Design and Implementation of Digital Signal Processing Hardware for a Software Radio Reciever

Jake Talbot
Utah State University

DESIGN AND IMPLEMENTATION OF DIGITAL SIGNAL PROCESSING HARDWARE FOR A SOFTWARE RADIO RECIEVER

by

Jake Talbot

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Jacob H. Gunther Major Professor	Dr. Todd K. Moon Committee Member

Dr. Aravind Dasu
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2008

Abstract

Design and Implementation of Digital Signal Processing Hardware for a Software Radio Reciever

by

Jake Talbot, Master of Science

Utah State University, 2008

Major Professor: Dr. Jacob H. Gunther
Department: Electrical and Computer Engineering

This project summarizes the design and implementation of field programmable gate array (FPGA) based digital signal processing (DSP) hardware meant to be used in a software radio system. The filters and processing were first designed in MATLAB and then implemented using very high speed integrated circuit hardware description language (VHDL). Since this hardware is meant for a software radio system, making the hardware flexible was the main design goal. Flexibility in the FPGA design was reached using VHDL generics and generate for loops. The hardware was verified using MATLAB generated signals as stimulus to the VHDL design and comparing the VHDL output with the corresponding MATLAB calculated signal. Using this verification method, the VHDL design was verified post place and route (PAR) on several different Virtex family FPGAs.

(123 pages)

To my beloved family: Courtney, Caden, and Lucas.

Acknowledgments

I am indebted to many people for the completion of this project. First and foremost, I would like to thank my major professor, Dr. Gunther. He has shown a tremendous amount of patience throughout the design process. He has always been eager to entertain questions and elicit advice whenever I would arrive at his office, often times unannounced. I would also like to thank my committee members, Dr. Moon and Dr. Dasu, for their patience and willingness to help me throughout the course of this project. Next, I would like to thank my wife for her loving support throughout my college career, especially for helping me edit the first draft of this thesis. She often supplied motivation when I felt like I had none. Thanks also to my office mates: Roger West, John Flake, Cameron Grant, and Darin Nelson who have helped me greatly these last few semesters. Finally, I would like to extend thanks to my parents, Steve and Jill. They have provided unending support and encouragement to me throughout my whole life.

Jake Talbot

Contents

List of Tables

List of Figures

Chapter 1

Background

1.1 Introduction

In recent years, there has been a great demand for wireless communications technology. As a result of this increasing demand, several new wireless communication standards have been created and put into use. With the advent of all of these different wireless standards, it is desirable to have a radio receiver that is capable of communicating with several different standards. This requires a radio to be able to reconfigure its features (demodulation, error correction, etc.) according to the type of communications standard it is trying to interact with. Another advantage of a radio that can be reconfigured is the fact that the radio does not become obsolete with the creation of a new wireless standard. A radio that can be reconfigured is called a software radio.

1.2 Software Radios

In order for a radio to be able to reconfigure itself based on the signals it is receiving, it has to be largely defined with software. In other words, these types of radios are able to reconfigure the hardware using software. This terminology is somewhat vague. How much flexibility does a radio need to have in order to be called a software radio? The following quote helps to define what it means to be a software radio.

> A good working definition of a software radio is *a radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software* [1, pg. 2].

As an example, a radio that utilizes a digital signal processor or microprocessor is not necessarily a software radio. On the other hand, if a radio defines its modulation, error correction and encryption in its software and also exhibits some flexibility over the radio frequency (RF) hardware, it is clearly a software radio. Generally, a software radio refers to a radio that is flexible with respect to the software, but the software operates on a static hardware platform.

In order to maximize flexibility, a software radio receiver digitizes the received signal as soon as possible to utilize the flexibility of digital signal processing (DSP). This is usually done after an analog front end filters, amplifies, and mixes the signal to an intermediate frequency (IF). The signal is then digitized and translated to baseband using a digital down converter (DDC).

1.3 Project Scope

In this project, hardware will be designed and implemented that will perform an IF to baseband conversion of a received signal. This hardware is intended for use in a software radio. The functions that this hardware will perform are outlined below:

- Demodulation: The incoming signal will be translated from an intermediate frequency to baseband.
- Decimation: The baseband signal will be filtered and downsampled to a more manageable sampling rate.

Even with RF hardware partially demodulating the carrier signal to an intermediate frequency, the sample rate of the system is still fairly high (about 100 - 500 MHz). This being the case, a DSP chip will not be able to process the data coming from the analog to digital converter (ADC) fast enough for a real time setting. This leaves either a field programmable gate array (FPGA) or an application specific integrated circuit (ASIC) as the target hardware for this project. Being in a University setting, an FPGA is the more economical choice. Using an FPGA is desirable also because they can be reprogrammed on the fly, coinciding with the reconfigurability required of a software radio.

Having determined that an FPGA will be used to perform the processing, very high speed integrated circuit hardware description language (VHDL) will be used to implement the hardware. VHDL was chosen since the use of generic modules and generate for loops (a VHDL construct, discussed in detail in Chapter 3) will make the FPGA design flexible and reconfigurable. With generic modules, the processing the FPGA performs can be greatly altered or modified by simply changing the generic values of the top level design. Generate for loops that are based on the generic values will then be used to create the structures of the decimating filters.

For this design, the intermediate frequency is assumed to be $\frac{1}{4}$ of the ADC sample rate. This imposes some restrictions on the analog front end of the receiver. These restrictions on the RF hardware make the demodulation circuitry multiplier free (see Chapter 2 for more details). Another implication of these simplifying assumptions is that the received signal is going to be oversampled by a large factor, leading to a high decimation factor required of the DDC hardware. For example, if $F_S = 100\,MHz$ and the bandwidth of the desired signal is $100\,kHz$, the decimation factor will be in the neighborhood of 1000.

Since the DDC hardware is going to need to decimate the input signal by a large factor, the decimation will be broken up into a cascade of two to three decimating filters. The first filter in the cascade is going to have to operate at a fairly high sample rate. Because of this, the first filter in the cascade was chosen to be a cascaded integrator-comb (CIC) filter [1–4]. These filters have a low-pass response and are multiplier-free. With applications of the Noble identities, these filters can also be made to decimate or interpolate. After the CIC filter partially decimates the input signal, a more complicated filter can be implemented to perform the rest of the decimation. Due to this, the next filter(s) in the cascade were chosen to be polyphase decimating filters [2, 5]. The theory of each of these types of filters are described in the following sections.

1.4 CIC Filter Theory

As mentioned before, a cascaded integrator comb (CIC) filter is a multiplier free filter that has a low-pass response[1]. These filters can also be used as a decimating filter. As the

[1]This development follows closely from [1].

name suggests, a CIC filter is constructed by cascading two simple filter structures together: combs and integrators. Figure 1.1 shows the structure of each of these building blocks.

The more integrator and comb filters that are cascaded together, the better the CIC filter does at filtering. The CIC filter exhibits better stopband attenuation, but the sinc shaping in the passband is more pronounced. In essence, a cascade of an integrator and a comb filter is equivalent to an FIR filter with an impulse response of a rectangular window of length M. This translates to a sinc shaped filter response. Figure 1.2 shows the frequency response of CIC filters with one, three, and five stages to further illustrate this point.

In order to make this a decimating filter, N integrator filters are cascaded together followed by N comb filters and finally by a downsampler. Figure 1.3 shows a block diagram of this cascade.

The transfer function of the decimating CIC filter shown in fig. 1.3 is

$$H(z) = \left(\frac{1 - z^{-RM}}{1 - z^{-1}} \right)^N ,$$

leading to the frequency response

$$H(\omega) = \left(\frac{\sin\left(\frac{\omega RM}{2}\right)}{\sin\left(\frac{\omega}{2}\right)} \right)^N ,$$

where R is the decimation factor of the filter, M is the differential delay (or the number of samples to delay the input signal in the comb stages), and N is the number of stages in the CIC. Using one of the Noble identities, the downsampler can be pushed before the comb filter cascade. This is shown in fig. 1.4.

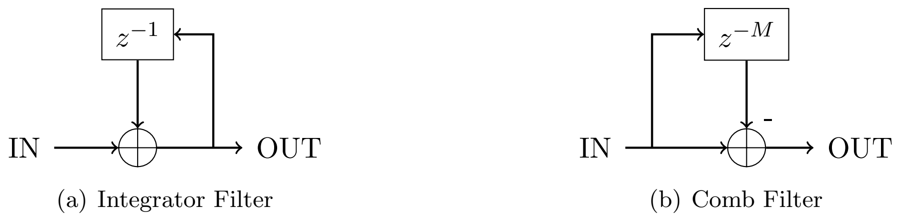
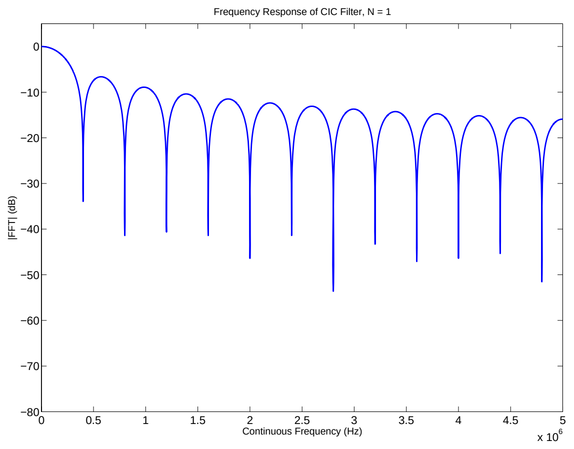
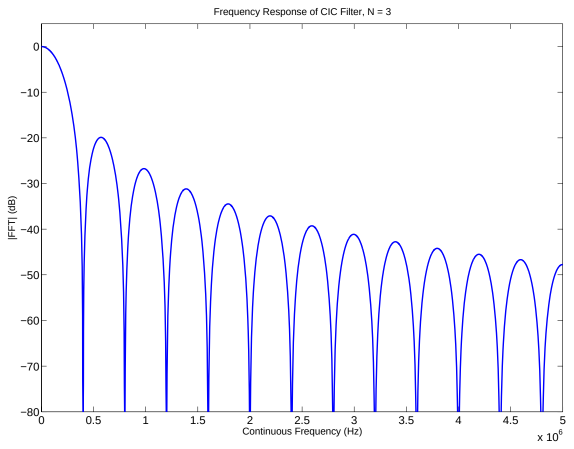


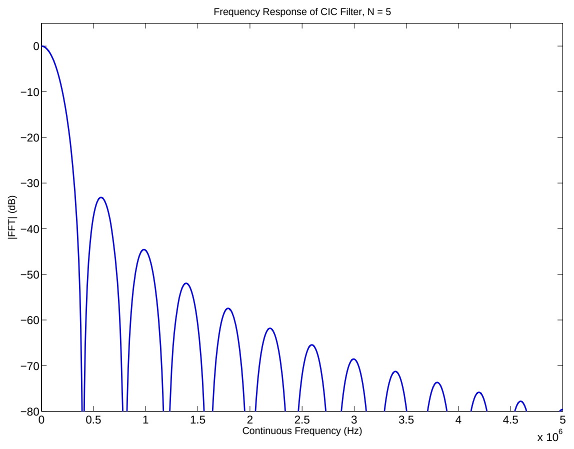
Fig. 1.1: Basic building blocks of a CIC filter.



(a) One stage CIC filter



(b) Three stage CIC filter



(c) Five stage CIC filter

Fig. 1.2: Frequency response of three CIC filters, each with a different number of stages.

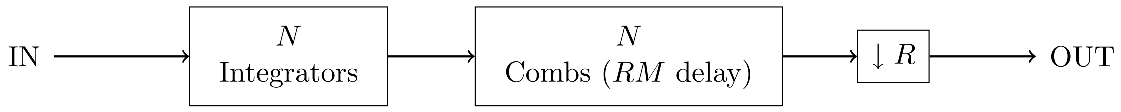


Fig. 1.3: Block diagram of a CIC decimator.

The structure shown in fig. 1.4 is desirable because in this configuration, the comb cascade can operate at the lower sample rate. There are several advantages for using a CIC filter. The first reason is that they are multiplier free, making them ideal for high sample rate applications. Secondly, they can be organized such that they decimate the incoming signal while at the same time filtering with a low-pass filter to avoid aliasing in the frequency domain. One disadvantage of the CIC filter, however, is that they have relatively large gain. The gain of a CIC filter is $(RM)^N$. This leads to large accumulator registers in the integrator stages when fixed-point arithmetic is used. If the bit width of the accumulator registers is not sufficient to allow for the gain of the filter, they can overflow and cause the filter to be unstable. It has been shown [6] that if the output bit width follows

$$B_{\text{out}} = B_{\text{in}} + N\lceil \log_2 RM \rceil, \tag{1.1}$$

where B_{out} is the bit width of the CIC filter output and B_{in} is the bit width of the input to the CIC filter, then the accumulators will not overflow and the filter will be stable. It can be seen from (1.1) that the bit width of the accumulators can be quite large. With modern FPGAs however, registers are plentiful. Due to this, full precision according to (1.1) will be kept throughout the CIC filter and quantized before the next filter processes the data.

Another disadvantage of the CIC filter is that it has a sinc shaped frequency response. This could lead to unwanted attenuation in the passband of the filter. To correct this, a

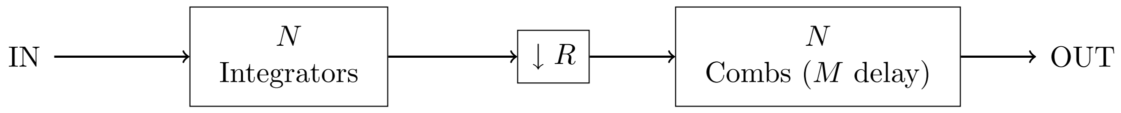


Fig. 1.4: Block diagram of a decimating CIC filter in which the downsampler has been pushed before the comb cascade using a Noble identity. Notice that the comb filters now delay the input signal by M samples instead of RM samples as in fig. 1.3.

CIC correction filter [7] usually follows a decimating CIC filter to correct the unwanted droop. The filter that will follow the CIC filter and decimate the signal to its final desired sample rate is a polyphase FIR filter.

1.5 Polyphase FIR Filter Theory

A decimating FIR filter is constructed by taking a prototype filter (a low-pass filter with a cutoff frequency near $\frac{\pi}{M}$ radians or $\frac{1}{2M}$ cycles, where M is the desired decimation factor) and decomposing the coefficients into several shorter polyphase filters in such a way that a Noble identity can be invoked to push the downsampling operation before the polyphase representations of the filter[2]. This is desirable for this application for two main reasons:

1. The polyphase filters operate at the slower sample rate.
2. A single FIR filter can be used to both downsample and filter the input signal to avoid aliasing in the downsampling process.

The first step in constructing a polyphase representation is to decompose the coefficients, $h(n)$, into M different polyphase filters, each with $\frac{N}{M}$ taps where N is the number of taps in the prototype filter and M is the decimation factor. This decomposition can be visualized by writing the Z-transform of $h(n)$ in the following way:

$$H(Z) = \begin{array}{ccccccc} h(0) & + & h(M+0)z^{-M} & + & h(2M+0)z^{-2M} & + & \dots \\ +h(1)z^{-1} & + & h(M+1)z^{-(M+1)} & + & h(2M+1)z^{-(2M+1)} & + & \dots \\ +h(2)z^{-2} & + & h(M+2)z^{-(M+2)} & + & h(2M+2)z^{-(2M+2)} & + & \dots \\ \dots & & & & & & \\ +h(M-1)z^{-(M-1)} & + & h(2M-1)z^{-(2M-1)} & + & \dots & & \end{array}$$

[2]This discussion follows closely from [2] and [3].

$$= \begin{array}{c} H_0(z^M) \\ +z^{-1}H_1(z^M) \\ +z^{-2}H_2(z^M) \\ \cdots \\ +z^{M-1}H_{M-1}(z^M). \end{array} \tag{1.2}$$

Now, each row in (1.2) can be looked at as a polynomial in z^M, with each row offset with a one sample delay from the row above it. Each row now represents a polyphase filter. Figure 1.5 shows a block diagram that performs the same processing as (1.2) and then downsamples the signal by M.

We are now in a position to apply a Noble identity and push the downsampler in front of the polyphase filters. The block diagram showing this operation is shown in fig. 1.6. The sequence of delay elements and downsamplers preceding the polyphase filters in fig. 1.6 is often replaced with a commutator switch.

One or two cascaded stages of these polyphase filters will follow the decimating CIC to finish the decimation of the oversampled input signal coming from the ADC in the system. This concludes the theory of the filters used in this project. There will be two polyphase filters if the droop in the passband of the CIC causes enough distortion to require a CIC compensation filter. If the droop in the passband is acceptable, however, there will only be one decimating polyphase filter in the system.

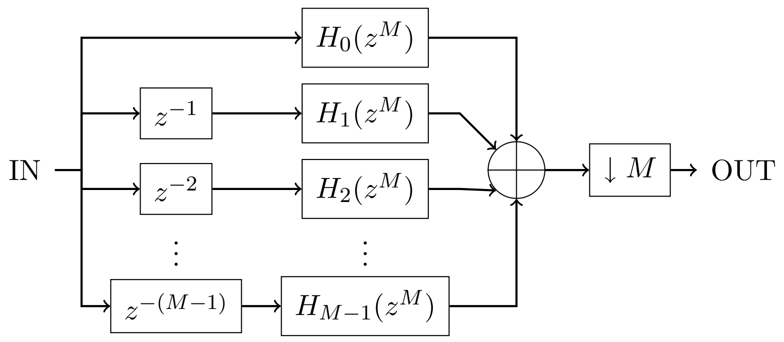


Fig. 1.5: Block diagram showing the processing from (1.2) using polyphase filters and delay elements. The filtered signal is then downsampled.

