait for next sequence
            NS_ADC <= SET_SAMP;
         end if;

      when SET_SAMP =>
```

```
         -- assert GOT_SAMP to set SAMP_DONE output
         GOT_SAMP <= '1';
         -- set remaining outputs of FSM
         CS    <= '0';
         RD_CNT_EN   <= '0';
         SET_Xn   <= '0';
         NS_ADC   <= IDLE_ADC;

      when others =>
         NS_ADC   <= IDLE_ADC;

      end case;
   end process ADC_comb_proc;

-------------------------------------------------------------------------------
-- process to control WRT_REG register for DAC output
-------------------------------------------------------------------------------
   wrt_reg_ctrl   : process( DAC_SCLK_TMP, RST ) is
   begin
      -- asynchronous reset
      if RST = '1' then
         -- initialize write register
         WRT_REG  <= "0000000000000000";
      elsif rising_edge(DAC_SCLK_TMP) then
         if LOAD_Yn = '1' then
            -- load write register with current output for DAC
            WRT_REG <= CTRL & Yn;
         end if;
         if WRT_CNT_EN = '0' then
            -- reset WRT_CNT for next write sequence
            WRT_CNT <= 0;
         elsif WRT_CNT_EN = '1' then
            -- shift WRT_REG and increment counter
            WRT_REG  <= WRT_REG(14 downto 0) & '0';
            WRT_CNT  <= WRT_CNT + 1;
         end if;
      end if;
   end process wrt_reg_ctrl;

-------------------------------------------------------------------------------
-- FSM, control of SPI interface to DAC
-------------------------------------------------------------------------------
   -- synchronous process for state changes
   DAC_sync_proc  : process( NS_DAC, DAC_SCLK_TMP, RST  ) is
   begin
      -- asynchronous reset
      if RST = '1' then
         PS_DAC   <= IDLE_DAC;
      elsif rising_edge(DAC_SCLK_TMP) then
         PS_DAC <= NS_DAC;
      end if;
   end process DAC_sync_proc;

   -- combinational process for state logic
   DAC_comb_proc  : process( PS_DAC, WRT_EN, WRT_CNT ) is
   begin
      case PS_DAC is
      -- state, idle/initialization
      when IDLE_DAC =>
         -- initialize signals
         SYNC     <= '0';
         WRT_CNT_EN  <= '0';
         LOAD_Yn  <= '0';
         if WRT_EN = '1' then
```

```
            -- change to state SET_DAC on assertion of WRT_EN flag
            NS_DAC   <= SET_DAC;
         else
            NS_DAC   <= IDLE_DAC;
         end if;

      -- state, setup for write sequence
      when SET_DAC =>
         WRT_CNT_EN  <= '0';
         -- bring SYNC high to start transfer sequence to DAC
         SYNC  <= '1';
         -- assert flag to sync WRT_REG with desired output
         LOAD_Yn  <= '1';
         -- change state to RUN_DAC to start data transfer
         NS_DAC   <= RUN_DAC;

      -- state, data transfer to DAC
      when RUN_DAC =>
         -- SYNC low to begin DAC data transfer
         SYNC     <= '0';
         -- enable shifting of WRT_REG
         WRT_CNT_EN  <= '1';
         LOAD_Yn <= '0';
         if WRT_CNT < 15 then
            -- data transfer incomplete
            NS_DAC <= RUN_DAC;
         elsif WRT_CNT = 15 then
            -- reset WRT_REG for next data shift
            WRT_CNT_EN <= '0';
            -- go to SET_DAC state to begin another write sequence
            NS_DAC <= SET_DAC;
         end if;

      when others =>
         NS_DAC   <= IDLE_DAC;

      end case;
   end process DAC_comb_proc;
end behavioral;
```

SA_MULT_16BIT.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      16-bit Shift-Add Multiplier
-- Module Name:      SA_MULT_16BIT - behavioral
-- Project Name:     senior_project
-- Description:      16-bit Shift-Add Multiplier accepts two 16-bit inputs
--                   and computes their 32-bit product using shift-add 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 SA_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 SA_MULT_16BIT;

architecture behavioral of SA_MULT_16BIT is
-- carry-in signal for each stage of multiplier
signal CI   : vect_16x20 := (others => (others => '0'));
-- carry-out signal for each stage of multiplier
signal CO   : vect_16x20 := (others => (others => '0'));
-- partial sums generated at each stage of multiplier
signal S    : vect_16x20 := (others => (others => '0'));
-- partials products to be accumulated
signal PP   : vect_16x20 := (others => (others => '0'));
-- inputs to each stage of the multiplier
signal As   : vect_16x20 := (others => (others => '0'));
signal Bs   : vect_16x20 := (others => (others => '0'));

begin
   -- generate stages of multiplier
   SA_rows  : for i in 0 to 15 generate
      SA : for j in 0 to 14 generate
         -- first stage is composed of half adders
         SA_HA : if i = 0 generate
            HA : HALFADDER
            port map( As(i)(j), Bs(i)(j+1), CO(i)(j), S(i)(j) );
         end generate SA_HA;
            -- all other stages composed of full adders
         SA_FA : if i > 0 generate
            FA : FULLADDER
            port map( As(i)(j), Bs(i)(j), CI(i)(j), CO(i)(j), S(i)(j) );
         end generate SA_FA;
      end generate SA;
   end generate SA_rows;

   -- determine partial products to be accumulated
   set_PP   : for i in 0 to 15 generate
      PP(i) <= B when A(i) = '1' else
               "0000000000000000";
   end generate;

   connect_gen : for i in 0 to 15 generate
      -- assign values to half adders in first stage of multiplier
      row0  : if i = 0 generate
         As(i) <= PP(i + 1);
         Bs(i) <= PP(i);
         -- assign LSB of product
         P(i)  <= Bs(i)(0);
      end generate row0;
      -- assign values to full adders in second stage of multiplier
      row1  : if i = 1 generate
         As(i)(14 downto 0)   <= As(i - 1)(15) & S(i - 1)(14 downto 1);
         Bs(i) <= PP(i + 1);
         CI(i) <= CO(i - 1);
         -- assign bit 1 of product
         P(i)  <= S(i - 1)(0);
      end generate row1;
      -- assign values to full adders in second to fourteenth stages of multiplier
      row2to14 : if i > 1 and i < 15 generate
         As(i)(14 downto 0)   <= Bs(i - 1)(15) & S(i - 1)(14 downto 1);
         Bs(i) <= PP(i + 1);
         CI(i) <= CO(i - 1);
         -- assign bits 2 to 14 of product
         P(i)  <= S(i - 1)(0);
      end generate row2to14;
```

```
      -- assign values to last stage or multiplier
      row15 : if i = 15 generate
         As(i)(14 downto 0)   <= Bs(i - 1)(15) & S(i - 1)(14 downto 1);
         Bs(i)(14 downto 0)   <= CO(i)(13 downto 0) & '0';
         CI(i) <= CO(i - 1);
         -- assign bit 15 of product
         P(i)  <= S(i - 1)(0);
         -- assign bits 16 to 31 of product
         P(31 downto 16)   <= CO(i)(14) & S(i)(14 downto 0);
      end generate row15;
   end generate connect_gen;
end behavioral;
```

BOOTH_MULT_16BIT.vhd

```
----------------------------------------------------------------------------------
-- 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;
```

MULT18X18.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      MULT18x18 Multiplier
-- Module Name:      MULT18X18 - behavioral
-- Project Name:     senior_project
-- Description:      MULT18x18 Multiplier accepts two 16-bit inputs and
--                   computes their 32-bit product using the Spartan 3E
--                   dedicated multiplier.
----------------------------------------------------------------------------------
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 MULT18X18 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 MULT18X18;

architecture behavioral of MULT18X18 is

begin
   P  <= A * B;
```

```
end behavioral;
```

RC_ADDER_32BIT.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      32-bit Ripple-Carry Adder
-- Module Name:      RC_ADDER_16BIT - behavioral
-- Project Name:     senior_project
-- Description:      32-bit Ripple-Carry Adder accepts two 32-bit inputs
--                   and calculates their sum by using ripple-carry method.
----------------------------------------------------------------------------------
library UNISIM;
use UNISIM.Vcomponents.all;

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 RC_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 RC_ADDER_32BIT;

architecture behavioral of RC_ADDER_32BIT is
-- internal carries generated by full adders
signal CO_INT  : std_logic_vector(31 downto 0) := (others => '0');
-- temp sum register
signal S_INT   : std_logic_vector(31 downto 0) := (others => '0');
-- sum corrected for overflow
signal S_ADJ   : std_logic_vector(31 downto 0) := (others => '0');

begin
   CO <= CO_INT(31);
   S  <= S_ADJ;

   -- generate adders in ripple-carry format
   RC : for i in 0 to 31 generate
      -- generate half adder for bit 0 of sum
      RC0   : if i = 0 generate
         HA : HALFADDER
         port map( A(i), B(i), CO_INT(i), S_INT(i) );
      end generate RC0;

      -- generate full adders for bits 1 to 31 of sum
      RC1to31  : if i > 0 generate
         FA : FULLADDER
         port map( A(i), B(i), CO_INT(i - 1), CO_INT(i), S_INT(i) );
      end generate RC1to31;
   end generate RC;

   -- overflow correction
   OV : OVERFLOW
   port map( A(31), B(31), S_INT, S_ADJ );
```

```
end behavioral;
```

CLa_ADDER_32BIT.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      32-bit Carry-Lookahead Adder
-- Module Name:      CLA_ADDER_32BIT - behavioral
-- Project Name:     senior_project
-- Description:      32-bit Carry-Lookahead Adder accepts two 32-bit inputs
--                   and calculates their sum by using carry-lookahead logic.
----------------------------------------------------------------------------------
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 CLa_ADDER_32BIT is
   port( A  : in  std_logic_vector(31 downto 0);
         B  : in  std_logic_vector(31 downto 0);
         CO : out std_logic;
         ADD_S : out std_logic_vector(31 downto 0) );
end CLa_ADDER_32BIT;

architecture behavioral of CLa_ADDER_32BIT is
-- carry, generate, and propagate bits
signal CI, G, P   : std_logic_vector(31 downto 0) := (others => '0');
signal PG, GG  : std_logic_vector(9 downto 0) := (others => '0');
signal GCI  : std_logic_vector(10 downto 0) := (others => '0');
signal S_INT   : std_logic_vector(31 downto 0) := (others => '0');
signal S_ADJ   : std_logic_vector(31 downto 0) := (others => '0');

begin
   ADD_S <= S_ADJ;
-- S  <= S_INT;
   GCI(0)   <= '0';
   GCI(8)   <= '0';
   GCI(10)  <= '0';

   -- generate partial full adders
   RC_gp : for i in 0 to 31 generate
      FA_gp : PARTIAL_FA
      port map( A(i), B(i), CI(i), G(i), P(i), S_INT(i) );
   end generate RC_gp;

   -- generate 4-bit CLA logic units
   CLa_lower   : for i in 1 to 8 generate
      CLa_0to7 : CLa_4BIT
      port map( G(4 * i - 1 downto 4 * i - 4), P(4 * i - 1 downto 4 * i - 4),
                GCI(i - 1), CI(4 * i - 1 downto 4 * i - 4),
                PG(i - 1), GG(i - 1) );
   end generate CLa_lower;

   CLa_upper0  : CLa_4BIT
   port map( GG(3 downto 0), PG(3 downto 0),
             GCI(8), GCI(3 downto 0),
             PG(8), GG(8) );
```

```
   CLa_upper1  : CLa_4BIT
   port map( GG(7 downto 4), PG(7 downto 4),
             GCI(9), GCI(7 downto 4),
             PG(9), GG(9) );

   -- CLa_top (2-bit Carry-Lookahead)
   GCI(9)   <= (GCI(10) and PG(8)) or GG(8);

   CO <= (GCI(10) and PG(8) and PG(9)) or
         (GG(8) and PG(9)) or GG(9);

   OV : OVERFLOW
   port map( A(31), B(31), S_INT, S_ADJ );
end behavioral;
```

CSe_ADDER_32BIT.vhd

```
----------------------------------------------------------------------------------
-- 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;
use work.CTRL_CONSTANTS.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;
```

CSa_ACCUM_32BIT.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      32-bit Carry-Save Acculator
-- Module Name:      SA_MULT_16BIT - behavioral
-- Project Name:     senior_project
-- Description:      32-bit Carry-Save Acculator accepts a 32-bit by 20 register
--                   and returns the accumulated 32-bit 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;
use work.CTRL_CONSTANTS.all;

entity CSa_ACCUM_32BIT is
   port( PART_S   : in  vect_32x20;
         ADD_CO         : out std_logic;
         ADD_S       : out std_logic_vector(31 downto 0) );
end CSa_ACCUM_32BIT;
```

```
architecture behavioral of CSa_ACCUM_32BIT is
-- internal carry-ins generated at each stage of accumulation
signal CI   : vect_32x20 := (others => (others => '0'));
-- internal carry-outs generated at each stage of accumulation
signal CO   : vect_32x20 := (others => (others => '0'));
-- unadjusted sum generated at each stage of accumulation
signal S : vect_32x20 := (others => (others => '0'));
-- sum that has been adjusted for overflow
signal S_ADJ   : std_logic_vector(31 downto 0) := (others => '0');
-- inputs to each stage of the accumulator
signal As   : vect_32x20 := (others => (others => '0'));
signal Bs   : vect_32x20 := (others => (others => '0'));

begin
   -- assign output sum with sum that has been adjusted for overflow
   ADD_S <= S_ADJ;
   -- assign the carry-out bit of accumulator
   ADD_CO   <= CO(F_LENGTH)(31);

   -- generate stages of accumulator
   SA_rows  : for i in 0 to F_LENGTH generate
      SA : for j in 0 to 31 generate
         -- first stage is composed of half adders
         SA_HA : if i = 0 generate
            HA : HALFADDER
            port map( As(i)(j), Bs(i)(j), CO(i)(j), S(i)(j) );
         end generate SA_HA;
         -- all other stages composed of full adders
         SA_FA : if i > 0 generate
            FA : FULLADDER
            port map( As(i)(j), Bs(i)(j), CI(i)(j),
                      CO(i)(j), S(i)(j) );
         end generate SA_FA;
      end generate SA;
   end generate SA_rows;

   connect_gen : for i in 0 to F_LENGTH generate
      -- assign values to half adders in first stage of accumulator
      row0  : if i = 0 generate
         As(i) <= PART_S(i + 1);
         Bs(i) <= PART_S(i);
      end generate row0;

      -- assign values to full adders in stages 2 to F_LENGTH - 1 of accumulator
      row1to18 : if i > 0 and i < F_LENGTH generate
         As(i) <= PART_S(i + 1);
         Bs(i) <= S(i - 1);
         CI(i) <= CO(i - 1)(30 downto 0) & '0';
      end generate row1to18;

      -- assign values to last stage of accumulator
      row19 : if i = F_LENGTH generate
         As(i) <= CO(i - 1)(30 downto 0) & '0';
         Bs(i) <= S(i - 1);
         carry_chain : for j in 0 to 31 generate
            carry0   : if j = 0 generate
               CI(i)(j) <= '0';
            end generate carry0;
            carry1to31  : if j > 0 generate
               CI(i)(j) <= CO(i)(j - 1);
            end generate carry1to31;
         end generate carry_chain;

         -- adjust sum for overflow
```

```
         OV19  : OVERFLOW
         port map( As(i)(31), Bs(i)(31), S(i), S_ADJ );
      end generate row19;
   end generate connect_gen;
end behavioral;
```