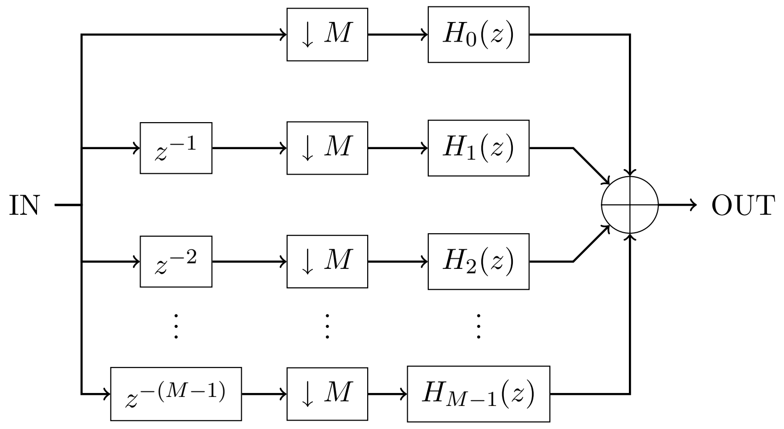


Fig. 1.6: Block diagram showing a decimating polyphase filter after a noble identity has been applied to the block diagram in fig. 1.5, pushing the downsamplers in front of the polyphase filters. This is the processing that will be performed after the CIC filter in the design of this project.

1.6 Conclusion

In this chapter, we have described a need for this project and given sufficient background theory to understand the terminology used in the following chapters of this report. This chapter has also defined the DSP that is involved with this project and outlined the processing that needs to be taken place for the DDC of the incoming software radio signal. The following chapters will describe the design (in MATLAB) of the filters and the VHDL implementation and verification. The final chapter will summarize the work completed on this project and outlines possible future work that is related to the work done in this project. Appendix B and Appendix C show the MATLAB and VHDL listings of this design, respectively. These appendices are included for reference.

Chapter 2
MATLAB Design

2.1 Introduction

As a first step in the design process, a preliminary MATLAB design was made. The intended FPGA system was simulated using MATLAB before the digital system was implemented using VHDL. This step in the design process served several purposes. First, a more thorough understanding of how the hardware should work was attained when this step was carried out. Second, intermediate signals from the MATLAB design of the system can be input into a VHDL testbench and used to stimulate certain parts of the design. Third, the MATLAB design was used to determine the tolerable precision of the polyphase FIR filter coefficients. Fourth, the performance of the system can be visualized and verified much easier in a MATLAB environment than a VHDL environment. Because of these reasons, a preliminary MATLAB design was implemented.

2.2 Specifications

To verify correct functionality of the filters in the digital down conversion processing chain, some general requirements of the spectrum of the incoming signal had to be defined. These specifications are shown in Table 2.1. Note that these specifications are only to test a single configuration of the software radio. It is necessary to specify a set of specifications so that the hardware can be verified for this class of signals. This being said, a main goal of the software radio will be to successfully process other, different types of signals when it is implemented in a communications system.

To further visualize each specifications role and importance, fig. 2.1 shows an example spectrum.

Table 2.1: Input signal specifications.

Specification	Description	Value
F_S	The sampling frequency of the ADC used in the system.	100 MHz
F_0	The carrier frequency of the desired signal. For this design, this specification is fixed at $\frac{1}{4}F_S$. This makes the demodulation multiplier-less (see discussion below).	$\frac{1}{4}F_S = 25$ MHz
W	This specifies the one-sided bandwidth of the desired spectrum.	50 kHz
ΔF	Frequency separation between wanted signal and unwanted signals in the spectrum.	125 kHz
δF	Transition band required of the anti-aliasing filter for the initial decimation.	25 kHz

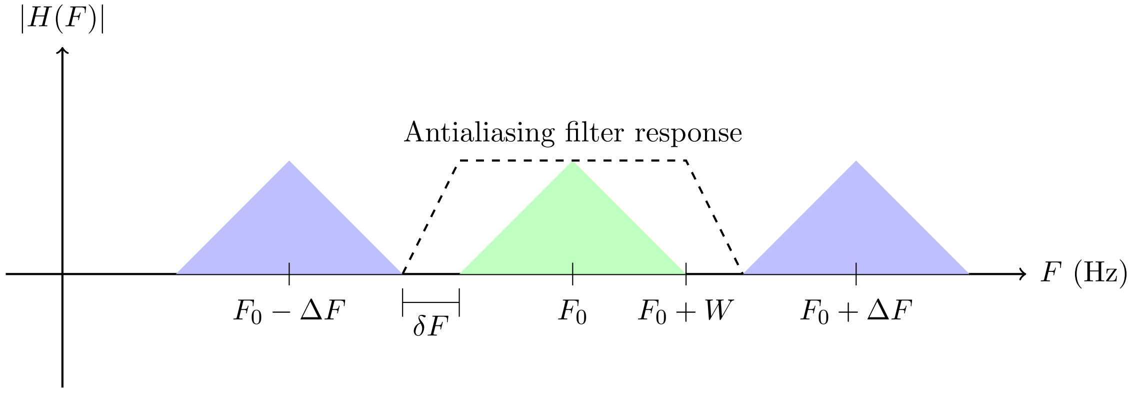


Fig. 2.1: A figure showing the role of each of the specifications outlined in Table 2.1.

Choosing F_0 to be $\frac{1}{4}F_S$ makes the demodulator multiplier-less in the following way. Normally, the demodulator modulates by $\sin(\cdot)$ and $\cos(\cdot)$ at the carrier frequency F_0, but since $F_0 = \frac{1}{4}F_S$, the modulators can be simplified:

$$\begin{aligned} \cos(2\pi F_0 n) &= \cos\left(\frac{2\pi}{4}F_S n\right) \\ &= \cos\left(\frac{\pi}{2}F_S n\right) \\ &= \begin{cases} 1 & n = 0, 4, 8, \ldots \\ 0 & n = 1, 3, 5, \ldots \\ -1 & n = 2, 6, 10, \ldots \end{cases} . \end{aligned} \tag{2.1}$$

The demodulator in the in-phase branch of the receiver can be similarly simplified:

$$\sin\left(\frac{\pi}{2}F_S n\right) = \begin{cases} 1 & n = 1, 5, 9, \ldots \\ 0 & n = 0, 2, 4, \ldots \\ -1 & n = 3, 7, 11, \ldots \end{cases} . \tag{2.2}$$

It is important to note, though, that this F_0 is not necessarily the true carrier frequency of the desired signal. It is more likely that F_0 is going to be an intermediate frequency where an analog front end to the digital receiver has performed a partial demodulation. So to be more precise, F_0 is the intermediate carrier frequency.

Another important specification that needs to be fixed for this example scenario is the overall decimation factor. Suppose that a square root raised cosine (SRRC) pulse shape is used with an excess bandwidth (α) of 100%. Also assume that we want the final downsampled signal to be oversampled by a factor of two, that is:

$$F_{S,final} = 2R_S,$$

where R_S is the symbol rate and $F_{S,final}$ is the final sampling rate after the decimation stages. Because of the SRRC pulse shape, the bandwidth of the signal (W) can be related to the sampling rate:

$$W = \frac{1+\alpha}{2T_S} = \frac{1}{T_S} = R_S,$$

so that

$$F_{S,final} = 2R_S = 2W = 100kHz,$$

and finally

$$D_{TOT} = \frac{F_S}{F_{S,final}} = \frac{100MHz}{100kHz} = 1000,$$

where D_{TOT} is the overall decimation factor.

Notice that D_{TOT} can be factored as $2^3 \cdot 5^3 = 8 \cdot 125$. This is convenient because it means that each decimation stage is able to decimate by an integer factor. The CIC filter will decimate by a factor of 125, leaving the polyphase FIR filter to decimate by the remaining factor of 8.

To test the design of the filters and to verify their proper operation, a test signal consisting of a sum of cosines at four different frequencies will be processed by the filter cascade. The four different frequencies were chosen to be: $F_0 + W$, $F_0 - W$, $F_0 + \delta F$, and finally $F_0 - \delta F$. Obviously, after the filtering and downsampling stages, the first two frequency components should be intact while the second two frequency components should be filtered out. Figure 2.2 shows the test signal that is applied to the system.

Figure 2.3 shows the signal after the test signal was demodulated with the sequence of $\{\pm 1, 0\}$ shown in (2.1).

After demodulation, the signal will be downsampled and decimated by a factor of 125 by the CIC filter. Immediately following this decimation, a polyphase filter will filter and downsample the signal by the remaining factor of 8. Figure 2.4 shows a block diagram of the top level system.

2.3 CIC Filter Design

Having defined the specifications of this test spectrum, the filters can be designed. As discussed in Chapter 1, the demodulated signal will first be processed by a CIC filter.

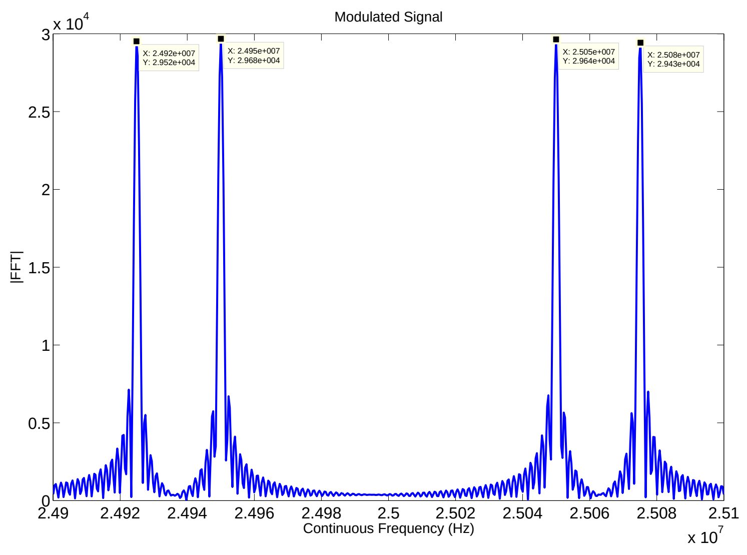


Fig. 2.2: Spectrum of the test signal that contains the four frequency components at F_0+W, F_0-W, $F_0+\delta F$, and $F_0-\delta F$. The first two are desired, whereas the second two frequency components are undesired and should be filtered out.

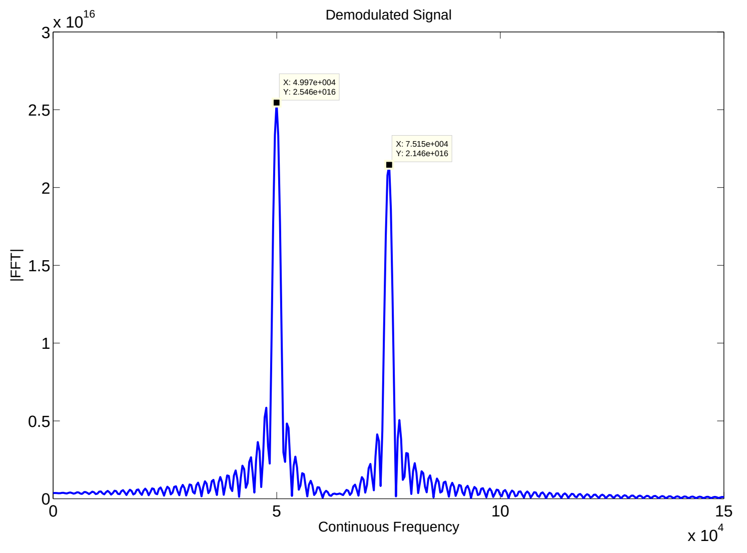


Fig. 2.3: Spectrum of the test signal after demodulation by cosine.

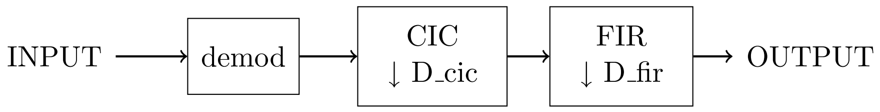


Fig. 2.4: Top-level block diagram of the system.

At this stage in the downsampling process, the incoming signal is heavily oversampled ($F_S/2 = 50MHz$ whereas $W = 50kHz$). This means that the main lobe of the CIC filter has to be fairly narrow, requiring a high number of stages. Through experimentation, it was found that a CIC filter with five stages had a sufficiently narrow main lobe. Figure 2.5 shows the frequency response of a CIC filter with five stages and a differential delay of two.

Notice that the CIC filter will not completely filter out the unwanted frequency component, and also that it does attenuate the desired signal slightly. This attenuation, however, is negligible (in this case) and does not affect the functionality of the receiver. This filter does, however, avoid aliasing before the downsampling operation because it filters out everything except for the two signals that are close to baseband. The demodulated signal after being CIC filtered and decimated by 125 is shown in fig. 2.6.

Since the effects of the sinc-shaping on the signal from the CIC signal are negligible, there is no need to make the polyphase FIR filter a CIC correction filter [7] and it can simply be designed as a low-pass filter.

2.4 Polyphase FIR Filter Design

After the decimation accomplished by the CIC filter, the two frequency components that remain are separated enough in the spectrum that the polyphase FIR filter can properly filter the remaining unwanted frequency components out of the spectrum while, at the same time, downsampling to the desired sampling rate, $F_{S,final}$.

Since the CIC filter decimated the signal by a factor of 125, W is now effectively at $(50 \cdot 125)kHz = 6.25MHz$ in the spectrum while the unwanted frequency component lies at $(75 \cdot 125)kHz = 9.375MHz$ at a sample rate of $\frac{F_S}{125} = 800kHz$. This being the case, the prototype filter was designed to be a low-pass filter with a passband (F_{pass}) of $6.25MHz$ and a stop-band (F_{stop}) of $9.375MHz$. This filter was designed using a Chebyshev window

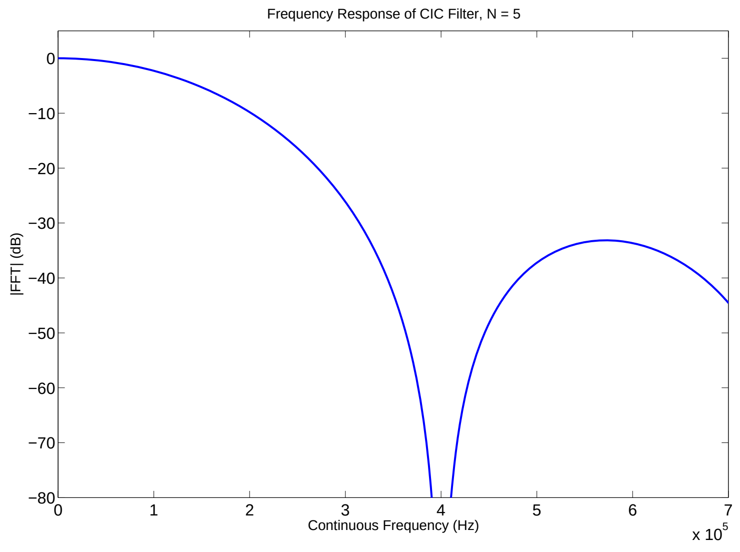


Fig. 2.5: Frequency response of the CIC filter with five stages. At W, the frequency of interest, the attenuation is about -0.56 dB. At $W + \delta F$, the first unwanted frequency component, the attenuation is about -1.28 dB.

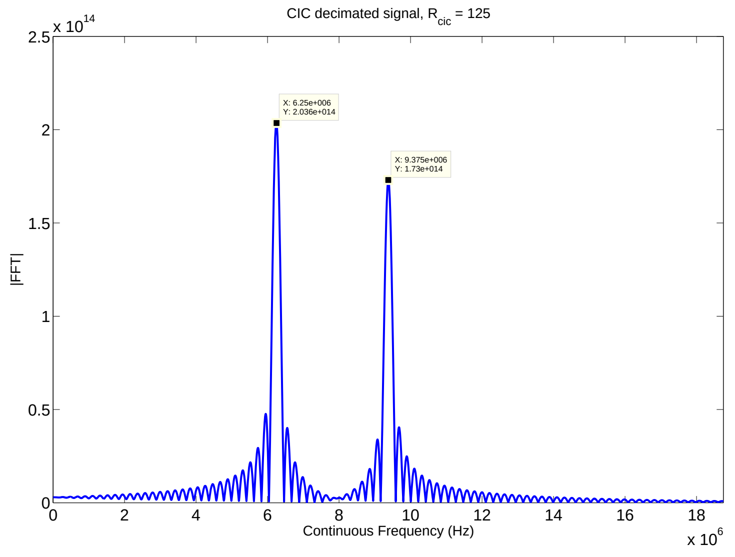


Fig. 2.6: Test spectrum after CIC filter has decimated it by 125.

with 60 dB of attenuation in the stop-band. To achieve the specified stop-band attenuation, 240 filter taps were used. Figure 2.7 shows the frequency response of the designed filter.

In the VHDL implementation, the filter coefficients are going to be represented as two's-complement signed integers. This means that the designed filter coefficients need to be quantized. Sufficient precision needs to be maintained to keep the necessary stop-band attenuation. After experimentation, it was determined that 14 bits of precision in the coefficients were sufficient. Figure 2.8 shows the filter response of the filter with quantized coefficients along with the full precision filter.

Using the designed prototype filter taps, a polyphase filter is created using the process outlined in Chapter 1. In this case, there are 240 taps and a downsampling factor of eight, so the polyphase filterbank will have eight filters, each with 30 ($\frac{240}{8}$) taps (see fig. 1.6).

After the polyphase filter is designed, the CIC decimated signal is further filtered and downsampled. Figure 2.9 shows the final, fully decimated signal. As you can see, the filter cascade filters out all the unwanted signal components for this example spectrum and avoids aliasing in the downsampling operations simultaneously.

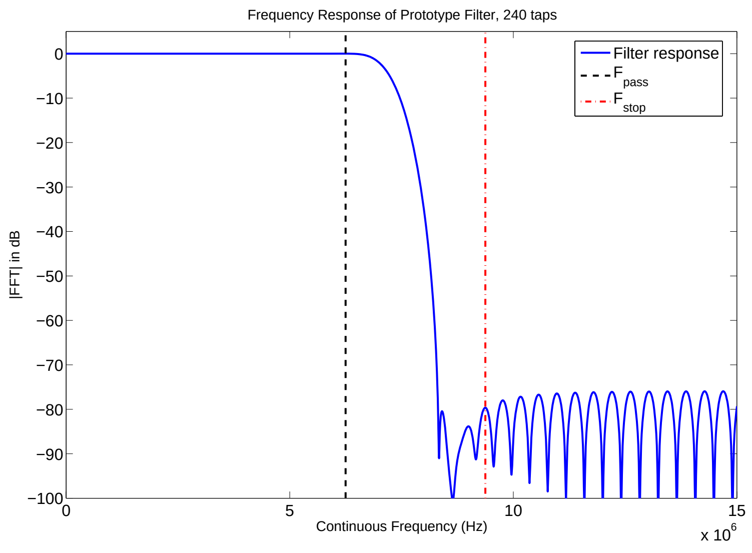


Fig. 2.7: Frequency response of the designed low-pass prototype filter. Notice that the stop-band attenuation is actually 80 dB. This is because the filter was designed above specs to account for quantization error when fixed point coefficients are used.

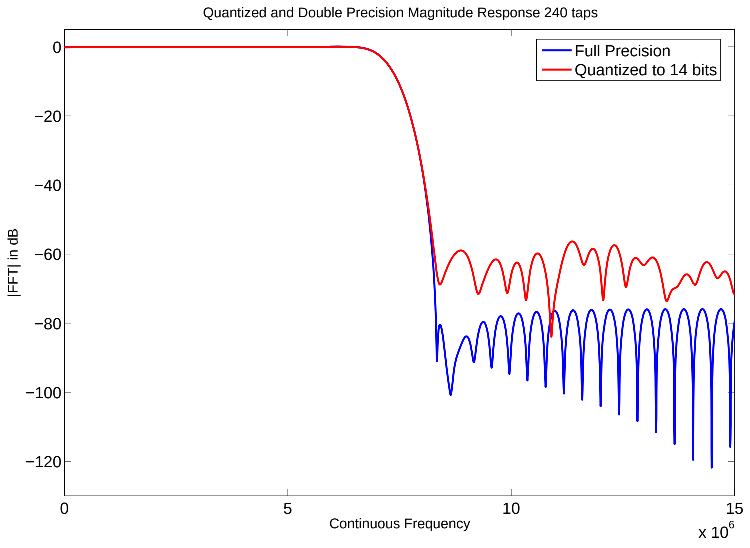


Fig. 2.8: Frequency response of the quantized low-pass prototype filter. The frequency response of the full precision filter is also shown for comparison.

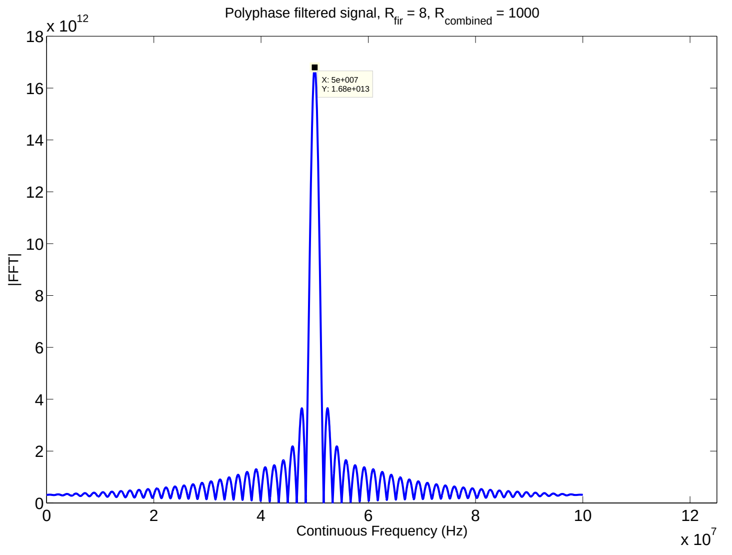


Fig. 2.9: Spectrum of the fully decimated signal. This signal is sampled at the desired sample rate, $F_{S,final} = 100kHz$, which is twice the symbol rate of the communications system.

2.5 Conclusion

In this chapter, a test signal was created and the appropriate filters were designed to properly process this signal. As you can see from the preceeding sections, the filters designed properly filter and downsample the signal such that aliasing is avoided. Also, the signal is decimated to the appropriate sample rate. The sample rate is low enough now that something like a DSP chip can be used to incorporate further flexibility in the signal processing that remains to make decisions on what symbols were sent.

Chapter 3
VHDL Design

3.1 Introduction

For reasons discussed in Chapter 1, the down conversion of the signal is performed in an FPGA. This allows for the flexibility required of a software radio. In order to make the design flexible, VHDL generics were incorporated into the design. In addition to generics, generate for loops (which were based on the generic parameters) were used to generate the structure of the filters. In order to accommodate the different sampling rates that are in the design, one global clock signal is used to drive the flip flops. Clock enable signals are then created and used to drive clock enabled flip flops for the slower sampling rate portions.

To implement the math functions required in the antialiasing filters, two's-complement signed, fixed point integer arithmetic is implemented on the FPGA. This facilitates the use of the VHDL operators (`+,-,*`), thus enabling the use of generic adders and multipliers. Also, initializing the FIR filter tap ROMs is done by loading them with files generated by MATLAB. The generic values of the VHDL design are described next.

3.2 Generics

As mentioned in the previous section, one important aspect of the design are the generic parameters. These are used to make the FPGA design flexible. In order to change the behavior of the processing, one simply has to change the related generics. Table 3.1 outlines the generics that are used to describe the top level of the design hierarchy. These generics then get mapped to the appropriate modules in the lower levels of the design hierarchy.

Table 3.1: Description of generics.

generic	description
B_casc_in	The input bit width of the processing chain, essentially the output bit width of the ADC used in the system.
B_cic_out	The output bit width of the CIC filter *and* the bit width of the accumulators in the integrator stages.
D_cic	The decimation factor of the CIC filter.
N_stages_cic	The number of stages in the CIC filter.
N_poly_taps	The number of taps in the polyphase FIR filter.
B_poly_coeffs	The number of bits in the coefficients of the FIR filter.
B_poly_in	The input bit width of the polyphase FIR filter. This is effectively the number of bits to quantize the CIC filter output to before the FIR filter processes the data.
B_poly_out	The full precision output of the polyphase FIR filter. This is also the number of bits used in the accumulator registers in the acc_D modules.
B_casc_out	The output bit width of the filter cascade. This is a quantized version of the FIR filter output.
D_fir	The decimation factor of the FIR filter. Also the number of elements in the tap ROM modules.
B_rom_addr	The number of bits in the tap ROM module address. This is $\log_2$(D_fir).

As you can see, the behavior of the VHDL design can be greatly modified by simply modifying the generic values. It is important to note, however, that changing a generic changes the entire design. This means that the design needs to be synthesized, mapped, and routed again. This can be a potential problem for a software radio. One possible work-around is to have several different FPGA configurations (corresponding to different generic values) in on-board memory. The software can then decide which programming image to use to reprogram the FPGA with. A description of the VHDL modules used in the design hierarchy follows this section.

3.3 Design Hierarchy

The VHDL design has several different levels of hierarchy. This section explains each level of the hierarchy in detail. Figure 3.1 shows the top level of the hierarchy.

The following sections descend into the design hierarchy supplying descriptions of each module.

3.3.1 Demodulator

As shown in fig. 3.1, the demodulator module is the first block in the processing chain. For the reasons discussed in Chapter 2, there are no multipliers in the demodulator circuitry. This module assigns the output according to a two bit counter and (2.1).

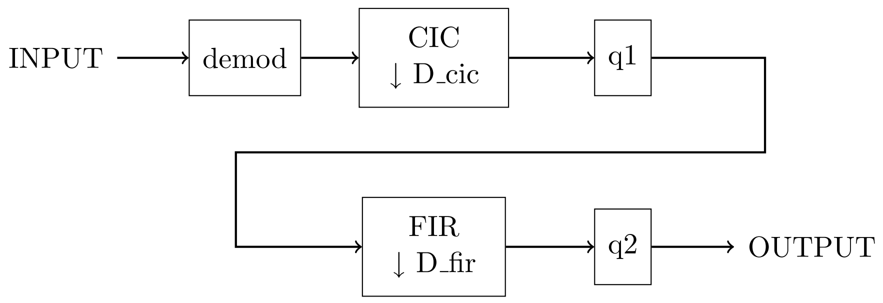


Fig. 3.1: Block diagram of the top level in the design hierarchy. The decimation factor generics for each filter are specified. The q1 and q2 blocks shown are quantizers. They perform quantization using truncation, keeping the upper bits of the input.

3.3.2 CIC Hierarchy

The CIC filter is broken down into four different levels of hierarchy. The top level of the CIC hierarchy is shown in fig. 3.2.

A few important things to note about the CIC filter design are:

- The input is sign extended to B_cic_out bits, and this amount of precision is kept throughout.
- B_cic_out is assumed to be sufficient for the accumulations that are occurring in the integrator stages. In other words, the adders in the CIC filter implementation have no carry bits.
- The CIC filter is pipelined: the accumulator registers are arranged in such a way that there is only one addition in between the register layers [8]. Also, pipeline registers were added in the comb stages in order to only require one subtraction operation in between registers.

The next sections describe the Integrators, Downsample, and Combs modules shown in fig. 3.2.

CIC Integrators

The integrators module of the CIC filter is simply a cascade of several accumulators. How many accumulators to cascade is governed by the N_cic_stages generic. A generate for loop based on this generic is used to cascade the appropriate number of accumulators in this design. Figure 3.3 shows a block diagram of the structure of this module.

It is important to note also that the integrators have to operate at the highest sample rate, therefore, it is critical that they are sufficiently optimized. The next section shows the design of the downsampler block shown in fig. 3.2.

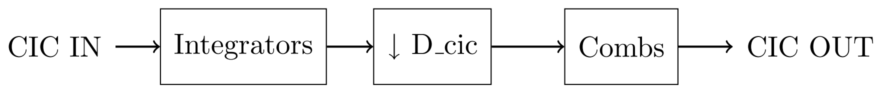


Fig. 3.2: The top-level of the CIC filter hierarchy.

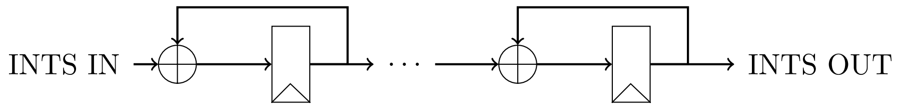


Fig. 3.3: Integrators module structure. This structure was generated using a generate for loop based on the N_stages_cic generic of the top-level module. Registers are shown as a block with a triangle on the bottom.

CIC Downsampler

As mentioned in sec. 3.1, the downsampling operation in the CIC filter is accomplished using a clock enabled register. Two controllers are also in the downsampler module. One control accounts for the latency due to the accumulator registers and outputs an enable signal to the second controller to signal it when to start counting. This ensures that the clock enabled flip flop passes on the first good sample to the rest of the design because it has properly waited for the first sample to propagate through the accumulators. The second controller generates the clock enable signal that drives the comb flip flops and some of the flip flops in the polyphase FIR filter. The first controller is based on the N_stages_cic generic whereas the second controller counts clock cycles according to the D_cic generic. Figure 3.4 shows a block diagram of the downsampler module.

After the downsampling operation, the data rate is divided by the D_cic generic. This means that the comb sections (and the first half of the FIR filter) can operate at the slower rate.

CIC Combs

The final stage of the CIC filter is a cascade of N comb filters. These filters operate

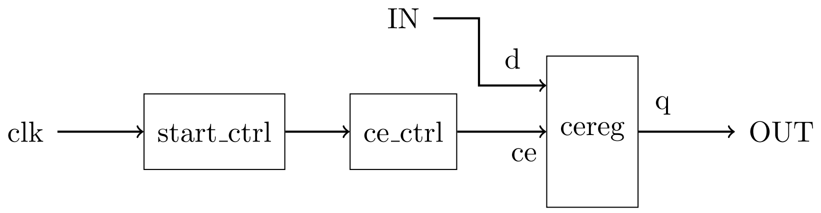


Fig. 3.4: Block diagram of the downsampler module in the CIC filter hierarchy. The start_ctrl module enables the ce_ctrl module to grab the first good sample.

at $\frac{F_S}{\text{D\_cic}}$. As in the CIC integrator stages, a generate for loop based on the N_stages_cic generic was used to cascade the desired number of comb filters together. A block diagram of the structure of the comb filter cascade is shown in fig. 3.5.

As you can see from fig. 3.5, pipeline registers were added to reduce the combinational path delay. With the extra register layers, there is only one subtractor in between a register. After the comb stage, the signal has been filtered and downsampled by a factor of D_cic. After the CIC filter, the signal needs to be downsampled to the final sample rate. This is accomplished with a polyphase FIR filter.

3.3.3 FIR Filter Hierarchy

The polyphase FIR filter is the final step in the digital down conversion of the signal. As mentioned before, after the CIC filter partially decimates the input signal, it gets quantized and then input to the FIR filter. This is so that the FIR filter can maintain full precision throughout the filtering operations. Also, to make the FIR filter flexible, tap ROM modules are loaded from text files. This design method makes it easy to design filter coefficients in MATLAB or another software tool and load them into the VHDL design. A block diagram of the top level of the FIR filter hierarchy is shown in fig. 3.6.

As you can see from fig. 3.6, the top level of the polyphase FIR filter hierarchy consists simply of the controllers that are needed to feed the signals needed to do the filtering into the FIR taps block. The FIR taps block is where the filtering actually takes place. Descriptions of the controllers shown are given here:

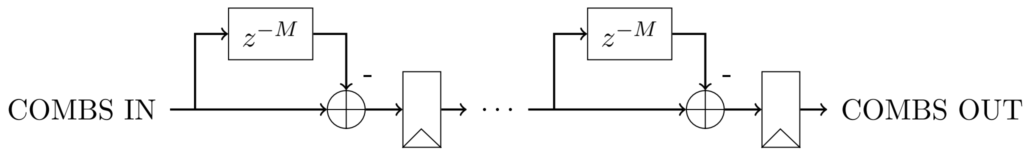


Fig. 3.5: A block diagram of the comb filter cascade. The M parameter is called the differential delay. There is no generic specifying this parameter; it is fixed at two for this design.

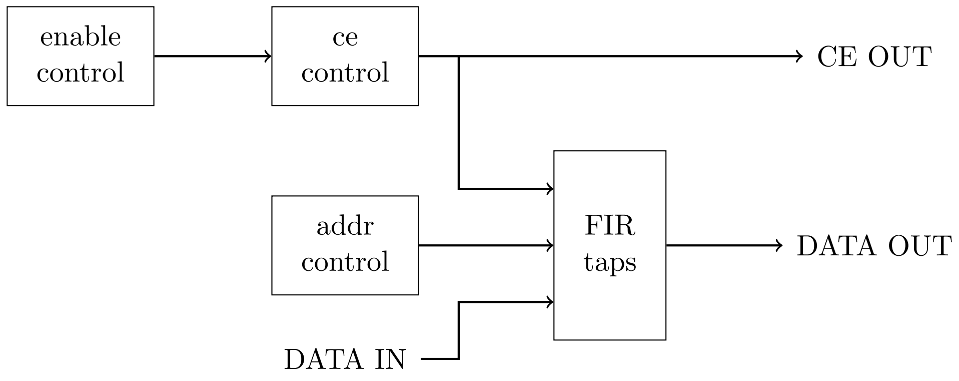


Fig. 3.6: Block diagram showing the top level of the hierarchy for the polyphase FIR filter.

enable control This controller serves much the same purpose as that of the start_ctrl controller shown in fig. 3.4. Namely, it outputs an enable signal to the ce control module signaling it when the first good sample has arrived. The major difference between start_ctrl and enable control is that enable control counts the clock enable signal output from the downsampling module in the CIC filter instead of the global clock signal. This controller accounts for the latency introduced by the pipeline registers in the comb section of the CIC filter.

ce control This controller uses both D_cic and D_fir to determine how many cycles of the global clock to count before its clock enable signal is output. Note that this controller counts only the global clock, it does not depend on the clock enable controller from the downsample module in the CIC filter.

addr control This controller is a simple down counter that is used to address the tap ROMs in the FIR taps module. It is important to note that this counter can count down from an arbitrary number, it does not have to be a power of two.

The following section describes the structure and functionality of the FIR taps module shown in fig. 3.6.

FIR Taps Module

The FIR taps module is the heart of the polyphase FIR filter. This is where the downsampling and filtering takes place. In a usual polyphase filter bank implementation, there would be several filters operating in parallel (the number of filters is equal to the decimation rate of the filter, in this case, this is described by the D_fir generic). In order to save resources on the FPGA, this structure was "collapsed" into one filter. This was accomplished by using tap ROM modules that hold D_fir coefficients and modules that accumulate the tap multiplier outputs for D_fir cycles [9]. This accumulation is where the effective downsampling operation takes place as well. Figure 3.7 shows the block diagram of this "collapsed" polyphase FIR implementation. It is also important to note that the structure was implemented using a generate for loop based on the N_poly_taps generic.

Some notes about the design of the FIR taps module are discussed here. The $ACCD_i$ modules shown in fig. 3.7 accumulate their input for D_fir cycles. This is where the downsampling operation takes place in the filter. Full precision is kept throughout the filter and quantized at the end. Full precision is kept in the following way: first, the output bit width of the multiplier is the sum of the bit widths of the two inputs (in terms of generics, this is B_rom_coeffs + B_poly_in). Second, enough guard bits in the accumulator inside the $ACCD_i$ modules are added to accommodate both for the accumulation and the adder chain that follows. This is done to avoid having to implement carry chain logic through the adder chain. The bit width of the accumulator registers in the $ACCD_i$ modules is set using the generic B_poly_out.