OVERFLOW.vhd

```
----------------------------------------------------------------------------------
-- 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;
use work.CTRL_CONSTANTS.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;
```

HALFADDER.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      1-bit Half Adder
-- Module Name:      HALFADDER - behavioral
-- Project Name:     senior_project
-- Description:      1-bit half adder that accepts two 1-bit numbers and returns
--                   their sum and carry 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;
use work.CTRL_CONSTANTS.all;

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

architecture behavioral of HALFADDER is

-- Half Adder
begin
-- assign values to the sum bit and carry-out bit
   CO <= A and B;
   S  <= A xor B;
end behavioral;
```

FULLADDER.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      1-bit Full Adder
-- Module Name:      FULLADDER - behavioral
-- Project Name:     senior_project
-- Description:      1-bit full adder that accepts two 1-bit numbers and a carry
--                   and returns their 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;
use work.CTRL_CONSTANTS.all;

entity FULLADDER is
   port( A  : in  std_logic;
         B  : in  std_logic;
         CI : in  std_logic;
         CO : out std_logic;
```

```
         S  : out std_logic );
end FULLADDER;

architecture behavioral of FULLADDER is

-- Full Adder
begin
-- assign values to the sum bit and carry-out bit
   S  <= A xor B xor CI;
   CO <= ((A or B) and CI) or (A and B);
end behavioral;
```

PARTIAL_FA.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      1-bit Full Adder w/ generate and propagate bits
-- Module Name:      PARTIAL_FA - behavioral
-- Project Name:     senior_project
-- Description:      1-bit full adder that accepts two 1-bit inputs and returns
--                   their sum, generate and propagate bits.
----------------------------------------------------------------------------------
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 PARTIAL_FA is
   port( A  : in  std_logic;
         B  : in  std_logic;
         CI : in  std_logic;
         G  : out std_logic;
         P  : out std_logic;
         S  : out std_logic);
end PARTIAL_FA;

architecture behavioral of PARTIAL_FA is

-- Full Adder
begin
-- assign values to the sum, generate, and propagate output bits
   G  <= A and B;
   P  <= A xor B;
   S  <= A xor B xor CI;
end behavioral;
```

RC_ADDER_4BIT.vhd

```
----------------------------------------------------------------------------------
-- 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;
use work.CTRL_CONSTANTS.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   : FULLADDER
         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   : FULLADDER
         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;
```

CLa_4BIT.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      4-bit Carry-Lookahead Logic
-- Module Name:      CLA_4BIT - behavioral
-- Project Name:     senior_project
-- Description:      4-bit Carry-Lookahead Logic performs carry lookahead logic
--                   based on generate and propagate 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;
use work.CTRL_CONSTANTS.all;

entity CLa_4BIT is
   port( G  : in  std_logic_vector(3 downto 0);
         P  : in  std_logic_vector(3 downto 0);
         CI : in  std_logic;
         C  : out std_logic_vector(3 downto 0);
         PG : out std_logic;
         GG : out std_logic );
end CLa_4BIT;

architecture behavioral of CLa_4BIT is

-- 4-bit CLA logic
begin
   C(0)  <= CI;

   C(1)  <= (CI and P(0)) or G(0);

   C(2)  <= (CI and P(0) and P(1)) or
            (G(0) and P(1)) or G(1);

   C(3)  <= (CI and P(0) and P(1) and P(2)) or
            (G(0) and P(1) and P(2)) or
            (G(1) and P(2)) or G(2);

   GG <= (G(0) and P(1) and P(2) and P(3)) or
         (G(1) and P(2) and P(3)) or
         (G(2) and P(3)) or G(3);

   PG <= P(0) and P(1) and P(2) and P(3);
end behavioral;
```

SIPO_SHR.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      Serial-in, Parallel-out Shift Register
-- Module Name:      SIPO_SHR - behavioral
-- Project Name:     senior_project
-- Description:      Serial-in, Parallel-out Shift Register accepts a clock,
--                   clear and preset inputs, and a 16-bit signal. It returns a
--                   16-bit by 20 register containing stored input values that
--                   have been shifted one index value away from 0 on each
--                   positive clock edge.
----------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library work;
use work.DEFINITIONS.all;

entity SIPO_SHR is
   port( CLK   : in  std_logic;
         CLR   : in  std_logic;
         PRE   : in  std_logic;
         D     : in  std_logic_vector(15 downto 0);
         Q     : out vect_16x20 );
end SIPO_SHR;

architecture behavioral of SIPO_SHR is
-- temporary signal to store values for shifting
signal Q_TMP   : vect_16x20 := (others => (others => '0'));

-- SIPO Shift Register
begin
   -- set output
   Q  <= Q_TMP;

   -- process shifts register values one index place away from 0
   -- on each positive clock edge
   shift : process ( CLK, CLR, PRE, D ) is
   variable STARTUP  : std_logic := '0';
   begin
      if STARTUP = '1' then
         for i in 0 to 19 loop
            Q_TMP(i) <= "0000000000000000";
         end loop;
         STARTUP  := '0';
      -- asynchronous reset
      elsif CLR = '1' then
         for i in 0 to 19 loop
            Q_TMP(i) <= "0000000000000000";
         end loop;
      -- asynchronous preset
      elsif PRE = '1' then
         for i in 0 to 19 loop
            Q_TMP(i) <= "1111111111111111";
         end loop;
      -- shift register values
      elsif rising_edge(CLK) then
         Q_TMP(1 to 19) <= Q_TMP(0 to 18);
         Q_TMP(0) <= D;
      end if;
```