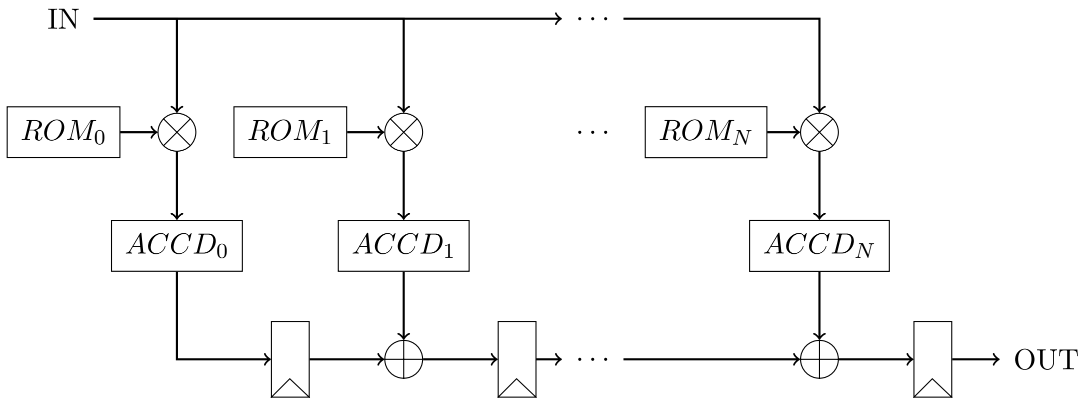


Fig. 3.7: Block diagram showing the structure of the FIR taps module in the polyphase FIR filter design hierarchy.

3.4 Conclusion

This chapter summarized the key points and design methodologies used in translating the DSP design discussed in Chapter 2 into an FPGA design using VHDL. This VHDL implementation obtains the flexibility required of a software radio using the generics shown in Table 3.1. The hierarchy of the VHDL design was then summarized and described. The next topic to be discussed is the method of verification used in the design process.

Chapter 4
Verification

4.1 Introduction

In this chapter, post place and route (PAR) simulations of the VHDL design are compared with outputs from the MATLAB design to verify the correct functionality of the synthesized, placed, and routed VHDL design.

Using Xilinx ISE 9.1i software tools[1], the VHDL design was synthesized to several FPGA targets. The post PAR simulations discussed in this chapter used a Virtex5-95sxt FPGA as the target. ModelSim[2] simulation software was used to perform the simulations.

4.2 Verification Method

In order to verify the VHDL design, intermediate signals from the MATLAB design were quantized and written to files. These files were then read into a VHDL testbench (using the VHDL textio package) and used to stimulate the design. Next, the VHDL module output was written to a file to compare with the corresponding MATLAB signal. For example, if the CIC filter implementation was to be tested, the demodulator output from MATLAB would be quantized and written to a file. Then the testbench would read in this file and use it to stimulate the CIC filter. The output of the CIC filter module would then be written to a file. Finally, the CIC filter output from the MATLAB simulation could be quantized and compared with the VHDL output.

Using this verification method, a simulation was conducted on the top level of the VHDL design hierarchy. Using the test signal described in Chapter 2 as stimulus, the final downsampled signal from the VHDL module matched exactly to the same signal from the

[1] © Xilinx Inc. 1995-2007
[2] © Mentor Graphics Corporation 2006

Table 4.1: Generic values used to generate fig. 4.2.

generic	value
B_casc_in	14
B_cic_out	54
D_cic	125
N_stages_cic	5
N_poly_taps	30
B_poly_coeffs	14
B_poly_in	26
B_poly_out	50
B_casc_out	24
D_fir	8
B_rom_addr	3

MATLAB simulation (as long as the MATLAB signal was quantized to the same fixed point precision). Figure 4.2 shows the post PAR ModelSim simulation results. Table 4.1 shows the generic values used for the simulation shown in fig. 4.2.

As you can see from fig. 4.2, the output signal changes at a much slower rate than the input signal. It is important to note also, that the ce_out signal shown in the figure is also an output of the system. This is so that the processing that follows the decimation can be synchronized to the output of the FPGA. Figure 4.2 also verifies that the design meets the desired sampling rate of the system. In other words, the integrator section of the CIC filter is able to operate at the desired frequency of $100MHz$. In order to verify the functionality of the system however, the output shown in fig. 4.2 needs to be compared with the equivalent MATLAB signal. This comparison is shown in fig. 4.1.

As you can see from fig. 4.1, the VHDL implementation and the MATLAB simulation perform the same desired processing of the test signal. A comparison of the VHDL output from fig. 4.2 with the VHDL output from fig. 4.1 can also be used as further verification.

4.3 FPGA Utilization

For reference, a Xilinx ISE 9.1i PAR report that summarizes the utilization of the target FPGA is included in fig. 4.3. Note that the processing that has been described throughout this report has been for a one-dimensional signal (e.g., BPSK or PAM).

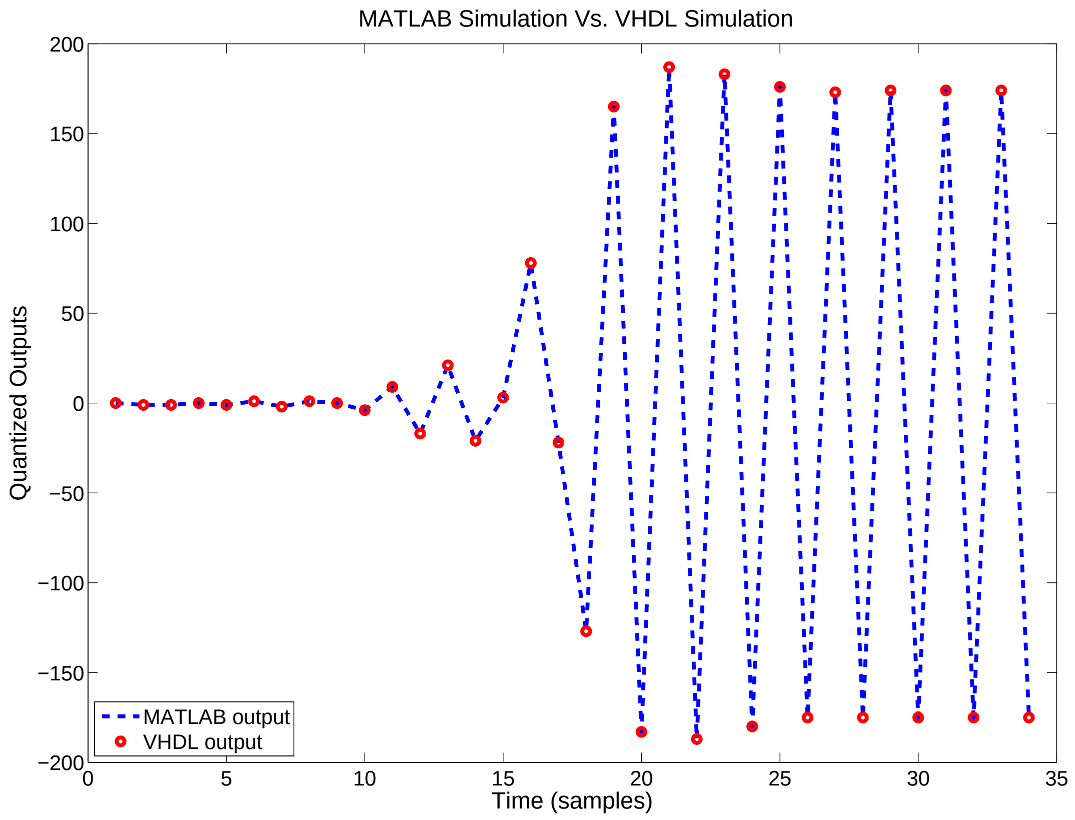


Fig. 4.1: Plot showing quantized outputs from the MATLAB simulation alongside the corresponding outputs from the post PAR top level VHDL simulation.

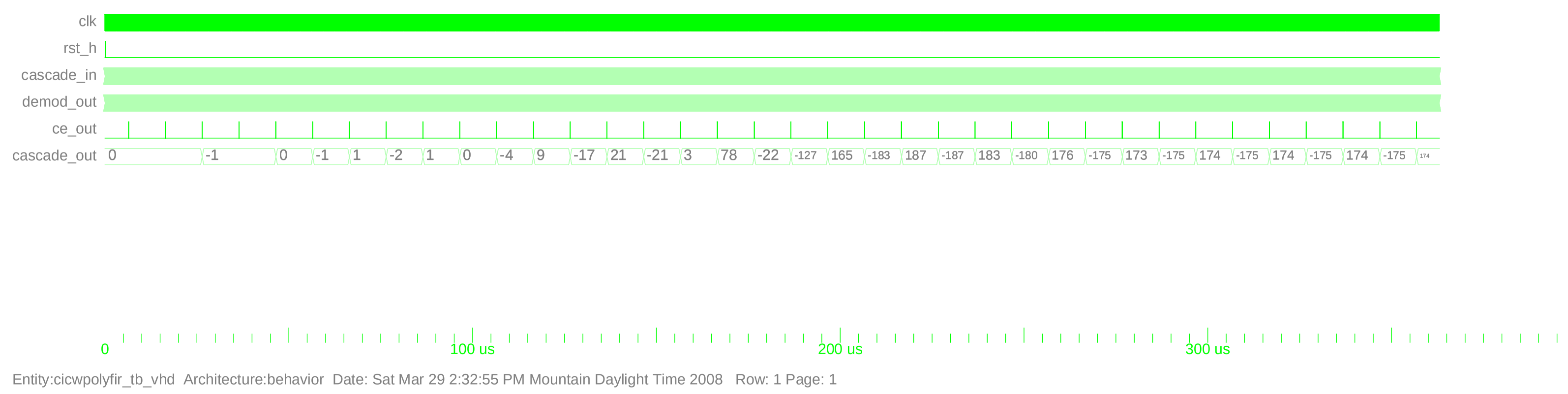


Fig. 4.2: Waveform showing the first 350μs of the top level post PAR simulation. The simulation was performed in ModelSim using input stimulus from the test signal generated in MATLAB. This test signal is described in Chapter 2.

```
Device Utilization Summary:

   Number of BUFGs                            2 out of 32      6%
   Number of DSP48Es                         60 out of 640     9%
   Number of External IOBs                   41 out of 640     6%
      Number of LOCed IOBs                    0 out of 41      0%

   Number of RAMB18X2s                       30 out of 244    12%
   Number of Slice Registers               6949 out of 58880  11%
      Number used as Flip Flops            6949
      Number used as Latches                  0
      Number used as LatchThrus               0

   Number of Slice LUTS                    6988 out of 58880  11%
   Number of Slice LUT-Flip Flop pairs    11039 out of 58880  18%
```

Fig. 4.3: An excerpt of a Xilinx ISE 9.1i PAR report outlining the target FPGA utilization for the VHDL design. The target FPGA here is a Xilinx Virtex5 95sxt.

If a two-dimensional signal such as QPSK or QAM needs to be processed, an in-phase processing branch needs to be included. Seeing as the processing would be identical for the in-phase branch, these utilization reports need to be roughly doubled if a signal that requires an in-phase branch is required.

One thing to note about fig. 4.3 is that this design uses 60 `DSP48` blocks when there are only 30 taps in the FIR filter. This is because in this design, the multiplication that takes place requires a precision that is above the specification of the `DSP48` block, so two of these blocks have to be used for each multiplication in the FIR filter.

As it can be seen from fig. 4.3, this design uses only a small fraction of the resources available for the target FPGA. Even if a QAM or QPSK signal needs to be processed, this design would only utilize about 20% of the resources available. This opens up the possibility of some of the downstream processing also being done on the FPGA.

4.4 Conclusion

In this chapter, the VHDL design and the MATLAB design were compared to verify the functionality of the VHDL implementation. The method of verification was described and then put to use to verify the functionality of the VHDL implementation. Furthermore, this chapter described the device utilization as reported by the Xilinx ISE 9.1i tool.

Chapter 5

Summary and Future Work

5.1 Introduction

FPGA-based hardware that will take a software radio signal as input and decimate it to the desired sampling rate has been designed, implemented, and verified. This design is planned to be the first processing element in a processing chain that will make up a software radio receiver. MATLAB was used to simulate the design at a high level and then VHDL was used to implement the design on an FPGA. Successful post PAR simulations have been obtained for several different FPGAs of the Xilinx Virtex family.

5.2 Work Completed

The scope of this project has been to design, implement, simulate, and verify the hardware design summarized above. A summary of the work completed is shown below.

- Development of DSP theory: Decimating filter architectures and applications were studied. As a result, a cascade of a CIC filter and a polyphase FIR filter was decided to be designed and implemented. Initially, the polyphase FIR filter was going to be made into a CIC correction filter, but later in the design process this was deemed unnecessary.

- DSP Design: After the general structure of the DSP processing was established, the filters had to be designed. Experiments were conducted using MATLAB to decide on filter parameters such as:

  - The number of stages in the CIC.

  - The polyphase filter specifications: Passband frequency, cutoff frequency, and stopband attenuation.
  - Polyphase filter design: The polyphase FIR filter was designed using an ideal frequency response and windowing using a Chebyshev window.

- MATLAB simulation: After the filters were designed, a MATLAB script was written that simulated the cascade of the two filters. A test signal that would simulate a realistic input signal was created and ran through the filter cascade. Several intermediate signals were quantized and written to files in order to verify the VHDL implementation.

- VHDL implementation: The filter cascade was implemented using VHDL. The use of generics and generate for loops allowed for the flexibility required of hardware used in a software radio.

- Verification: In order to deal with the fact that several sampling rates were present in this design, multi-cycle path specifications had to be input into the Xilinx tools to verify that the design would meet timing requirements (see Appendix A). The VHDL implementation was synthesized, placed and routed, and thoroughly simulated using the MATLAB signals mentioned above. Finally, the VHDL output was compared with the MATLAB simulation to ensure that the FPGA would perform the correct processing on the input signal.

In summary, some of the necessary hardware for a software radio receiver has been designed. Since the hardware is targeted for an FPGA and was written in VHDL, the design is flexible and portable. The VHDL code has also been commented to allow for future work on the design if needed. The MATLAB script used to verify the DSP design is also used to simplify the design process. In particular, a MATLAB script has been written that writes the coefficients of the FIR filter to several text files such that they can easily be loaded into the VHDL modules.

5.3 Future Work

The first major area of future work in regards to this project consists of verifying the design in an actual FPGA. Because of time and budget constraints, this has not been accomplished at the present time. There are also several other areas of future work that are related to this project.

The scope of the project discussed in this report consists of only one of several hardware elements that are needed for a software radio receiver. This being the case, several other projects can be done to construct the other necessary elements that make up an entire software radio receiver. Some of these elements are:

- An analog front end to the receiver: This circuitry takes the signal from the receiver antenna, demodulates it to the required intermediate frequency ($\frac{1}{4}F_S$), and then passes this to an ADC.

- Symbol recovery processing: After the signal is translated from the intermediate frequency and downsampled to the desired symbol rate (by the hardware designed for this project), several other processing steps need to be taken before the sent symbols can be recovered. This processing consists of matched filtering, symbol timing recovery, carrier phase recovery, equalization, and minimum distance decisions. Since the sample rate of the design at this point is a little more manageable, this processing will most likely be done in a DSP chip. This is desirable because these chips can be programmed in a high-level language such as C, giving the design engineer more flexibility. As mentioned in Chapter 4, however, there is probably room in the FPGA to do some of these kinds of processing tasks as well.

- Board design, layout, and fabrication: After the necessary DSP hardware has been designed and implemented, a printed circuit board (PCB), consisting of the necessary hardware, needs to be designed an implemented. This is the ultimate goal of the overall project of which the design in this report is but a small part. If a PCB can be designed, then it can be fabricated at a much smaller cost than a development board with the required functionality. This is an ideal situation for an educational setting.

5.4 Conclusion

In conclusion, an FPGA design that is flexible and portable has been designed for use in a software radio system. This design is not too large to fill a modern FPGA, so additional processing can be performed on the same device if desired. The hardware designed in this project processes the received signal straight from the ADC and decimates it to a more manageable sample rate where a DSP chip can finish the necessary processing in a more flexible manner, coinciding with the methodology of a software radio.

References

[1] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering.* Prentice Hall, 2002.

[2] F. J. Harris, *Multirate Signal Processing for Communication Systems.* Prentice Hall, 2004.

[3] M. Rice, *Digital Communications: A Discrete Time Approach.* Prentice Hall, 2008.

[4] T. Hentschel, *Sample Rate Conversion in Software Configurable Radios.* Artech House, 2002.

[5] T. Bose, *Digital Signal and Image Processing.* John Wiley and Sons, 2004.

[6] E. Hogenauer, "An economical class of digital filters for decimation and interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1981.

[7] *Understanding CIC Compensation Filters* [Online]. Available: `http://www.altera.com/literature/an/an455.pdf`.

[8] *Cascaded Integrator-Comb (CIC) Filter V3.0* [Online]. Available: `http://china.xilinx.com/ipcenter/catalog/logicore/docs/cic.pdf`.

[9] *Continuously Variable Fractional Rate Decimator* [Online]. Available: `http://www.xilinx.com/support/documentation/application_notes/xapp936.pdf`.

Appendices

Appendix A

Synthesis Options and Timing Constraints

A.1 Introduction

This appendix describes the synthesis options that were used to synthesize this design to the FPGA using the Xilinx software tools. This appendix also outlines and describes the methods that were used to apply timing constraints to the design. These timing constraints specify the period for the global clock signal and identify the paths in the design that can take multiple cycles of the global clock due to the downsampling operations present in the processing. These paths are called multi-cycle paths.

A.2 Synthesis Options

In order to get the design to synthesize, place, and route properly, several of the settings of the Xilinx 9.1i synthesizer, mapper, and router were changed. Table A.1 shows the settings that were modified from the default value in the Xilinx tool used.

The effect of the synthesis options shown in Table A.1 are outlined below:

- Ensure that the synthesizer utilizes the block RAM resources embedded in the fabric for the tap ROM modules.
- The keep hierarchy option tells the synthesizer to keep the hierarchy set up in the VHDL files instead of flattening the hierarchy and optimizing the boundaries.
- Shift register extraction was turned off because it utilized twice as many flip flops for the delay elements in the comb stages. The synthesizer can use LUT elements in each slice as shift registers, but reset logic cannot be included. Reset logic needed to be in the delay elements of the comb filters, so this option was turned off.

Table A.1: Modified implementation options.

Setting	Description	Default
Synthesis Options		
Optimization effort	high	low
Keep hierarchy	yes	no
RAM style	block	auto
ROM style	block	auto
Shift register extraction	off	on
Resource sharing	off	on
Register duplication	off	on
Equivalent register removal	off	on
MAP Options		
Placer extra effort	normal	none
Combinatorial logic optimization	on	off
Register duplication	on	off
Trim unconnected signals	off	on
Optimization strategy	balanced	area
PAR Options		
PAR effort (overall)	high	standard
Extra effort (highest PAR level only)	normal	none

- Equivalent register removal was turned off because the synthesizer optimized elements out of the tap ROM modules if there were two coefficients that happened to be the same stored in the same ROM.

- The other options were set to make the design more optimized and run a little faster.

The main effect of the MAP and PAR options was to turn the optimization effort of the tools up. The trim unconnected signals option of the MAP tool had to be turned off because of the quantizer modules. The mapper would see that some of the signals had no load and optimize the logic that created the signals away, effectively optimizing the entire design to nothing. This was obviously a bad thing, so the option that made the mapper do this was turned off.

A.3 Timing Constraints

In order to set constraints for the timing of the design, a user constraints file is input into the Xilinx software tool. The tool then uses this file when it performs a place and route. For this design, the period of the clock was constrained to meet the $100MHz$ sample rate. Because of the downsampling operations in both the CIC filter and the polyphase FIR filter, several multi-cycle path constraints also had to be set up. These constraints tell the Xilinx tool that there are paths that are allowed to operate slower than the global clock period constraint. The user constraint file for this design is *filt_cascade.ucf*. This file is shown below.

```
# global timing constraints
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 6 ns HIGH 50 %; # period is overconstrained a little
OFFSET = IN 5 ns BEFORE "clk"  ; # pad to setup constraint
OFFSET = OUT 7.5 ns AFTER "clk"  ;

# stepping level
CONFIG stepping = "2";

# global constraints on high-fanout, non clock nets
# IF KEEP HIERARCHY IS ENABLED, EN_OUT_1 NET DOESN'T EXIST
#NET "cic0/decimator/dsampler/en_out_1" MAXDELAY = 2 ns;
#NET "cic0/decimator/dsampler/en_out_1" MAXSKEW = 1.5 ns;
```

```
NET "cic0/decimator/dsampler/en_out" MAXDELAY = 2 ns;
NET "cic0/decimator/dsampler/en_out" MAXSKEW = 1.5 ns;

# timing groups for comb sections of cic filter
INST "cic0/decimator/outreg/outp_*" TNM = "dsample_out";
INST "cic0/combs/comb_gen[0].cmb1.first_cmb/outreg/outp_*" TNM = "comb0_out";
INST "cic0/combs/comb_gen[0].cmb1.first_cmb/delay/delayreg2/outp_*" TNM = "comb0sreg_out";
INST "cic0/combs/comb_gen[1].mid_cmbs.cmb_mids/outreg/outp_*" TNM = "comb1_out";
INST "cic0/combs/comb_gen[1].mid_cmbs.cmb_mids/delay/delayreg2/outp_*" TNM = "comb1sreg_out";
INST "cic0/combs/comb_gen[2].mid_cmbs.cmb_mids/outreg/outp_*" TNM = "comb2_out";
INST "cic0/combs/comb_gen[2].mid_cmbs.cmb_mids/delay/delayreg2/outp_*" TNM = "comb2sreg_out";
INST "cic0/combs/comb_gen[3].mid_cmbs.cmb_mids/outreg/outp_*" TNM = "comb3_out";
INST "cic0/combs/comb_gen[3].mid_cmbs.cmb_mids/delay/delayreg2/outp_*" TNM = "comb3sreg_out";
INST "cic0/combs/comb_gen[4].last_cmb.cmb_last/outreg/outp_*" TNM = "comb4_out";
INST "cic0/combs/comb_gen[4].last_cmb.cmb_last/delay/delayreg2/outp_*" TNM = "comb4sreg_out";

# timing groups for accumulator flip flops for fir filter
INST "fir0/fir/tap_gen[29].last_tap.accD_last/accumulator_*" TNM = FFS "accD29_acc";
INST "fir0/fir/tap_gen[28].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD28_acc";
INST "fir0/fir/tap_gen[27].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD27_acc";
INST "fir0/fir/tap_gen[26].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD26_acc";
INST "fir0/fir/tap_gen[25].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD25_acc";
INST "fir0/fir/tap_gen[24].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD24_acc";
INST "fir0/fir/tap_gen[23].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD23_acc";
INST "fir0/fir/tap_gen[22].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD22_acc";
INST "fir0/fir/tap_gen[21].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD21_acc";
INST "fir0/fir/tap_gen[20].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD20_acc";
INST "fir0/fir/tap_gen[19].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD19_acc";
INST "fir0/fir/tap_gen[18].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD18_acc";
INST "fir0/fir/tap_gen[17].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD17_acc";
INST "fir0/fir/tap_gen[16].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD16_acc";
INST "fir0/fir/tap_gen[15].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD15_acc";
INST "fir0/fir/tap_gen[14].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD14_acc";
INST "fir0/fir/tap_gen[13].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD13_acc";
INST "fir0/fir/tap_gen[12].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD12_acc";
INST "fir0/fir/tap_gen[11].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD11_acc";
INST "fir0/fir/tap_gen[10].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD10_acc";
INST "fir0/fir/tap_gen[9].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD9_acc";
INST "fir0/fir/tap_gen[8].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD8_acc";
INST "fir0/fir/tap_gen[7].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD7_acc";
INST "fir0/fir/tap_gen[6].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD6_acc";
INST "fir0/fir/tap_gen[5].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD5_acc";
INST "fir0/fir/tap_gen[4].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD4_acc";
INST "fir0/fir/tap_gen[3].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD3_acc";
INST "fir0/fir/tap_gen[2].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD2_acc";
INST "fir0/fir/tap_gen[1].mid_taps.accD_mid/accumulator_*" TNM = FFS "accD1_acc";
INST "fir0/fir/tap_gen[0].first_tap.accD_0/accumulator_*" TNM = FFS "accD0_acc";

# combine all groups above into one big timegroup
TIMEGRP "accum_regs" =   "accD29_acc" "accD28_acc" "accD27_acc" "accD26_acc"
                                "accD25_acc" "accD24_acc" "accD23_acc" "accD22_acc"
                                "accD21_acc" "accD20_acc" "accD19_acc" "accD18_acc"
                                "accD17_acc" "accD16_acc" "accD15_acc" "accD14_acc"
                                "accD13_acc" "accD12_acc" "accD11_acc" "accD10_acc"
                                "accD9_acc" "accD8_acc" "accD7_acc" "accD6_acc"
                                "accD5_acc" "accD4_acc" "accD3_acc" "accD2_acc"
                                "accD1_acc" "accD0_acc";

# timing groups for accumulator outputs in fir filter
INST "fir0/fir/tap_gen[29].last_tap.accD_last/acc_out_*" TNM = FFS "accD29_out";
INST "fir0/fir/tap_gen[28].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD28_out";
INST "fir0/fir/tap_gen[27].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD27_out";
INST "fir0/fir/tap_gen[26].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD26_out";
INST "fir0/fir/tap_gen[25].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD25_out";
INST "fir0/fir/tap_gen[24].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD24_out";
INST "fir0/fir/tap_gen[23].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD23_out";
INST "fir0/fir/tap_gen[22].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD22_out";
INST "fir0/fir/tap_gen[21].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD21_out";
INST "fir0/fir/tap_gen[20].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD20_out";
INST "fir0/fir/tap_gen[19].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD19_out";
INST "fir0/fir/tap_gen[18].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD18_out";
INST "fir0/fir/tap_gen[17].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD17_out";
INST "fir0/fir/tap_gen[16].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD16_out";
INST "fir0/fir/tap_gen[15].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD15_out";
INST "fir0/fir/tap_gen[14].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD14_out";
INST "fir0/fir/tap_gen[13].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD13_out";
INST "fir0/fir/tap_gen[12].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD12_out";
INST "fir0/fir/tap_gen[11].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD11_out";
INST "fir0/fir/tap_gen[10].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD10_out";
INST "fir0/fir/tap_gen[9].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD9_out";
INST "fir0/fir/tap_gen[8].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD8_out";
INST "fir0/fir/tap_gen[7].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD7_out";
INST "fir0/fir/tap_gen[6].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD6_out";
INST "fir0/fir/tap_gen[5].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD5_out";
INST "fir0/fir/tap_gen[4].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD4_out";
INST "fir0/fir/tap_gen[3].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD3_out";
INST "fir0/fir/tap_gen[2].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD2_out";
INST "fir0/fir/tap_gen[1].mid_taps.accD_mid/acc_out_*" TNM = FFS "accD1_out";
INST "fir0/fir/tap_gen[0].first_tap.accD_0/acc_out_*" TNM = FFS "accD0_out";
# combine all above groups into one big group
TIMEGRP "acc_outputs" =   "accD29_out" "accD28_out" "accD27_out" "accD26_out"
                                "accD25_out" "accD24_out" "accD23_out" "accD22_out"
                                "accD21_out" "accD20_out" "accD19_out" "accD18_out"
                                "accD17_out" "accD16_out" "accD15_out" "accD14_out"
                                "accD13_out" "accD12_out" "accD11_out" "accD10_out"
                                "accD9_out" "accD8_out" "accD7_out" "accD6_out"
                                "accD5_out" "accD4_out" "accD3_out" "accD2_out"
                                "accD1_out" "accD0_out";

# timing groups for tmp registers in accD modules of fir filter
INST "fir0/fir/tap_gen[29].last_tap.accD_last/tmp_*" TNM = FFS "accD29_tmp";
INST "fir0/fir/tap_gen[28].mid_taps.accD_mid/tmp_*" TNM = FFS "accD28_tmp";
INST "fir0/fir/tap_gen[27].mid_taps.accD_mid/tmp_*" TNM = FFS "accD27_tmp";
INST "fir0/fir/tap_gen[26].mid_taps.accD_mid/tmp_*" TNM = FFS "accD26_tmp";
INST "fir0/fir/tap_gen[25].mid_taps.accD_mid/tmp_*" TNM = FFS "accD25_tmp";
INST "fir0/fir/tap_gen[24].mid_taps.accD_mid/tmp_*" TNM = FFS "accD24_tmp";
INST "fir0/fir/tap_gen[23].mid_taps.accD_mid/tmp_*" TNM = FFS "accD23_tmp";
INST "fir0/fir/tap_gen[22].mid_taps.accD_mid/tmp_*" TNM = FFS "accD22_tmp";
INST "fir0/fir/tap_gen[21].mid_taps.accD_mid/tmp_*" TNM = FFS "accD21_tmp";
INST "fir0/fir/tap_gen[20].mid_taps.accD_mid/tmp_*" TNM = FFS "accD20_tmp";
INST "fir0/fir/tap_gen[19].mid_taps.accD_mid/tmp_*" TNM = FFS "accD19_tmp";
INST "fir0/fir/tap_gen[18].mid_taps.accD_mid/tmp_*" TNM = FFS "accD18_tmp";
INST "fir0/fir/tap_gen[17].mid_taps.accD_mid/tmp_*" TNM = FFS "accD17_tmp";
INST "fir0/fir/tap_gen[16].mid_taps.accD_mid/tmp_*" TNM = FFS "accD16_tmp";
INST "fir0/fir/tap_gen[15].mid_taps.accD_mid/tmp_*" TNM = FFS "accD15_tmp";
INST "fir0/fir/tap_gen[14].mid_taps.accD_mid/tmp_*" TNM = FFS "accD14_tmp";
INST "fir0/fir/tap_gen[13].mid_taps.accD_mid/tmp_*" TNM = FFS "accD13_tmp";
INST "fir0/fir/tap_gen[12].mid_taps.accD_mid/tmp_*" TNM = FFS "accD12_tmp";
INST "fir0/fir/tap_gen[11].mid_taps.accD_mid/tmp_*" TNM = FFS "accD11_tmp";
INST "fir0/fir/tap_gen[10].mid_taps.accD_mid/tmp_*" TNM = FFS "accD10_tmp";
INST "fir0/fir/tap_gen[9].mid_taps.accD_mid/tmp_*" TNM = FFS "accD9_tmp";
INST "fir0/fir/tap_gen[8].mid_taps.accD_mid/tmp_*" TNM = FFS "accD8_tmp";
INST "fir0/fir/tap_gen[7].mid_taps.accD_mid/tmp_*" TNM = FFS "accD7_tmp";
INST "fir0/fir/tap_gen[6].mid_taps.accD_mid/tmp_*" TNM = FFS "accD6_tmp";
INST "fir0/fir/tap_gen[5].mid_taps.accD_mid/tmp_*" TNM = FFS "accD5_tmp";
INST "fir0/fir/tap_gen[4].mid_taps.accD_mid/tmp_*" TNM = FFS "accD4_tmp";
INST "fir0/fir/tap_gen[3].mid_taps.accD_mid/tmp_*" TNM = FFS "accD3_tmp";
INST "fir0/fir/tap_gen[2].mid_taps.accD_mid/tmp_*" TNM = FFS "accD2_tmp";
INST "fir0/fir/tap_gen[1].mid_taps.accD_mid/tmp_*" TNM = FFS "accD1_tmp";
INST "fir0/fir/tap_gen[0].first_tap.accD_0/tmp_*" TNM = FFS "accD0_tmp";
# group all above groups into a big timegroup
TIMEGRP "accD_tmps" = "accD29_tmp" "accD28_tmp" "accD27_tmp" "accD26_tmp"
                      "accD25_tmp" "accD24_tmp" "accD23_tmp" "accD22_tmp"
                      "accD21_tmp" "accD20_tmp" "accD19_tmp" "accD18_tmp"
                      "accD17_tmp" "accD16_tmp" "accD15_tmp" "accD14_tmp"
                      "accD13_tmp" "accD12_tmp" "accD11_tmp" "accD10_tmp"
                      "accD9_tmp" "accD8_tmp" "accD7_tmp" "accD6_tmp"
                      "accD5_tmp" "accD4_tmp" "accD3_tmp" "accD2_tmp"
                      "accD1_tmp" "accD0_tmp";

# timing groups for tap output registers in fir filter
INST "fir0/fir/tap_gen[29].last_tap.reg_last/outp_*" TNM = FFS "firtap29_out";
INST "fir0/fir/tap_gen[28].mid_taps.reg_mid/outp_*" TNM = FFS "firtap28_out";
INST "fir0/fir/tap_gen[27].mid_taps.reg_mid/outp_*" TNM = FFS "firtap27_out";
INST "fir0/fir/tap_gen[26].mid_taps.reg_mid/outp_*" TNM = FFS "firtap26_out";
INST "fir0/fir/tap_gen[25].mid_taps.reg_mid/outp_*" TNM = FFS "firtap25_out";
INST "fir0/fir/tap_gen[24].mid_taps.reg_mid/outp_*" TNM = FFS "firtap24_out";
INST "fir0/fir/tap_gen[23].mid_taps.reg_mid/outp_*" TNM = FFS "firtap23_out";
INST "fir0/fir/tap_gen[22].mid_taps.reg_mid/outp_*" TNM = FFS "firtap22_out";
INST "fir0/fir/tap_gen[21].mid_taps.reg_mid/outp_*" TNM = FFS "firtap21_out";
INST "fir0/fir/tap_gen[20].mid_taps.reg_mid/outp_*" TNM = FFS "firtap20_out";
INST "fir0/fir/tap_gen[19].mid_taps.reg_mid/outp_*" TNM = FFS "firtap19_out";
INST "fir0/fir/tap_gen[18].mid_taps.reg_mid/outp_*" TNM = FFS "firtap18_out";
INST "fir0/fir/tap_gen[17].mid_taps.reg_mid/outp_*" TNM = FFS "firtap17_out";
INST "fir0/fir/tap_gen[16].mid_taps.reg_mid/outp_*" TNM = FFS "firtap16_out";
INST "fir0/fir/tap_gen[15].mid_taps.reg_mid/outp_*" TNM = FFS "firtap15_out";
INST "fir0/fir/tap_gen[14].mid_taps.reg_mid/outp_*" TNM = FFS "firtap14_out";
INST "fir0/fir/tap_gen[13].mid_taps.reg_mid/outp_*" TNM = FFS "firtap13_out";
INST "fir0/fir/tap_gen[12].mid_taps.reg_mid/outp_*" TNM = FFS "firtap12_out";
INST "fir0/fir/tap_gen[11].mid_taps.reg_mid/outp_*" TNM = FFS "firtap11_out";
INST "fir0/fir/tap_gen[10].mid_taps.reg_mid/outp_*" TNM = FFS "firtap10_out";
INST "fir0/fir/tap_gen[9].mid_taps.reg_mid/outp_*" TNM = FFS "firtap9_out";
INST "fir0/fir/tap_gen[8].mid_taps.reg_mid/outp_*" TNM = FFS "firtap8_out";
INST "fir0/fir/tap_gen[7].mid_taps.reg_mid/outp_*" TNM = FFS "firtap7_out";
INST "fir0/fir/tap_gen[6].mid_taps.reg_mid/outp_*" TNM = FFS "firtap6_out";
INST "fir0/fir/tap_gen[5].mid_taps.reg_mid/outp_*" TNM = FFS "firtap5_out";
INST "fir0/fir/tap_gen[4].mid_taps.reg_mid/outp_*" TNM = FFS "firtap4_out";
INST "fir0/fir/tap_gen[3].mid_taps.reg_mid/outp_*" TNM = FFS "firtap3_out";
INST "fir0/fir/tap_gen[2].mid_taps.reg_mid/outp_*" TNM = FFS "firtap2_out";
INST "fir0/fir/tap_gen[1].mid_taps.reg_mid/outp_*" TNM = FFS "firtap1_out";
INST "fir0/fir/tap_gen[0].first_tap.reg_0/outp_*" TNM = FFS "firtap0_out";
# group all above groups into one timegroup
TIMEGRP "tap_outputs" =   "firtap29_out" "firtap28_out" "firtap27_out" "firtap26_out"
                                "firtap25_out" "firtap24_out" "firtap23_out" "firtap22_out"
                                "firtap21_out" "firtap20_out" "firtap19_out" "firtap18_out"
                                "firtap17_out" "firtap16_out" "firtap15_out" "firtap14_out"
                                "firtap13_out" "firtap12_out" "firtap11_out" "firtap10_out"
                                "firtap9_out" "firtap8_out" "firtap7_out" "firtap6_out"
                                "firtap5_out" "firtap4_out" "firtap3_out" "firtap2_out"
                                "firtap1_out" "firtap0_out";

# timing groups for rom outputs in fir filter
### USE THESE GROUPS IF BLOCK RAMS ARE USED
INST "fir0/fir/tap_gen[0].first_tap.rom_0/Mrom_rom_out_mux00001" TNM = RAMS rom0_out;
INST "fir0/fir/tap_gen[1].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom1_out;
INST "fir0/fir/tap_gen[2].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom2_out;
INST "fir0/fir/tap_gen[3].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom3_out;
INST "fir0/fir/tap_gen[4].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom4_out;
INST "fir0/fir/tap_gen[5].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom5_out;
INST "fir0/fir/tap_gen[6].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom6_out;
INST "fir0/fir/tap_gen[7].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom7_out;
INST "fir0/fir/tap_gen[8].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom8_out;
INST "fir0/fir/tap_gen[9].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom9_out;
INST "fir0/fir/tap_gen[10].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom10_out;
INST "fir0/fir/tap_gen[11].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom11_out;
INST "fir0/fir/tap_gen[12].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom12_out;
INST "fir0/fir/tap_gen[13].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom13_out;
INST "fir0/fir/tap_gen[14].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom14_out;
INST "fir0/fir/tap_gen[15].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom15_out;
INST "fir0/fir/tap_gen[16].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom16_out;
INST "fir0/fir/tap_gen[17].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom17_out;
INST "fir0/fir/tap_gen[18].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom18_out;
INST "fir0/fir/tap_gen[19].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom19_out;
INST "fir0/fir/tap_gen[20].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom20_out;
INST "fir0/fir/tap_gen[21].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom21_out;
INST "fir0/fir/tap_gen[22].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom22_out;
INST "fir0/fir/tap_gen[23].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom23_out;
INST "fir0/fir/tap_gen[24].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom24_out;
INST "fir0/fir/tap_gen[25].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom25_out;
INST "fir0/fir/tap_gen[26].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom26_out;
INST "fir0/fir/tap_gen[27].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom27_out;
INST "fir0/fir/tap_gen[28].mid_taps.rom_mid/Mrom_rom_out_mux00001" TNM = RAMS rom28_out;
INST "fir0/fir/tap_gen[29].last_tap.rom_last/Mrom_rom_out_mux00001" TNM = RAMS rom29_out;
# group all above groups into one big timegroup
TIMEGRP "rom_outputs"  = "rom29_out" "rom28_out" "rom27_out" "rom26_out" "rom25_out"
                               "rom24_out" "rom23_out" "rom22_out" "rom21_out" "rom20_out"
                               "rom19_out" "rom18_out" "rom17_out" "rom16_out" "rom15_out"
                               "rom14_out" "rom13_out" "rom12_out" "rom11_out" "rom10_out"
                               "rom9_out" "rom8_out" "rom7_out" "rom6_out" "rom5_out"
                               "rom4_out" "rom3_out" "rom2_out" "rom1_out" "rom0_out";
```

```
### USE THESE GROUPS IF DISTRIBUTED ROMS ARE USED
#INST "fir0/fir/tap_gen[27].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom27_out";
#INST "fir0/fir/tap_gen[26].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom26_out";
#INST "fir0/fir/tap_gen[25].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom25_out";
#INST "fir0/fir/tap_gen[24].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom24_out";
#INST "fir0/fir/tap_gen[23].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom23_out";
#INST "fir0/fir/tap_gen[22].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom22_out";
#INST "fir0/fir/tap_gen[21].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom21_out";
#INST "fir0/fir/tap_gen[20].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom20_out";
#INST "fir0/fir/tap_gen[19].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom19_out";
#INST "fir0/fir/tap_gen[18].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom18_out";
#INST "fir0/fir/tap_gen[17].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom17_out";
#INST "fir0/fir/tap_gen[16].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom16_out";
#INST "fir0/fir/tap_gen[15].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom15_out";
#INST "fir0/fir/tap_gen[14].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom14_out";
#INST "fir0/fir/tap_gen[13].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom13_out";
#INST "fir0/fir/tap_gen[12].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom12_out";
#INST "fir0/fir/tap_gen[11].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom11_out";
#INST "fir0/fir/tap_gen[10].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom10_out";
#INST "fir0/fir/tap_gen[9].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom9_out";
#INST "fir0/fir/tap_gen[8].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom8_out";
#INST "fir0/fir/tap_gen[7].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom7_out";
#INST "fir0/fir/tap_gen[6].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom6_out";
#INST "fir0/fir/tap_gen[5].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom5_out";
#INST "fir0/fir/tap_gen[4].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom4_out";
#INST "fir0/fir/tap_gen[3].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom3_out";
#INST "fir0/fir/tap_gen[2].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom2_out";
#INST "fir0/fir/tap_gen[1].mid_taps.rom_mid/rom_out_*" TNM = FFS "rom1_out";
#INST "fir0/fir/tap_gen[0].first_tap.rom_0/rom_out_*" TNM = FFS "rom0_out";
## group all above groups into one big timegroup
#TIMEGRP "rom_outputs" = "rom27_out" "rom26_out" "rom25_out" "rom24_out"
#                         "rom23_out" "rom22_out" "rom21_out" "rom20_out"
#                         "rom19_out" "rom18_out" "rom17_out" "rom16_out"
#                         "rom15_out" "rom14_out" "rom13_out" "rom12_out"
#                         "rom11_out" "rom10_out" "rom9_out" "rom8_out"
#                         "rom7_out" "rom6_out" "rom5_out" "rom4_out"
#                         "rom3_out" "rom2_out" "rom1_out" "rom0_out";

# multi cycle path specifications for CIC filter
TIMESPEC "TS_cic_mcp1" = FROM "dsample_out"       TO "comb0_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp2" = FROM "comb0sreg_out"  TO "comb0_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp3" = FROM "comb0_out"      TO "comb1_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp4" = FROM "comb1sreg_out"  TO "comb1_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp5" = FROM "comb1_out"      TO "comb2_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp6" = FROM "comb2sreg_out"  TO "comb2_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp7" = FROM "comb2_out"      TO "comb3_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp8" = FROM "comb3sreg_out"  TO "comb3_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp9" = FROM "comb3_out"      TO "comb4_out" "TS_clk" * 125;
TIMESPEC "TS_cic_mcp10" = FROM "comb4sreg_out" TO "comb4_out" "TS_clk" * 125;

# multi cycle path specifications for FIR filter
TIMESPEC "TS_fir_mcp1" = FROM "comb4_out"      TO "accum_regs" "TS_clk" * 125;
TIMESPEC "TS_fir_mcp2" = FROM "rom_outputs"       TO "accum_regs" "TS_clk" * 125;
TIMESPEC "TS_fir_mcp3" = FROM "comb4_out"      TO "accD_tmps" "TS_clk" * 125;
TIMESPEC "TS_fir_mcp4" = FROM "rom_outputs"        TO "accD_tmps" "TS_clk" * 125;
TIMESPEC "TS_fir_mcp5" = FROM "acc_outputs"        TO "tap_outputs" "TS_clk" * 1000;

# assign output buffers to use fast slew rate drivers
NET "cascade_out<*>" SLEW = FAST;
#NET "cascade_out<1>" SLEW = FAST;
#NET "cascade_out<2>" SLEW = FAST;
#NET "cascade_out<3>" SLEW = FAST;
#NET "cascade_out<4>" SLEW = FAST;
#NET "cascade_out<5>" SLEW = FAST;
#NET "cascade_out<6>" SLEW = FAST;
#NET "cascade_out<7>" SLEW = FAST;
#NET "cascade_out<8>" SLEW = FAST;
#NET "cascade_out<9>" SLEW = FAST;
#NET "cascade_out<10>" SLEW = FAST;
#NET "cascade_out<11>" SLEW = FAST;
#NET "cascade_out<12>" SLEW = FAST;
#NET "cascade_out<13>" SLEW = FAST;
#NET "cascade_out<14>" SLEW = FAST;
#NET "cascade_out<15>" SLEW = FAST;
#NET "cascade_out<16>" SLEW = FAST;
#NET "cascade_out<17>" SLEW = FAST;
#NET "cascade_out<18>" SLEW = FAST;
#NET "cascade_out<19>" SLEW = FAST;
#NET "cascade_out<20>" SLEW = FAST;
#NET "cascade_out<21>" SLEW = FAST;
#NET "cascade_out<22>" SLEW = FAST;
#NET "cascade_out<23>" SLEW = FAST;
NET "ce_out" SLEW = FAST;
```