```
   end process shift;
end behavioral;
```

CLK_DIV.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      Clock Divider
-- Module Name:      CLK_DIV - behavioral
-- Project Name:     senior_project
-- Description:      Clock Divider accepts a clock signal, divides it by the
--                   integer DIV, and returns the divided clock signal.
----------------------------------------------------------------------------------
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CLK_DIV is
   port( DIV   : in  integer;
         CLK   : in  std_logic;
         SCLK  : out std_logic );
end CLK_DIV;

architecture behavioral of CLK_DIV is

-- Clock DIvider
begin
   -- process divides clock by the input integer DIV
   clock :  process(CLK) is
   -- variables to keep track of clock dividing and temporary clock signal
   variable DIV_CNT  :  integer := 0;
   variable SCLK_TMP : std_logic := '0';

   begin
      if rising_edge(CLK) then
         if DIV_CNT < DIV - 1 then
            -- increment DIV_CNT if dividing not done
            DIV_CNT  := DIV_CNT + 1;
         elsif DIV_CNT = DIV - 1 then
            -- negate temporary clock signal if dividing done
            SCLK_TMP := not(SCLK_TMP);
            -- reset counter
            DIV_CNT  := 0;
         end if;

         -- set output clock signal
         SCLK  <= SCLK_TMP;
      end if;
   end process clock;
end behavioral;
```

DEFINITIONS.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      User Definitions
-- Module Name:      DEFINITIONS - package
-- Project Name:     senior_project
-- Description:      DEFINITIONS contains all component declarations used
--                   throughout the project.
----------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package DEFINITIONS is
   type int_x20 is array (0 to 19) of integer;
   type int_x30 is array (0 to 29) of integer;
   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);
   type vect_32x20 is array (0 to 19) of std_logic_vector(31 downto 0);
   type vect_32x30 is array (0 to 29) of std_logic_vector(31 downto 0);

   component HALFADDER is
      port ( A    : in  std_logic;
             B    : in  std_logic;
             CO   : out std_logic;
             S    : out std_logic );
   end component;

   component FULLADDER 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 PARTIAL_FA is
      port( A  : in  std_logic;
            B  : in  std_logic;
            CI : in  std_logic;
            G  : out std_logic;
            P  : out std_logic;
            S  : out std_logic );
   end component;

   component RC_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 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 CLa_4BIT is
      port( G  : in  std_logic_vector(3 downto 0);
            P  : in  std_logic_vector(3 downto 0);
```

```
         CI : in  std_logic;
         C  : out std_logic_vector(3 downto 0);
         PG : out std_logic;
         GG : out std_logic );
end component;

component CLa_ADDER_32BIT is
   port( A     : in  std_logic_vector(31 downto 0);
         B     : in  std_logic_vector(31 downto 0);
         CO    : out std_logic;
         ADD_S : out std_logic_vector(31 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 CSa_ACCUM_32BIT is
   port( PART_S   : in  vect_32x20;
         ADD_CO   : out std_logic;
         ADD_S    : 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;

component SA_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 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 MULT18X18 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 CLK_DIV is
   port( DIV   :  in    integer;
         CLK   :  in    std_logic;
         SCLK  :  out   std_logic );
end component;

component SIPO_SHR is
   port( CLK   : in  std_logic;
         CLR   : in  std_logic;
         PRE   : in  std_logic;
         D     : in  std_logic_vector(15 downto 0);
         Q     : out vect_16x20 );
end component;
```

```
component CONVERTER_CTRL is
   port( CLK         :  in    std_logic;
         RST         :  in    std_logic;
         RD_EN       :  in    std_logic;
         WRT_EN      :  in    std_logic;
         MISO        :  in    std_logic;
         Yn          :  in    std_logic_vector(11 downto 0);
         ADC_SCLK    :  out   std_logic;
         DAC_SCLK    :  out   std_logic;
         CS          :  out   std_logic;
         SYNC        :  out   std_logic;
         MOSI        :  out   std_logic;
         Xn          :  out   std_logic_vector(11 downto 0);
         SAMP_DONE   :  out   std_logic );
end component;

component SAMPLE_CTRL is
   port( CLK         : in  std_logic;
         RST         : in  std_logic;
         EN          : in  std_logic;
         MISO        : in  std_logic;
         Yn          : in  std_logic_vector(11 downto 0);
         OUTPUT_DONE : in  std_logic;
         ADC_SCLK    : out std_logic;
         DAC_SCLK    : out std_logic;
         CS          : out std_logic;
         SYNC        : out std_logic;
         MOSI        : out std_logic;
         Xn          : out std_logic_vector(11 downto 0);
         XnmM        : out vect_16x20;
         YnmN        : out vect_16x20;
         SAMP_DONE   : out std_logic );
end component;

component DSP_BB is
   port( MULT_A      : in vect_16x20;
         MULT_B      : in vect_16x20;
         ADD_A : in vect_32x20;
         ADD_B : in vect_32x20;
         PART_S   : in vect_32x20;
         P           : out vect_32x20;
         S           : out vect_32x20;
         As    : out vect_16x20;
         Bs    : out vect_16x30;
         C_VECT   : out std_logic_vector(15 downto 0) );
end component;

component NORMAL is
   port( CLK         : in  std_logic;
         RST         : in  std_logic;
         EN          : in  std_logic;
         SAMP_DONE   : in  std_logic;
         XnmM        : in  vect_16x20;
         YnmN        : in  vect_16x20;
         As          : in  vect_16x20;
         Bs          : in  vect_16x30;
         C_VECT      : in  std_logic_vector(15 downto 0);
         P           : in vect_32x20;
         S           : in vect_32x20;
         MULT_B      : out vect_16x20;
         MULT_A      : out vect_16x20;
         ADD_A : out vect_32x20;
         ADD_B : out vect_32x20;
```

```
            PART_S   : out vect_32x20 ;
            Yn          : out std_logic_vector(11 downto 0);
            OUTPUT_DONE : out std_logic );
   end component;

   component CASCADE is
      port( CLK         : in  std_logic;
            RST         : in  std_logic;
            EN          : in  std_logic;
            SAMP_DONE   : in  std_logic;
            XnmM        : in  vect_16x20;
            YnmN        : in  vect_16x20;
            As          : in  vect_16x20;
            Bs          : in  vect_16x30;
            C_VECT      : in  std_logic_vector(15 downto 0);
            P           : in vect_32x20;
            S           : in vect_32x20;
            MULT_B      : out vect_16x20;
            MULT_A      : out vect_16x20;
            ADD_A : out vect_32x20;
            ADD_B : out vect_32x20;
            PART_S   : out vect_32x20;
            Yn          : out std_logic_vector(11 downto 0);
            OUTPUT_DONE : out std_logic );
   end component;


   component CASC_DFI is
      port( CLK         : in  std_logic;
            RST         : in  std_logic;
            EN          : in  std_logic;
            SAMP_DONE   : in  std_logic;
            XnmM        : in  vect_16x20;
            YnmN        : in  vect_16x20;
            As          : in  vect_16x20;
            Bs          : in  vect_16x30;
            C_VECT      : in  std_logic_vector(15 downto 0);
            P           : in vect_32x20;
            S           : in vect_32x20;
            MULT_B      : out vect_16x20;
            MULT_A      : out vect_16x20;
            ADD_A : out vect_32x20;
            ADD_B : out vect_32x20;
            PART_S   : out vect_32x20;
            Yn          : out std_logic_vector(11 downto 0);
            OUTPUT_DONE : out std_logic );
   end component;
end package;
```

CTRL_CONSTANTS.vhd