The first several lines of *filt_cascade.ucf* set up the period constraint of the global clock and set up the input to pad and pad to output constraints as well. The next lines of *filt_cascade.ucf* set up the muti-cycle path constraints of the design. The `TNM -- NET` syntax was used for this. This syntax uses the instantiation names of the synthesized nets in the design to create timing groups. After several of these groups are outlined for each path, the `TIMEGRP` statements group these individual groups into one big group. After each of these groups are set up, the `TIMESPEC -- FROM -- TO` statements define how long each path is allowed to take based on the global clock period constraint. This was the general method used to specify the multi-cycle paths in the design due to the downsampling operations performed. The last section of the file tells the MAP tool to use fast slew rate drivers for the output signals. This speeds up the pad to out timing specifications a great deal.

It is important to note here that since the multi-cycle path constraints are based on the instantiation names of the design, they could possibly change if the generic values of the design are changed. For example, if the number of stages in the CIC filter is modified, then some paths outlined in *filt_cascade.ucf* would not be constrained or not exist anymore. This was a main reason for including this appendix. If the structure of the filter cascade is modified, the user constraints file needs to be modified or the design will not be constrained properly and the MAP tool will not be able to optimize the design sufficiently.

A.4 Conclusion

The synthesis, MAP, and PAR options that were changed to make the design synthesize and route correctly were outlined in this appendix. Also, the timing specifications and the method for constraining the design was also outlined.

Appendix B
MATLAB Listings

B.1 Main Script

des2.m

```
%% des2.m
% Jake Talbot
% June 21, 2007
% des2.m (design #2) − this script is the second design of the intermediate
% frequency to baseband conversion process of the software radio receiver.
% des2.m is very similar to des1.m, except for the differences outlined
% below:
%
% The major difference of des2.m from des1.m is the fact that there are
% only two filters in the decimation cascade instead of three.  This is
% because it was found that at the passband edge of the CIC filter, there
% was only ¬0.5 dB of attenuation, so a CIC compensation filter was deemed
% not necessary.  That being the case, the cascade of a decimate by 2
% polyphase filter and a decimate by four polyphase filter in des1.m has
% been changed to a single decimate by eight polyphase filter in this
% script, des2.m.
%
% Another change is that this script writes the quantized filter
% coefficients and all other necessary inputs/outputs to several files to
% eventually be used in a VHDL testbench to test the design implementation.
%
% August 1, 2007
% Revision #1 : changed sample rate to 100 MHz and carrier frequency to
% 25 MHz in order to facilitate an easier implementation of the CIC filter
% on the FPGA.
%
% August 17, 2007
% Revision #2 : quantized the output of the cic filter to 26 bits instead
% of leaving it at full precision.  This is so I could test the polyphase
% filter without having to accomodate for the huge bit width coming from
% the output of the cic.
%
% September 8, 2007
% Revision #3 : quantized the output of the polyphase filter to 24 bits
% instead of leaving it at full precision.  This is because the next
% element in the signal processing chain (probably a DSP) will not be able
% to accomodate 50 output bits from the FPGA.
%
% January 7, 2008
% Revision #4 : reworked how the data was plotted. Instead of overwriting
% old figures with new ones and pausing the script, all the data gets
% plotted at the end in one subfigure.
%
% January 15, 2008
% Revision #5 : added the "delete *.txt" line to ensure that the newest
% versions of all the .txt files are used in the VHDL implementation.
%
% March 15, 2008
% Revision #6 : added some more plots that were included in my thesis.

clc;
clf;
close all;
clear all;
delete *.txt;

%% Set up design parameters
noise_flag = 0;          % flag for adding noise (1 = add noise, 0 = don't)
Fs_adc = 100e6;          % Sampling frequency coming from ADC (Hz)
R_cic = 125;             % factor that CIC decimates by
M = 2;                   % differential delay of CIC filter
N = 5;                   % number of stages in CIC filter
R_fir = 8;               % decimation factor for polyphase filter
Fo = 25e6;               % carrier frequency (Hz)
NFFT = 2^15;             % number of points in fft calculation
fcns = (0:NFFT - 1)*Fs_adc/NFFT;   % vector of continuous frequencies
NFFT2 = 2^18;            % number of points in fft calculation
fcns2 = (0:NFFT2 - 1)*Fs_adc/NFFT2;   % vector of continuous frequencies
Bin = 14;                % # of bits from ADC
Bcoeffs = 14;            % # of bits for filter coefficients
Bcicout = 26;            % # of bits to quantize output of cic to
Bfirout = 50;            % # of bits on the output of the polyphase fir filter
Bcascout = 24;           % # of bits after quantizing output of fir filter
fwant = 50e3;            % one sided bandwidth (Hz) of signal of interest
funwant = 75e3;          % frequency where another unwanted signal resides in spectrum

%% Create a test signal to be demodulated down to baseband
F1 = Fo + funwant;           % first unwanted frequency component
F2 = Fo - funwant;           % second unwanted frequency component
Fo_1 = Fo + fwant;           % first of the wanted frequency component
Fo_2 = Fo - fwant;           % second of the wanted frequency component
T1 = 1/F1;                   % period of highest frequency component
n = 0:1/Fs_adc:15000*T1;     % time index for cosines
n = n*(180/pi);
test_sig = cosd(2*pi*F1*n) + cosd(2*pi*F2*n) + cosd(2*pi*Fo_1*n) ...
           + cosd(2*pi*Fo_2*n) + cosd(2*pi*(F1 + 100e3)*n);
if(noise_flag)               % add noise into signal
    var = 0.01;              % variance of Gaussian white noise
    noise = var * randn(1, length(n));  % vector of noise
    test_sig = test_sig + noise;        % add noise to modulated signal
```

```
end

% quantize test_sig and write to a file for VHDL testbench
test_sig1 = test_sig/(max(abs(test_sig)) + 1);
test_sig_q = floor((test_sig1 * 2^(Bin - 1)));
fileprint(['test_sigB', num2str(Bin)], test_sig_q, Bin);

%% Demodulate the test signal
mod_sig = zeros(1, length(n));      % allocate vector
mod_sig(1:4:length(n)) = 1;                 % cos(pi/2*(0:4:end)) = 1
mod_sig(3:4:length(n)) = -1;                % cos(pi/2*(2:4:end)) = -1

bb_sig = mod_sig .* test_sig;       % calculate dp result
bb_sig_q = mod_sig .* test_sig_q;   % calculate quantized result

% write quantized demodulated signal to a file for VHDL testbench
fileprint(['demod_sigB', num2str(Bin)], bb_sig_q, Bin);

%% Design decimating CIC filter
Bacc = Bin + ceil(N*log2(R_cic*M));    % # of bits required in accumulators
hsingle = ones(1, R_cic*M);     % impulse response of a single CIC
hcic = hsingle;
for j = 1:N - 1;
    hcic = conv(hcic, hsingle); % cascade N stages of single CIC filters
end

%% Filter and downsample signal w/ CIC
bb_sig = conv(bb_sig, hcic);    % filter the baseband signal
bb_sig_q = conv(bb_sig_q, hcic);    % filter the quantized signal

int_sig1 = downsample(bb_sig, R_cic);   % downsample the filtered signal
int_sig1_q = downsample(bb_sig_q, R_cic);

% quantize the values of int_sig1_q down to 26 bits
int_sig1_q = floor((int_sig1_q / 2^Bcicout));

% write the quantized, decimated signal to a file for VHDL testbenches
fileprint(['dec125B', num2str(Bcicout)], int_sig1_q, Bcicout);

%% Design decimate by 8 polyphase filter
Fc = 7.291967e6;        % cns. cutoff frequency (Hz)
L = 240;                % # of filter taps
hprac = lp_fir(Fc, Fs_adc, L);  % design an lpf

% quantize and write filter coefficients to a file for VHDL implementation
hprac = hprac/(max(abs(hprac)) + 1);
hprac_q = floor((hprac * 2^(Bcoeffs - 1)));
fileprint(['polycoeffsL', num2str(L), 'B', num2str(Bcoeffs)], hprac_q, Bcoeffs);

%% Filter and downsample signal w/ decimating polyphase filter
fin_sig = polyfilt(int_sig1, hprac, R_fir, Bcoeffs, 0);    %polyphase decimate
fin_sig_q = polyfilt(int_sig1_q, hprac_q, R_fir, Bcoeffs, 1);

% quantize the polyphase filter output to Bcascout
fin_sig_q = floor(fin_sig_q / 2^(Bfirout - Bcascout + 2));
```

```
% write final quantized downsampled signal to file for VHDL testbench
fileprint(['out_sigB', num2str(Bcascout)], fin_sig_q, Bcascout);

%% Plot spectra of all intermediate signals

% plot the spectrum of the modulated signal
figure(1);      % open a new figure
plot(fcns2, abs(fft(test_sig, NFFT2)), 'linewidth', 2);
xlabel('Continuous Frequency (Hz)', 'fontsize', 14);
ylabel('|FFT|', 'fontsize', 14);
title('Modulated Signal', 'fontsize', 14);
xlim([(25e6 - 100e3), (25e6 + 100e3)]);

% plot the spectrum of the demodulated signal
figure(2);
plot(fcns2, abs(fft(bb_sig, NFFT2)), 'linewidth', 2);
title('Demodulated Signal', 'fontsize', 14);
xlabel('Continuous Frequency', 'fontsize', 14);
ylabel('|FFT|', 'fontsize', 14);
xlim([0, 150e3]);

% plot the downsampled and CIC filtered signal
figure(3);
plot(fcns, abs(fft(int_sig1, NFFT)), 'linewidth', 2);
xlabel('Continuous Frequency (Hz)', 'fontsize', 14);
ylabel('|FFT|', 'fontsize', 14);
title(['CIC decimated signal, R_{cic} = ' num2str(R_cic)], 'fontsize', 14);
xlim([0, 125*150e3]);

% plot the polyphase FIR decimated signal
figure(4);
plot(fcns, abs(fft(fin_sig, NFFT)), 'linewidth', 2);
xlabel('Continuous Frequency (Hz)', 'fontsize', 14);
ylabel('|FFT|', 'fontsize', 14);
title(['Polyphase filtered signal, R_{fir} = ' num2str(R_fir)...
       ', R_{combined} = ' num2str(R_cic*R_fir)], 'fontsize', 14);
xlim([0, 1000*125e3]);

% plot the frequency response of the CIC filter
figure(5);
Hcic = abs(fft(hcic, NFFT2));
plot(fcns2, 10*log10(Hcic/max(Hcic)), 'linewidth', 2);
title(['Frequency Response of CIC Filter, N = ', num2str(N)], ...
        'fontsize', 14);
xlabel('Continuous Frequency (Hz)', 'fontsize', 14);
ylabel('|FFT| (dB)', 'fontsize', 14);
xlim([0 7e5]);
ylim([-80 5]);
```

B.2 Functions Used

fileprint.m

```
function [] = fileprint(name, vector, numbits)
% Jake Talbot
% June 21, 2007
% fileprint.m - this script is used to open a file with the filename
% "name.txt", and depending
% on the both flag, writes the values to the opened file.
%
% INPUT ARGUMENTS
%   - name: a string that will be the name of the file written to.
%   - vector: a vector that is to be written to the file.
%   - numbits: the number of bits that the vector has been quantized to.

fname = [name, '.txt'];         % create filename
fid = fopen(fname, 'wt');       % open file to write to in text mode
fprintf(fid, ['%-', num2str(numbits), '.0f\n'], vector);
fclose(fid);    % close the file
```

lp_fir.m

```
function [lpf_coeffs] = lp_fir(Fc, Fs, L)
% Jake Talbot
% June 20, 2007
% lp_fir.m (low pass fir) - this function designs a lowpass filter using a
% chebyshev window with 65 or more dB attenuation in the stopband.
% INPUT ARGUMENTS
%   - Fc: the desired cutoff frequency in Hz.  Note that this is not the
%   frequency that specifies the passband or the stopband, it is the
%   frequency at which you want the filter to start attenuating.  It is
%   normally picked to be (Fstop - Fpass)/2 + Fpass, where Fpass is the
%   desired passband frequency and Fstop is the desired stopband frequency.
%   - Fs: the sample frequency of the system in Hz
%   - L: desired number of filter taps, must be divisible by 2
% OUTPUT ARGUMENTS
%   - lpf_coeffs: the coefficients of the designed filter

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Design impulse response of ideal, causal LPF
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fc = 2*(Fc/Fs);         % digital rep. of cutoff frequency
hid = fc * sinc(fc * ((0:L - 1) - L/2));    % ideal LPF impulse response

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Window ideal impulse response
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
win = window(@chebwin, L, 65);      % create a Chebyshev window
win = win';
lpf_coeffs = hid .* win;    % window ideal filter to make a practical one
```

polyfilt.m

```
function [out_sig] = polyfilt(in_sig, fir_filt, D, Bcoeffs, print)
% Jake Talbot
% June 20, 2007
% polyfilt.m (polyphase filter) - this MATLAB function is used to take a
% normal FIR filter, decompose into it's decimating polyphase
% representation, and filter and downsample the incoming signal.
% INPUT ARGUMENTS
%   - in_sig - the input signal to be decimated
%   - fir_filt - the coefficients of the normal fir filter
%   - D - the desired decimation factor
%   - Bcoeffs - the number of bits in quantized version of filter
%   - print - flag for printing the coeffs to several files or not (1 =
%     print, 0 = don't)
%
% OUTPUT ARGUMENTS
%   - out_sig - the decimated signal
%
% polyfilt.m uses polyphase type one decomposition and a commutator instead
% of a tapped delay line followed by downsamplers.  It is also important to
% note that polyfilt.m does not scale the output to recover from the
% implied scaling by D that occurs in the downsampling process.
%
% REVISIONS
% August 13, 2007
%   added code to print the polyphase filter's coefficients to several
%   files named "romi.txt" where the i is for the column of the polyphase
%   filter array polyfilts.  To do this, the number of bits in the
%   coefficients (Bcoeffs), and a flag to decide whether or not to print to
%   the files (print) were added to this function as input arguments.
%
% September 5, 2007
%   changed the filtering operation from the filter function to the conv
%   function so I could more thoroughly test the VHDL implementation.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set up general parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
N = length(fir_filt);   % find length of filter
fir_filt = [fir_filt, zeros(1, D*ceil(N/D)-N)]; % ensure that D divides N
N = length(fir_filt);   % find new length of filter
L = N/D;                % length of polyphase filters
M = length(in_sig);     % find length of input signal
in_sig = [in_sig, zeros(1, D*ceil(M/D) - M)];   % ensure that D divides K
M = length(in_sig);     % find new length of input signal
```

```
K = M/D;                   % # of columns in commutator array
out_len = K + L - 1;       % length of output array
polyfilts = zeros(D, L);      % array for polyphase filters, each row is a filter
polyin = zeros(D, K);         % array for input samples from commutator
polyout = zeros(D, out_len);       % array to hold polyphase filtered outputs

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decompose original filter into type I polyphase filters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ind = 1;              % index for original filter
for i = 1:L           % scan through columns
    for j = 1:D       % scan through rows
        polyfilts(j, i) = fir_filt(ind);      % deal out samples down columns
        ind = ind + 1;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Print columns of polyfilts to file(s) for VHDL implementation based upon
% the print flag
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if(print)
    for i = 1:L           % scan through columns of polyfilt
        fname = ['rom', num2str(i - 1)];
        printbinary( polyfilts(:,i), Bcoeffs, fname );  % print coeffs to file in binary
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Deal out input samples like a commutator would, each row goes with a
% corresponding polyphase filter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ind = 1;
polyin(1, 1) = in_sig(ind);     % first row of first column is only non-zero value in column
ind = ind + 1;
for i = 2:K           % loop through columns
    for j = D:-1:1    % loop through rows backwards
        polyin(j, i) = in_sig(ind);     % deal out samples commutator style
        ind = ind + 1;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Filter the dealt out input signals with the polyphase filters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1:D      % filter arrays row - wise
    polyout(i, :) = conv(polyfilts(i, :), polyin(i, :));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sum the filtered array column - wise to compute final output
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
out_sig = zeros(1, out_len);       % vector to hold the output signal
ind = 1;                      % index for output signal
for i = 1:out_len             % loop through columns
```

```
    for j = 1:D     % loop through rows
        out_sig(ind) = out_sig(ind) + polyout(j, i);    % add up all numbers in column
    end
    ind = ind + 1;
end                 % signal is ready to be output by the end.
```

printbinary.m

```
function [ ] = printbinary( x, B, fname )
%PRINTBINARY print values of vector x in binary form into file fname.txt
%   PRINTBINARY takes values of vector x, converts them to B bit two's complement
%   binary format, and then prints the binary representation to the file
%   "fname.txt".  This is used to print the coefficients of the
%   decimating polyphase filter(s) in the intermediate frequency to baseband
%   conversion process to a file so that they can be used to load the ROM
%   modules in the VHDL implementation.
%
%   INPUT ARGUMENTS
%       - x - the vector that you desire to print in binary format
%       - B - the number of bits in the binary representation
%       - fname - a string representing the file where you want the data to
%         be written to

fid = fopen([fname, '.txt'], 'wt'); % open a file in write mode
x = dec2bin(x, B);  % convert elements of vector into B - bit binary
[row col] = size(x);    % calculate number of rows and columns in binary vector
for i = 1:row  % loop through rows
    for j = 1:col
        if(x(i,j) == '/')   % if number is negative, convert '/'s to 1's
            x(i,j) = '1';
        end
        fprintf(fid, '%c', x(i,j)); % print elements of vector to a file, one character at a time
    end
    fprintf(fid, '%c\n', ' ');  % print a newline character to the file
end
fclose(fid);    % close the file
```

Appendix C
VHDL Listings

C.1 Top Level of the Hierarchy

filt_cascade.vhd

```
----------------------------------------------------------------------------------
-- Company: Utah State University
-- Engineer: Jake Talbot
--
-- Create Date:    10:36:44 08/20/2007
-- Design Name:      filt_cascade
-- Module Name:    filt_cascade - struc
-- Project Name:     cicwpolyfir2.ise
-- Target Devices: virtex4 - sx family or virtex5 - sx family
-- Tool versions:  ISE 9.1i
-- Description:
--      filt_cascade.vhd - This VHDL entity is the top level in the hierarchy of
--      the intermediate frequency to baseband signal processing chain.  Some of
--      the features of this design are as follows:  To perform downsampling, clock
--      enable signals are generated, not an internally divided clock.  Generic
--      adders, subtractors, and multipliers are used to facilitate easy changes of
--  the design.
--
--  Generics
--      B_casc_in  : # of bits on input of cascade, which is the input of the CIC.
--      B_cic_out  : # of bits on the output of the CIC, also the accumulator bit
--              width for the integrators and combs.  The accumulator bit width,
--              and thus the # of output bits is determined by the following
--              equation :
--              B_cic_out = B_casc_in + ceil(N * log2(D * M))
--              where M is the differential delay (fixed at 2), N is the number
--              of stages in the CIC (generic = N_stages_cic), and D is the
--              decimation factor (generic = D_cic).  If this equation is
--              followed, than the accumulators will not overflow.
--      D_cic : The decimation factor of the CIC, note that this value affects
--              the bit width of the accumulators and thus the output of the CIC.
--      N_stages_cic : # of stages in the CIC filter, this generic also influences
--                     the number of bits required in the accumulators of the CIC.
--      N_poly_taps : # of taps in the polyphase filter. Notice that this value
--              depends on the length of the original prototype filter and the
--              decimation factor of the polyphase filter (see D_fir description
--              for eqn.).
--      B_poly_coeffs : # of bits in the polyphase filter coefficients.
--      B_poly_in : # of bits on the input of the polyphase filter.  This is
--              effectively what you want the CIC output to be quantized to before
--              the polyphase filter processes the data.
--      B_poly_out : # of bits on the output of the polyphase filter.  Full precision
--               is usually maintained to avoid overflow errors.
--      B_casc_out : # of bits on the output of the filter cascade.  Since full
--               precision is maintained through the polyphase filter,
--               quantization is usually required.
--      D_fir : Decimation factor for the polyphase filter.  Note that the number of
--          taps in the filter is given by the equation:
--          n_taps = ceil(n_prototype_taps/D_fir)
--          where n_taps is the number of taps in the polyphase filter,
--          n_prototype_taps is the number of taps in the original filter, and
--          D_fir is the decimation factor.
--      B_rom_addr : number of bits in the coefficient ROM modules in the polyphse
--               filter.  This is given by the following equation:
--               B_rom = ceil(log2(D_fir))
--               This equation holds because there are D_fir elements in each
--               tap ROM of the polyphase filter.
--
--  Ports
--      clk : The global clock for the filter cascade. The paramaters above and/or
--            the synthesizer, MAP, and PAR settings have to be such that the design
--            can be run with a global clock period of at least 100 MHz.
--      rst_h : The global synchronous, asserted high reset input. This reset input
--          resets all flip-flops and RAM blocks.
--      cascade_in : The input to the filter cascade. This is the output of the ADC
--               in the receiver of the software radio.
--      ce_out : This is a clock enable output of the cascade. This clock enable
--           signal is meant to drive other clock enabled flip-flops that are
--           meant to operate at the same rate as the output of the filter
--           cascade.
--      cascade_out : The output of the filter cascade. This is usually the
--                quantized polyphase filter output.
--
--  Internal Signals
--      demod_out : The output of the demodulator, input to CIC filter.
--      tmp_cic_out : Output of CIC filter, input to first quantizer.
--      fir_in : Input to polyphase FIR filter, output of first quantizer.
--      fir_out : Output of polyphase FIR filter, input to second quantizer.
--      tmp_ce1 : Clock enable output of CIC filter, used to drive flip flops that
--            operate at the same sample rate as the output of the CIC filter.
--      tmp_ce2 : Clock enable output of polyphase FIR filter, used to drive flip
--            flops that operate at the final, slowest sample rate. This is
--            also the ce_out signal of the block.
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 1 - November 4, 2007 - I added the demodulator to the design.
--      This module is what takes the output of the ADC and demodulates the signal.
--     This added an extra cycle of latency to the system.  To correct this,
--      the value that the counter in the start_cnt module of the cic/decimator
--      module counts up to was changed.
--
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity filt_cascade is
  generic ( B_casc_in : integer := 12;
        B_cic_out : integer := 52;
        D_cic : integer := 125;
        N_stages_cic : integer := 5;
        N_poly_taps : integer := 30;
        B_poly_coeffs : integer := 14;
        B_poly_in : integer := 26;
        B_poly_out : integer := 50;
        B_casc_out : integer := 24;
        D_fir : integer := 8;
        B_rom_addr : integer := 3);
  port    ( clk : in STD_LOGIC;
        rst_h : in STD_LOGIC;
        cascade_in : in STD_LOGIC_VECTOR (B_casc_in - 1 downto 0);
        ce_out :  out STD_LOGIC;
        cascade_out : out STD_LOGIC_VECTOR (B_casc_out - 1 downto 0));
end filt_cascade;

architecture struc of filt_cascade is

  component demod_i
    generic ( B : integer);
    port    ( clk : in STD_LOGIC;
          rst_h : in STD_LOGIC;
          inp : in STD_LOGIC_VECTOR (B - 1 downto 0);
          outp : out STD_LOGIC_VECTOR (B - 1 downto 0));
  end component demod_i;

  component cic
    generic ( B_cic_in : integer;
          B_cic_out : integer;
          D_factor : integer;
          N : integer);
    port    ( clk : in STD_LOGIC;
          rst_h : in STD_LOGIC;
          cic_in : in STD_LOGIC_VECTOR (B_cic_in - 1 downto 0);
          ce_out : out STD_LOGIC;
          cic_out : out STD_LOGIC_VECTOR (B_cic_out - 1 downto 0));
  end component cic;

  component poly_fir_top
    generic ( N_taps : integer;
          N_cic_stages : integer;
          B_coeffs : integer;
          B_input : integer;
          B_output : integer;
          D_cic : integer;
          D_fir : integer;
          B_rom_addr : integer);
    port    ( clk : in STD_LOGIC;
          rst_h : in STD_LOGIC;
          ce_in : in STD_LOGIC;
          insample : in STD_LOGIC_VECTOR (B_input - 1 downto 0);
          ce_out : out STD_LOGIC;
          outsample : out STD_LOGIC_VECTOR (B_output - 1 downto 0));
  end component poly_fir_top;

  component quantizer
    generic ( B_in : integer;
          B_out : integer);
    port    ( inp : in STD_LOGIC_VECTOR (B_in - 1 downto 0);
          outp : out STD_LOGIC_VECTOR (B_out - 1 downto 0));
  end component quantizer;

  signal demod_out : STD_LOGIC_VECTOR (B_casc_in - 1 downto 0);
  signal tmp_cic_out : STD_LOGIC_VECTOR (B_cic_out - 1 downto 0);
  signal fir_in : STD_LOGIC_VECTOR (B_poly_in - 1 downto 0);
  signal fir_out : STD_LOGIC_VECTOR (B_poly_out - 1 downto 0);
  signal tmp_ce1, tmp_ce2 : STD_LOGIC;

begin

  demod0 : demod_i
    generic map ( B => B_casc_in)
    port map ( clk => clk, rst_h => rst_h, inp => cascade_in, outp => demod_out);

  cic0 : cic
    generic map ( B_cic_in => B_casc_in, B_cic_out => B_cic_out, D_factor => D_cic,
          N => N_stages_cic)
    port map ( clk => clk, rst_h => rst_h, cic_in => demod_out, ce_out => tmp_ce1,
               cic_out => tmp_cic_out);

  q0 : quantizer
    generic map ( B_in => B_cic_out, B_out => B_poly_in)
    port map ( inp => tmp_cic_out, outp => fir_in);

  fir0 : poly_fir_top
    generic map ( N_taps => N_poly_taps, N_cic_stages => N_stages_cic, B_coeffs => B_poly_coeffs,
          B_input => B_poly_in, B_output => B_poly_out, D_cic => D_cic, D_fir => D_fir,
                B_rom_addr => B_rom_addr)
    port map ( clk => clk, rst_h => rst_h, ce_in => tmp_ce1,
               insample => fir_in, ce_out => tmp_ce2, outsample => fir_out);
  q1 : quantizer
    generic map ( B_in => B_poly_out, B_out => B_casc_out)
    port map ( inp => fir_out, outp => cascade_out);

  ce_out <= tmp_ce2;

end struc;

configuration filt_cascade_con of filt_cascade is
  for struc
    for demod0 : demod_i
      use entity work.demod_i(behv);
    end for;

    for cic0 : cic
      use configuration work.cic_con;
    end for;

    for q0, q1 : quantizer
      use entity work.quantizer(behv);
    end for;

    for fir0 : poly_fir_top
      use configuration work.poly_fir_top_con;
    end for;
  end for;
end configuration filt_cascade_con;
```