```
----------------------------------------------------------------------------------
-- Company:          Cal Poly
-- Engineer:         Joseph Waddell
--
-- Design Name:      Control Constants
-- Module Name:      CTRL_CONSTANTS - package
-- Project Name:     senior_project
-- Description:      Control Constants contains all constants used to control
--                   various modules throughout the project.
----------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.DEFINITIONS.all;

package CTRL_CONSTANTS is
   -- STRUCTURE controls filter realization structure.
   --
   -- For Direct Form I realization set STRUCTURE = 0.
   -- For Direct Form II Cascade realization of Canonical 2nd order systems
   -- set STRUCTURE = 1.
   -- For Direct Form I Cascade realization of Canonical 2nd order systems
   -- set STRUCTURE = 2.
   constant STRUCTURE   : integer range 0 to 2 := 0;

   -- SorP is used to select serial or parallel operation of the filter.
   --
   -- For serial operation set SorP = 0.
   -- For parallel operation set SorP = 1.
   constant SorP        : integer range 0 to 1 := 0;

   -- SAMP_DIV is used to control sampling timing using the CLK_DIV module.
   -- It is used to divide the system clock to the desired sampling
   -- frequency. SAMP_DIV for a desired sampling frequency can be calculated
   -- by the following formula:
   --
   -- SAMP_DIV = 50,000,000 / ( 2 * Sampling Frequency )
   constant SAMP_DIV    : integer range 1 to 50000000 := 566;

   -- SCALE is the desired bit length used for signed integer scaling.
   --
   -- For no scaling set SCALE = 1
   constant SCALE       : integer range 1 to 16 := 1;

   -- MULTIPLIER allows for multiplier selection from the two available
   -- multipliers, the Shift-Add multiplier and the Spartan 3E dedicated
   -- MULT18X18 multiplier.
   --
   -- For the Shift-Add multiplier set MULTIPLIER = 0.
   -- For the Radix-4 Modified Booth multiplier set MULTIPLIER = 1.
   -- For the MULT18X18 multiplier set MULTIPLIER = 2.
   constant MULTIPLIER  : integer range 0 to 2 := 0;

   -- ADDER allows for adder selection during serial operation. The three
   -- available adders are the Ripple-Carry adder, the Carry-Select adder,
   -- and the Carry-Lookahead adder.
   --
   -- During parallel operation a Carry-Save accululator is used to
   -- facilitate parallel accumulation of filter terms.
   --
   -- For the Ripple-Carry adder set ADDER = 0.
   -- For the Carry-Select adder set ADDER = 1.
```

```
   -- For the Carry-Lookahead adder set ADDER = 2.
   constant ADDER       : integer range 0 to 2 := 0;

--------------------------------------------------------------------------------
-- Assign values to constants for Direct Form I realization.
--------------------------------------------------------------------------------
   -- Filter length is equal to N + M - 1.
   constant F_LENGTH    : integer range 0 to 19 := 0;

   -- N is the number of Ak terms as they appear on the RIGHT side of
   -- the difference equation.
   constant N           : integer range 0 to 19 := 0;
   -- Ak terms of the filter.
   constant Ak          : int_x20 := ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                       0, 0, 0, 0, 0, 0, 0, 0, 0, 0 );

   -- M is the number of Bk terms.
   constant M           : integer range 1 to 20 := 1;
--  Bk terms of the filter
   constant Bk          : int_x20 := ( 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                       0, 0, 0, 0, 0, 0, 0, 0, 0, 0 );

--------------------------------------------------------------------------------
-- Assign values to constants for Direct Form II Cascade realization.
--------------------------------------------------------------------------------
   -- This is the desired number of filter stages.
   constant F_STAGES : integer range 1 to 9 := 1;

   -- Ak terms as they appear on the RIGHT side of the difference equation
   -- for use with serial calculation implementation.
   constant Aki   : int_x30 := ( 0, 0,    -- stage 1
                                 0, 0,    -- stage 2
                                 0, 0,    -- stage 3
                                 0, 0,    -- stage 4
                                 0, 0,    -- stage 5
                                 0, 0,    -- stage 6
                                 0, 0,    -- stage 7
                                 0, 0,    -- stage 8
                                 0, 0,    -- stage 9
                                 ------------------------------------------------
                                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 );

   -- Bk terms for serial calculation implementation
   constant Bki_S : int_x30 := ( 0, 0, 0,    -- stage 1
                                 0, 0, 0,    -- stage 2
                                 0, 0, 0,    -- stage 3
                                 0, 0, 0,    -- stage 4
                                 0, 0, 0,    -- stage 5
                                 0, 0, 0,    -- stage 6
                                 0, 0, 0,    -- stage 7
                                 0, 0, 0,    -- stage 8
                                 0, 0, 0,    -- stage 9
                                 ------------------------------------------------
                                 0, 0, 0 );

   -- Constant for use in parallel calculation implementation
   constant C  : integer := 0;

   -- Bk terms for parallel calculation implementation
   constant Bki_P : int_x20 := ( 0, 0,    -- stage 1
                                 0, 0,    -- stage 2
                                 0, 0,    -- stage 3
                                 0, 0,    -- stage 4
                                 0, 0,    -- stage 5
```

```
                                0, 0,    -- stage 6
                                0, 0,    -- stage 7
                                0, 0,    -- stage 8
                                0, 0,    -- stage 9
                                ------------------------------------------------
                                0, 0 );

   -- Bk terms for use with CASC_DFI.vhd module. These coefficients pair
   -- with the Ak terms contained in Aki for this module.
   constant Bki   : int_x30 := ( 0, 0, 0,    -- stage 1
                                0, 0, 0,    -- stage 2
                                0, 0, 0,    -- stage 3
                                0, 0, 0,    -- stage 4
                                0, 0, 0,    -- stage 5
                                0, 0, 0,    -- stage 6
                                0, 0, 0,    -- stage 7
                                0, 0, 0,    -- stage 8
                                0, 0, 0,    -- stage 9
                                ------------------------------------------------
                                0, 0, 0 );
end package;
```

Appendix D: VHDL Testbenches

This appendix contains the VHDL testbenches used to test project modules.

FILT.vhd

```
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 FILT is
end FILT;

-- NOTE -  Some component entities may need slight alterations to
-- accomidate some test signals.
architecture behavioral of FILT is
   component NORMAL is
      port( CLK         : in  std_logic;
            RST         : in  std_logic;
            EN          : in  std_logic;
            SAMP_DONE   : in  std_logic;
            XnmM        : in  vect_16x20;
            YnmN        : in  vect_16x20;
            As          : in  vect_16x20;
            Bs          : in  vect_16x30;
            C_VECT      : in  std_logic_vector(15 downto 0);
            P           : in vect_32x20;
            S           : in vect_32x20;
            MULT_B      : out vect_16x20;
            MULT_A      : out vect_16x20;
            ADD_A       : out vect_32x20;
            ADD_B       : out vect_32x20;
            PART_S      : out vect_32x20 ;
            Yn          : out std_logic_vector(11 downto 0);
            OUTPUT_DONE : out std_logic;
            AVERAGE_tst : out std_logic_vector(15 downto 0);
         SC_OFF_tst     : out std_logic_vector(31 downto 0);
         PRODUCT_tst    :  out vect_32x20;
         SUM_tst        : out std_logic_vector(31 downto 0));
   end component;

   component CASCADE is
      port( CLK               : in  std_logic;
            RST               : in  std_logic;
            EN                : in  std_logic;
            SAMP_DONE         : in  std_logic;
            XnmM              : in  vect_16x20;
            YnmN              : in  vect_16x20;
            As                : in  vect_16x20;
            Bs                : in  vect_16x30;
            C_VECT            : in  std_logic_vector(15 downto 0);
            P                 : in vect_32x20;
            S                 : in vect_32x20;
            MULT_B            : out vect_16x20;
            MULT_A            : out vect_16x20;
            ADD_A             : out vect_32x20;
            ADD_B             : out vect_32x20;
            PART_S            : out vect_32x20;
            Yn                : out std_logic_vector(11 downto 0);
```

```
            OUTPUT_DONE       : out std_logic;
            SUM_tst           : out vect_32x20;
            MULT_B_TERMS_tst  : out integer range 0 to 19;
            MULT_A_TERMS_tst  : out integer range 0 to 19;
            CNT_TERMS_tst     : out integer range 0 to 19;
            PRODUCT_tst       : out vect_32x20;
            Wn_tst            : out vect_16x20;
            Wi_tst            : out vect_16x30;
            Yi_tst            : out vect_16x20;
            CNT_STAGE_tst     : out integer range 0 to 3 );
    end component;


    component CASC_DFI is
       port( CLK            : in  std_logic;
             RST            : in  std_logic;
             EN             : in  std_logic;
             SAMP_DONE      : in  std_logic;
             XnmM           : in  vect_16x20;
             YnmN           : in  vect_16x20;
             As             : in  vect_16x20;
             Bs             : in  vect_16x30;
             C_VECT         : in  std_logic_vector(15 downto 0);
             P              : in vect_32x20;
             S              : in vect_32x20;
             MULT_B         : out vect_16x20;
             MULT_A         : out vect_16x20;
             ADD_A          : out vect_32x20;
             ADD_B          : out vect_32x20;
             PART_S         : out vect_32x20;
             Yn             : out std_logic_vector(11 downto 0);
             OUTPUT_DONE    : out std_logic;
             AVERAGE_tst    : out std_logic_vector(15 downto 0);
             SC_OFF_tst     : out std_logic_vector(31 downto 0);
             PRODUCT_tst    :  out vect_32x20;
             SUM_tst        : out std_logic_vector(31 downto 0);
             CNT_STG_tst    : out integer range 0 to 20;
             Xi_tst         : out vect_16x20;
             Yi_tst         : out vect_16x20;
             CNT_TERMS_tst  : out integer range 0 to 20;
             STG_DONE_tst   : out std_logic );
    end component;


signal CLK  : std_logic := '0';
signal RST  : std_logic := '0';
signal EN   : std_logic := '0';
signal MISO : std_logic := '0';
signal ADC_SCLK   : std_logic := '0';
signal DAC_SCLK   : std_logic := '0';
signal CS   : std_logic := '0';
signal SYNC : std_logic := '0';
signal MOSI : std_logic := '0';
signal LED  : std_logic_vector(11 downto 0) := (others => '0');

signal BB_STATE   : std_logic_vector(2 downto 0) := "000";

signal Bs_tst  : vect_16x30 := (others => (others =>'0'));
signal As_tst  : vect_16x20 := (others => (others =>'0'));

signal MULT_B_TERMS_tst : integer range 0 to 20 := 0;
signal MULT_A_TERMS_tst : integer range 0 to 20 := 0;
signal CNT_TERMS_tst : integer range 0 to 20 := 0;
signal CNT_STG_tst   : integer range 0 to 3 := 0;
```

```
signal MULT_B_tst : vect_16x20 := (others => (others => '0'));
signal MULT_A_tst : vect_16x20 := (others => (others =>  '0'));
signal P_tst   : vect_32x20 := (others => (others => '0'));
signal PRODUCT_tst   : vect_32x20 := (others => (others => '0'));
signal PART_S_tst : vect_32x20 := (others => (others => '0'));

signal ADD_A_tst  : vect_32x20 := (others => (others => '0'));
signal ADD_B_tst  : vect_32x20 := (others => (others => '0'));
signal S_tst   : vect_32x20 := (others => (others => '0'));
signal d_SUM_tst  :  std_logic_vector(31 downto 0) := (others => '0');

signal SUM_tst :  vect_32x20 := (others => (others => '0'));

signal AVERAGE_tst   : std_logic_vector(15 downto 0) := (others => '0');

signal Wn_tst  :  vect_16x20 := (others => (others => '0'));
signal Xi_tst  :  vect_16x20 := (others => (others => '0'));
signal Yi_tst  :  vect_16x20 := (others => (others => '0'));
signal Wi_tst  :  vect_16x30 := (others => (others => '0'));
signal Xn   :  std_logic_vector(11 downto 0) := (others => '0');
signal Yn   :  std_logic_vector(11 downto 0) := (others => '0');
signal DBC_CLK :  std_logic := '0';
signal RST_DB  :  std_logic := '0';
signal EN_DB   :  std_logic := '0';
signal RST_DB_REG :  std_logic_vector(7 downto 0) := (others => '0');
signal EN_DB_REG  :  std_logic_vector(7 downto 0) := (others => '0');
signal XnmM : vect_16x20 := (others => (others => '0'));
signal YnmN : vect_16x20 := (others => (others => '0'));
signal SAMP_DONE  :  std_logic := '0';
signal STG_DONE_tst  :  std_logic := '0';
signal OUTPUT_DONE   :  std_logic := '0';

signal SAMP_CLK_tst  : std_logic := '0';
signal GET_SAMP_tst  : std_logic := '0';

signal SAMP_STATE : std_logic_vector(2 downto 0) := "000";
signal DIR_STATE  : std_logic_vector(2 downto 0) := "000";

signal C_VECT_tst :  std_logic_vector(15 downto 0) := (others => '0');
signal SC_OFF_tst : std_logic_vector(31 downto 0) := (others => '0');


begin
   dir   : if STRUCTURE = 0 generate
      d_filter : NORMAL port map( CLK, RST_DB, EN_DB, SAMP_DONE, XnmM, YnmN, As_tst, Bs_tst,
C_VECT_tst, P_tst,
                                  S_tst, MULT_B_tst, MULT_A_tst, ADD_A_tst, ADD_B_tst,
PART_S_tst, Yn,
                                  OUTPUT_DONE, AVERAGE_tst, SC_OFF_tst, PRODUCT_tst, d_SUM_tst );
   end generate dir;

   casc  : if STRUCTURE = 1 generate
      c_filter : CASCADE port map( CLK, RST_DB, EN_DB, SAMP_DONE, XnmM, YnmN, As_tst, Bs_tst,
C_VECT_tst, P_tst,
                                   S_tst, MULT_B_tst, MULT_A_tst, ADD_A_tst, ADD_B_tst,
PART_S_tst, Yn, OUTPUT_DONE,
                                    SUM_tst,  MULT_B_TERMS_tst, MULT_A_TERMS_tst, CNT_TERMS_tst,
PRODUCT_tst,
                                    Wn_tst, Wi_tst, Yi_tst, CNT_STG_tst );
   end generate casc;

-- casc_dir : if STRUCTURE = 2 generate
```

```
--    nc_filter   : CASC_DFI port map( CLK, RST_DB, EN_DB, SAMP_DONE, XnmM, YnmN, As_tst, Bs_tst,
C_VECT_tst, P_tst,
--                                  S_tst, MULT_B_tst, MULT_A_tst, ADD_A_tst, ADD_B_tst,
PART_S_tst, Yn,
--                                  OUTPUT_DONE, AVERAGE_tst, SC_OFF_tst, PRODUCT_tst, d_SUM_tst,
CNT_STG_tst, Xi_tst,
--                                  Yi_tst, BB_STATE, CNT_TERMS_tst, STG_DONE_tst );
-- end generate casc_dir;



   bb       : DSP_BB port map( MULT_A_tst, MULT_B_tst, ADD_A_tst, ADD_B_tst, PART_S_tst, P_tst,
S_tst, As_tst, Bs_tst, C_VECT_tst );

   sample   : SAMPLE_CTRL port map( CLK, RST_DB, EN_DB, MISO, Yn, OUTPUT_DONE,
                                    ADC_SCLK, DAC_SCLK, CS, SYNC, MOSI,
                                    Xn, XnmM, YnmN, SAMP_DONE, SAMP_CLK_tst, GET_SAMP_tst,
SAMP_STATE );

   debounce_clk   :  CLK_DIV port map( 2, CLK, DBC_CLK );

   input_proc  : process is
   variable FIRST : std_logic := '1';
   begin
---- max value input
--    wait for 2190 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--
--    loop
--       wait for 2800 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
```

```
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--    end loop;

---- one volt input
--    wait for 2190 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '1';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '1';
--
--    loop
```

```
--       wait for 2800 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--    end loop;

---- 0000000000000001 input
--    wait for 2190 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '0';
--    wait for 80 ns;
```

```
--    MISO  <= '0';
--    wait for 80 ns;
--    MISO  <= '1';
--
--    loop
--       wait for 2800 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--    end loop;

-- incrementing input
   wait for 2190 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
   wait for 80 ns;
   MISO  <= '0';
```

```
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '1';

   loop
      if FIRST = '0' then
         wait for 2800 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '1';
      end if;

      wait for 2800 ns;
      MISO  <= '0';
      wait for 80 ns;
      MISO  <= '0';
      wait for 80 ns;
      MISO  <= '0';
      wait for 80 ns;
      MISO  <= '0';
      wait for 80 ns;
      MISO  <= '0';
      wait for 80 ns;
      MISO  <= '0';
      wait for 80 ns;
      MISO  <= '0';
      wait for 80 ns;
      MISO  <= '0';
      wait for 80 ns;
      MISO  <= '0';
      wait for 80 ns;
```

```
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '1';
wait for 80 ns;
MISO  <= '0';

wait for 2800 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '1';
wait for 80 ns;
MISO  <= '1';

wait for 2800 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
```

```
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '1';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';

wait for 2800 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '1';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '1';

wait for 2800 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
MISO  <= '0';
wait for 80 ns;
```

```
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '0';
         wait for 80 ns;
         MISO  <= '1';
         wait for 80 ns;
         MISO  <= '1';
         wait for 80 ns;
         MISO  <= '0';

         FIRST := '0';
      end loop;

---- sine input
-- wait for 2190 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '1';
-- wait for 80 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '1';
-- wait for 80 ns;
-- MISO  <= '1';
-- wait for 80 ns;
-- MISO  <= '1';
-- wait for 80 ns;
-- MISO  <= '1';
-- wait for 80 ns;
-- MISO  <= '0';
-- wait for 80 ns;
-- MISO  <= '1';
-- wait for 80 ns;
-- MISO  <= '1';
--
--    loop
--       if FIRST = '0' then
--          wait for 2800 ns;
--          MISO  <= '0';
--          wait for 80 ns;
--          MISO  <= '0';
--          wait for 80 ns;
--          MISO  <= '0';
--          wait for 80 ns;
```

```
--          MISO  <= '0';
--          wait for 80 ns;
--          MISO  <= '1';
--          wait for 80 ns;
--          MISO  <= '0';
--          wait for 80 ns;
--          MISO  <= '0';
--          wait for 80 ns;
--          MISO  <= '0';
--          wait for 80 ns;
--          MISO  <= '0';
--          wait for 80 ns;
--          MISO  <= '1';
--          wait for 80 ns;
--          MISO  <= '1';
--          wait for 80 ns;
--          MISO  <= '1';
--          wait for 80 ns;
--          MISO  <= '1';
--          wait for 80 ns;
--          MISO  <= '0';
--          wait for 80 ns;
--          MISO  <= '1';
--          wait for 80 ns;
--          MISO  <= '1';
--       end if;

--       wait for 2800 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--
--       wait for 2800 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
```

```
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--
--       wait for 2800 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--
--       wait for 2800 ns;
--       MISO  <= '0';
```

```
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--
--       wait for 2800 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '0';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '1';
--       wait for 80 ns;
--       MISO  <= '0';
--
```

```
--       FIRST := '0';
--    end loop;

   end process input_proc;

   en_proc  : process is
   begin
      wait for 100 ns;
      EN <= '1';
   end process en_proc;

   clock_proc  : process is
   begin
      wait for 10 ns;
      CLK   <= not(CLK);
      wait for 10 ns;
      CLK   <= not(CLK);
   end process clock_proc;

   -- debounce process for button inputs
   debounce_proc : process ( CLK, DBC_CLK ) is
   begin
      if rising_edge(DBC_CLK) then
         RST_DB_REG(7 downto 1)  <= RST_DB_REG(6 downto 0);
         RST_DB_REG(0)  <= RST;

         EN_DB_REG(7 downto 1)   <= EN_DB_REG(6 downto 0);
         EN_DB_REG(0)   <= EN;
      end if;

      if rising_edge(CLK) then
         if EN_DB_REG = "11111111" then
            EN_DB <= '1';
         elsif EN_DB_REG = "00000000" then
            EN_DB <= '0';
         end if;

         if RST_DB_REG = "11111111" then
            RST_DB   <= '1';
         elsif RST_DB_REG = "00000000" then
            RST_DB   <= '0';
         end if;
      end if;
   end process debounce_proc;
end behavioral;
```

ADD_TB.vhd

```
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 ADD_TB IS
END ADD_TB;

ARCHITECTURE behavior OF ADD_TB IS

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

```
    COMPONENT RC_ADDER_32BIT
    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 CLa_ADDER_32BIT
    PORT(
         A : IN  std_logic_vector(31 downto 0);
         B : IN  std_logic_vector(31 downto 0);
         CO : OUT  std_logic;
         ADD_S : OUT  std_logic_vector(31 downto 0)
        );
    END COMPONENT;

    COMPONENT CSe_ADDER_32BIT
    PORT(
         A : IN  std_logic_vector(31 downto 0);
         B : IN  std_logic_vector(31 downto 0);
         CO : OUT  std_logic;
         ADD_S : OUT  std_logic_vector(31 downto 0)
        );
    END COMPONENT;

   COMPONENT CSa_ACCUM_32BIT is
      port( PART_S   : in  vect_32x20;
            ADD_CO         : out std_logic;
            ADD_S       : out std_logic_vector(31 downto 0) );
   end COMPONENT;

--  COMPONENT WRAPPER
--    PORT(
--       A  : in  std_logic_vector(31 downto 0);
--       B  : in  std_logic_vector(31 downto 0);
--       S  : out std_logic_vector(31 downto 0);
--       CO : out std_logic
--      );
--    END COMPONENT;

   COMPONENT WRAPPER is
      port( PART_S   : in  vect_32x20;
            ADD_S : out std_logic_vector(31 downto 0);
            ADD_CO   : out std_logic );
   end COMPONENT;


   --Inputs
--   signal A : std_logic_vector(31 downto 0) := (others => '0');
--   signal B : std_logic_vector(31 downto 0) := (others => '0');
-- signal S : std_logic_vector(31 downto 0);
   signal PART_S  : vect_32x20 := (others => (others => '0'));
-- signal CO : std_logic;
   signal ADD_S : std_logic_vector(31 downto 0);
   signal ADD_CO : std_logic;


--   signal A0 : std_logic_vector(31 downto 0) := (others => '0');
--   signal B0 : std_logic_vector(31 downto 0) := (others => '0');
-- signal S0 : std_logic_vector(31 downto 0);
-- signal CO0 : std_logic;
```

```
--
--   signal A1 : std_logic_vector(31 downto 0) := (others => '0');
--   signal B1 : std_logic_vector(31 downto 0) := (others => '0');
--   signal S1 : std_logic_vector(31 downto 0);
-- signal CO1 : std_logic;
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name

BEGIN

--  Instantiate the Unit Under Test (UUT)
--  uut: RC_ADDER_32BIT PORT MAP (
--          A => A,
--          B => B,
--          CO => CO,
--          S => S
--        );
--
-- uut: CSe_ADDER_32BIT PORT MAP (
--          A => A,
--          B => B,
--          CO => CO,
--          S => S
--        );
--
-- uut: CLa_ADDER_32BIT PORT MAP (
--          A => A,
--          B => B,
--          CO => CO,
--          S => S
--        );
   uut:  CSA_ACCUM_32BIT
      port map( PART_S, ADD_CO, ADD_S );

--uut: WRAPPER PORT MAP (
--         A,
--       B,
--       S,
--       CO
--        );
-- uut   : WRAPPER
--    port map( PART_S, ADD_S, ADD_CO );

--    A  <= "11111111111111111111111111111111";
--    B  <= "11111111111111111111111111111111";
--
----     wait for 20 ns;
--    A0 <= "01010101010100101010101010101010";
--    B0 <= "00101010101010101010101010101010";
--
----     wait for 20 ns;
--    A1 <= "01111111111111111111111111111111";
--    B1 <= "00000000000000000000000000000001";


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;


--       insert stimulus here
      PART_S(0)   <= "00000000000000000000000000000001";
```

```
      PART_S(1)   <= "00000000000000000000000000000001";
      PART_S(2)   <= "00000000000000000000000000000001";
      PART_S(3)   <= "00000000000000000000000000000001";
      PART_S(4)   <= "00000000000000000000000000000001";
      PART_S(5)   <= "00000000000000000000000000000001";
      PART_S(6)   <= "00000000000000000000000000000001";
      PART_S(7)   <= "00000000000000000000000000000001";
      PART_S(8)   <= "00000000000000000000000000000001";
      PART_S(9)   <= "00000000000000000000000000000001";
--

--    A  <= "11111111111111111111111111111111";
--    B  <= "11111111111111111111111111111111";
--
--    wait for 20 ns;
--    A0 <= "10101010101010101010101001010101";
--    B0 <= "10101010101010101010101010101010";
--
--    wait for 20 ns;
--    A1 <= "01111111111111111111111111111111";
--    B1 <= "00000000000000000000000000000001";

      wait;
   end process;

END;
```

MULT_TB.vhd

```
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 MULT_TB IS
END MULT_TB;

ARCHITECTURE behavior OF MULT_TB IS

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

    COMPONENT SA_MULT_16BIT
    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 BOOTH_MULT_16BIT
    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 MULT18X18
    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 WRAPPER
    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;

   --Inputs
   signal A : std_logic_vector(15 downto 0) := (others => '0');
   signal B : std_logic_vector(15 downto 0) := (others => '0');
   signal P : std_logic_vector(31 downto 0);

   signal A0 : std_logic_vector(15 downto 0) := (others => '0');
   signal B0 : std_logic_vector(15 downto 0) := (others => '0');
   signal P0 : std_logic_vector(31 downto 0);

   signal A1 : std_logic_vector(15 downto 0) := (others => '0');
   signal B1 : std_logic_vector(15 downto 0) := (others => '0');
   signal P1 : std_logic_vector(31 downto 0);

BEGIN

--  Instantiate the Unit Under Test (UUT)
   uut: BOOTH_MULT_16BIT PORT MAP (
          A => A,
          B => B,
          P => P
        );

   uut0: BOOTH_MULT_16BIT PORT MAP (
          A => A0,
          B => B0,
          P => P0
        );

   uut1: BOOTH_MULT_16BIT PORT MAP (
          A => A1,
          B => B1,
          P => P1
        );
--
-- uut: SA_MULT_16BIT PORT MAP (
--          A => A,
--          B => B,
--          P => P
--        );

-- uut: MULT18X18 PORT MAP (
--          A => A,
--          B => B,
--          P => P
--        );

--uut: WRAPPER PORT MAP (
--          A => A,
--          B => B,
--          P => P
```

```
--        );

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;


--       insert stimulus here
      A  <= "1111111111111111";
      B  <= "1111111111111111";

      A0 <= "0001101010001111";
      B0 <= "0001001110110110";

      A1 <= "0000000001111000";
      B1 <= "0000000000000010";


      wait;
   end process;

END;
```