C.2 Second Level of the Hierarchy

demod_i.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    10:26:13 07/06/2007
-- Design Name:      filt_cascade
-- Module Name:    demod_i - behv
-- Project Name:     cicwpolyfir2
-- Target Devices: virtex 4 - SX family
-- Tool versions: ISE 9.1i
-- Description: This VHDL entity is a demodulator for the in phase branch of the
--      DSP processing chain.  Since the carrier frequency is 1/4 the sampling frequency,
--      the modulator is multiplierless because
--                    { 1,  n = 0,4,8,...
--          cos(2*pi/n) = { 0,  n = 1,3,5,...
--                    { -1, n = 2,6,10,...
--      Below are descriptions of the generics, ports, and internal signals of this
--       module.
--
--   Generics
--       B : The # of bits on the input and output of the module.
--                    Since this module is the first thing in the processing chain of the
--           filter cascade, this should match the # of bits that the ADC outputs.
--
--   Ports
--       clk : The clock signal for the flip flops of this module. These flip-flops
--             must operate at the same sample rate as the ADC.
--       rst_h : The global synchronous, asserted high reset input.
--       inp : The input signal to the demodulator, this is the output of the
--             ADC.
--       outp : The output signal of the demodulator, this is the input to the CIC
--              filter.
--
-- Internal Signals
--       cnt : This signal is used as a counter to count the # of clock cycles.
--             Based on this, the demodulator decides whether to output a zero
--             (n odd), the input (n = 0,4,8,...), or the input multiplied by -1
--             (n= 2,6,10,...). Notice that the multiply by -1 is simply a 2's
--             complement negation of the input.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity demod_i is
  generic   ( B : integer := 12);       -- # of bits from ADC
  port      ( clk  : in STD_LOGIC;
          rst_h  : in STD_LOGIC;
          inp : in STD_LOGIC_VECTOR(B - 1 downto 0);
          outp : out STD_LOGIC_VECTOR(B - 1 downto 0));
end demod_i;

architecture behv of demod_i is
signal cnt : STD_LOGIC_VECTOR(1 downto 0);  -- variable to count clk cycles and/or time index n
begin
  demod_cos : process(clk)
  begin
    if clk'event and clk = '1' then
      if rst_h = '1' then
```

```
        cnt <= "00";
    outp <= (others => '0');
      else
        case cnt is
          when "00" => outp <= inp;
          when "01" | "11" => outp <= (others => '0');
          when "10" => outp <= (not inp) + 1;        -- 2's compliment negate the number
          when others => outp <= (others => 'Z');
        end case;
        cnt <= cnt + 1;  -- increment cnt on posedge of clock
      end if;   -- rst_h = '1'
    end if; -- clk'event and clk = '1'
  end process demod_cos;
end behv;
```

cic.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    07:36:20 08/03/2007
-- Design Name:
-- Module Name:    cic - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is the top level of the CIC design hierarchy.
--      This design uses a clock enable signal and clock enabled registers rather
--      than an internally divided clock.  Also, this design uses generic adder/subtractors
--      using the VHDL '+' or '-' operators. This CIC filter is designed with a differential
--      delay of 2. Below are descriptions of the generics  and ports of the cic filter.
--
--  Generics
--      B_cic_in  : # of bits on the input of the cic
--      B_cic_out : # of bits on the output of the cic.  This is also the number of
--                            bits that are in the accumulator registers of both the integrator
--              stages and comb stages.  This value is determined by the following
--              equation:
--
--                              Bacc = B_cic_in + ceil(N * log2(D_factor * M))
--
--                  where N is the number of stages in the filter, D_factor is the
--              decimation rate, and M is the differential delay (fixed at 2 for
--              this design).
--      D_factor  : This is the decimation factor for the CIC.
--      N : This is the number of stages in both the integrator and comb cascades
--          of the CIC filter..
--
--  Ports
--      clk : The global clock for the design.  The design parameters above have to
--            be such that the design can run at 100 MHz or more (10 ns clk period)
--      rst_h : A global synchronous, asserted high reset.  This goes to all flip flops
--          of the CIC filter.
--      cic_in : The samples for the CIC to process.  These are input at the high sample
--           rate.
--      ce_out : This is a clock enable signal that is used to drive clock enabled registers
--           after the CIC.  This signal is generated by the downsample module in this
--           design.
--      cic_out : The output of the CIC filter.  These are output at the divided sample rate.
--            Also, there is a latency (of divided clock cycles) that is equal to the number
--            of stages N.  This is due to the fact that the comb section of the CIC is pipelined.
--
--  Internal Signals
--      int_casc_out : Output of the integrator cascade, input to downsample module.
--      downsample_out : Output of the downsample module, these are output at the
--               divided sample rate. These are input to the comb cascade.
--      tmpce : Clock enable output from downsample block. Used to drive flip flops
--              in the comb cascade and any other flip flops that operate at the
--          downsampled rate of the CIC. This is also the ce_out output of the
--          CIC filter.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity cic is
  generic   ( B_cic_in : integer := 12;
          B_cic_out : integer := 52;
          D_factor : integer := 125;
          N : integer := 5);
  port      ( clk : in STD_LOGIC;
          rst_h : in STD_LOGIC;
          cic_in : in STD_LOGIC_VECTOR (B_cic_in - 1 downto 0);
          ce_out : out STD_LOGIC;
          cic_out : out STD_LOGIC_VECTOR (B_cic_out - 1 downto 0));
end cic;

architecture struc of cic is

  -- component declarations
  component int_cascade
    generic ( B_int_in : integer;
          B_int_out : integer;
          N : integer);
    port    ( clk : in STD_LOGIC;
          rst_h : in STD_LOGIC;
          casc_in : in STD_LOGIC_VECTOR (B_int_in - 1 downto 0);
          casc_out : out STD_LOGIC_VECTOR (B_int_out - 1 downto 0));
  end component int_cascade;

  component downsample
    generic ( B_downsample : integer;
          D_factor : integer;
          N : integer);
    port    ( clk : in STD_LOGIC;
          rst_h : in STD_LOGIC;
          data_in : in STD_LOGIC_VECTOR (B_downsample - 1 downto 0);
          clk_en : out STD_LOGIC;
          data_out : out STD_LOGIC_VECTOR (B_downsample - 1 downto 0));
  end component downsample;

  component comb_cascade
    generic ( B_comb_cascade : integer;
          N : integer);
    port    ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
          ce : in STD_LOGIC;
          casc_in : in STD_LOGIC_VECTOR (B_comb_cascade - 1 downto 0);
          casc_out : out STD_LOGIC_VECTOR (B_comb_cascade - 1 downto 0));
  end component comb_cascade;

  -- internal signals
  signal int_casc_out : STD_LOGIC_VECTOR(B_cic_out - 1 downto 0);   -- output of integrator stages
  signal downsample_out : STD_LOGIC_VECTOR(B_cic_out - 1 downto 0); -- output of downsample block
  signal tmpce : STD_LOGIC; -- ce output of downsample block

begin   -- arch. struc of cic

  -- instantiate integrator stages
  integrators : int_cascade
    generic map (B_int_in => B_cic_in, B_int_out => B_cic_out, N => N)
    port map (clk => clk, rst_h => rst_h, casc_in => cic_in, casc_out => int_casc_out);

  -- instantiate downsample block
  decimator : downsample
    generic map ( B_downsample => B_cic_out, D_factor => D_factor, N => N)
    port map (clk => clk, rst_h => rst_h, data_in => int_casc_out, clk_en => tmpce,
            data_out => downsample_out);

  -- instantiate comb stages
  combs : comb_cascade
    generic map (B_comb_cascade => B_cic_out, N => N)
    port map (clk => clk, rst_h => rst_h, ce => tmpce, casc_in => downsample_out,
            casc_out => cic_out);

  -- assign the clock enable output to the internal signal tmpce
  ce_out ≤ tmpce;
```

```
end struc;

-- configuration declaration for cic entity
configuration cic_con of cic is
  for struc

    for integrators : int_cascade
      use configuration work.int_cascade_con;
    end for;

    for decimator : downsample
      use configuration work.downsample_con;
    end for;

    for combs : comb_cascade
      use configuration work.comb_cascade_con;
    end for;

  end for;
end configuration cic_con;
```

quantizer.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    10:38:16 08/20/2007
-- Design Name:
-- Module Name:    quantizer - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is used to quantize both the full precision CIC
--      filter output and the full precision polyphase FIR filter output. The
--      quantization method used here is truncation. Below are descriptions of the
--      generics, ports, and internal signals that are used in this entity.
--
--  Generics
--      B_in : The bit width of the input, either the CIC output or the polyphase
--             FIR filter output.
--      B_out : The bit width of the input after truncation.
--
--  Ports
--      inp : The full precision input.
--      outp : The quantized representation of the input signal.
--
--  Internal Signals
--      There are no internal signals in this entity.
--
```

```
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity quantizer is
    generic ( B_in : integer := 52;
                    B_out : integer := 26);
    port        ( inp : in STD_LOGIC_VECTOR ( B_in - 1 downto 0);
                    outp : out STD_LOGIC_VECTOR ( B_out - 1 downto 0));
end quantizer;

architecture behv of quantizer is
begin

    -- truncate the input to B_out bits, taking the MSB's
    outp <= inp(B_in - 1 downto (B_in - B_out));

end behv;
```

poly_fir_top.vhd

```
------------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    14:29:56 08/08/2007
-- Design Name:
-- Module Name:    poly_fir_top - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a structural description of the polyphase FIR
--       filter that was designed for this project. This entity consists of several
--       controllers. One to control the tap ROM addresses, one to generate a clock
--       enable signal for the processing after the decimation of the FIR filter,
--       and one to generate an enable signal for the clock enable controller. This
--       entity also includes the actual FIR filter structure that performs the
--       filtering and decimation. Below are descriptions of the generics, ports,
--      and internal signals that this entity uses.
--
--  Generics
--      N_taps : The Number of taps in the FIR filter.
--      N_cic_stages : The Number of stages in the preceding CIC filter. This is
--                 needed to account for the latency inherent in the pipelined
--                 comb cascade in the CIC.
--      B_coeffs : The bit width of the filter coefficients used.
--      B_input : The bit width of the input to the polyphase FIR filter.
--      B_output : The full precision bit width of the output of the polyphase FIR
--             filter.
--      D_cic : The decimation factor of the CIC filter.
--      D_fir : The decimation factor of the polyphase FIR filter.
--      B_rom_addr : The number of bits required for the address of the tap ROMs.
--               This is dependent on the generic D_fir, since the # of elements
--               in each tap rom is equal to D_fir. Thus this generic can be
--               calculated by the following equation :
--
--               B_rom_addr = ceil(log2(D_fir))
--
--  Ports
--      clk : The global clock signal.
--      rst_h : The global, synchronous, asserted high reset signal.
--      ce_in : The clock enable signal used for logic that needs to operate at
--          the CIC output rate.
--      insample : The input samples, they are input to the polyphase FIR filter at
--             the CIC output rate.
--      ce_out : The clock enable signal used for logic that needs to operate at the
--           polyphase filter output rate.
--      outsample : The output of the polyphase FIR filter. This is full precision,
--              also, any logic that potentially follows this output needs to
--              be driven by the clock enable signal ce_out.
--
--  Internal  Signals
--      tmp_ce : The clock enable signal used to drive flip flops that need to
--           operate at the polyphase FIR filter output rate. This is also
--           the ce_out output of this entity.
--      tmp_en : The enable signal used to tell the ce_ctrl entity when to start
--           counting clock signals. This is needed to account for the latency
--           of the comb cascade of the CIC filter.
--      addr : The address signal that is generated by the addr_ctrl entity. This
--             signal is used to address data in the tap ROMs of the fir_taps
--             entity.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity poly_fir_top is
  generic   ( N_taps : integer := 30;
                N_cic_stages : integer := 5;
                B_coeffs : integer := 14;
                B_input : integer := 26;
                B_output : integer := 48;
                D_cic : integer := 10;
                D_fir : integer := 8;
                B_rom_addr : integer := 3);
  Port          ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                ce_in : in STD_LOGIC;
                insample : in STD_LOGIC_VECTOR (B_input - 1 downto 0);
                ce_out : out STD_LOGIC;
                outsample : out STD_LOGIC_VECTOR (B_output - 1 downto 0));
end poly_fir_top;

architecture struc of poly_fir_top is

  -- component declarations
  component ce_ctrl
    generic ( D_cic : integer;
                D_fir : integer);
    port    ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                en_in : in STD_LOGIC;
                ce_out : out STD_LOGIC);
  end component ce_ctrl;

  component fir_taps
    generic ( N_taps : integer;
                D_factor : integer;
                B_coeffs : integer;
                B_input : integer;
                B_output : integer;
                B_rom_addr : integer);
    port    ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                ce_fast : in STD_LOGIC;
                ce_slow : in STD_LOGIC;
                fir_in : in STD_LOGIC_VECTOR (B_input - 1 downto 0);
                rom_addr : in STD_LOGIC_VECTOR (B_rom_addr - 1 downto 0);
                fir_out : out STD_LOGIC_VECTOR (B_output - 1 downto 0));
  end component fir_taps;

  component addr_ctrl
    generic ( B_addr : integer);
    port    ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
```

```
                  en_in : in STD_LOGIC;
                  ce_in : in STD_LOGIC;
                  addr_out : out STD_LOGIC_VECTOR (B_addr − 1 downto 0));
  end component addr_ctrl;

  component en_ctrl
    generic ( N_cic_stages : integer);
    port    ( clk : in STD_LOGIC;
                  rst_h : in STD_LOGIC;
                  ce_in : in STD_LOGIC;
                  en_out : out STD_LOGIC);
  end component en_ctrl;

  -- internal signals
  signal tmp_ce : STD_LOGIC;
  signal addr : STD_LOGIC_VECTOR (B_rom_addr − 1 downto 0);
  signal tmp_en : STD_LOGIC;

begin

  ctrl0 : en_ctrl
    generic map ( N_cic_stages => N_cic_stages)
    port map ( clk => clk, rst_h => rst_h, ce_in => ce_in, en_out => tmp_en);

  ctrl1 : ce_ctrl
    generic map ( D_cic => D_cic, D_fir => D_fir)
    port map ( clk => clk, rst_h => rst_h, en_in => tmp_en, ce_out => tmp_ce);

  ctrl2 : addr_ctrl
    generic map ( B_addr => B_rom_addr)
    port map ( clk => clk, en_in => tmp_en, ce_in => ce_in, rst_h => rst_h, addr_out => addr);

  fir : fir_taps
    generic map ( N_taps => N_taps, D_factor => D_fir, B_coeffs => B_coeffs,
                  B_input => B_input, B_output => B_output, B_rom_addr => B_rom_addr)
    port map    ( clk => clk, rst_h => rst_h, ce_fast => ce_in, ce_slow => tmp_ce,
                  fir_in => insample, rom_addr => addr, fir_out => outsample);

  ce_out ≤ tmp_ce;

end struc;

configuration poly_fir_top_con of poly_fir_top is
  for struc
    for ctrl1 : ce_ctrl
      use entity work.ce_ctrl(behv);
    end for;

    for ctrl2 : addr_ctrl
      use entity work.addr_ctrl(behv);
    end for;

    for fir : fir_taps
      use configuration work.fir_taps_con;
    end for;
  end for;
end configuration poly_fir_top_con;
```

C.3 Third Level of the Hierarchy

int_cascade.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    11:36:29 07/30/2007
-- Design Name:
-- Module Name:    int_cascade - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a structural description of the integrator
--      cascade of the CIC filter.  This entity resides in the second level of the
--      hierarchy of the design.  This cascade sign extends the input to the CIC
--      to the number of bits on the output (thus the accumulator bit width) and then
--      cascades N integrator sections together.  Below is a description of the generics and
--      ports for this entity.
--
--  Generics
--      B_int_in : # of bits on the input to the integrator cascade, this is the
--                      number of bits on the input to the filter cascade.
--      B_int_out : # of bits on the output of the integrator cascade.
--      N : # of integrator stages to cascade together.
--
--  Ports
--      clk : Global clock for entire CIC, the registers in the integrator stages
--              don't need to be clock enabled registers because they need to operate
--            at the same sample rate of the ADC.
--      rst_h : Global synchronous, asserted high reset signal.
--      casc_in : Input to the integrator cascade.
--      casc_out : Output of the integrator cascade.
--
--  Internal Signals
--      intrn_sig : This is an array of standard logic vectors that are equal in
--              length to the accumulator bit width. Each row of the array is
--              used as an output of an integrator stage.
--      sgn_ext_in : This signal is the input to the cascade sign extended to the
--               accumulator bit width.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity int_cascade is
  generic       ( B_int_in : integer := 12;
                  B_int_out : integer := 52;
                  N : integer := 5);
  port          ( clk : in STD_LOGIC;
                  rst_h : in STD_LOGIC;
                  casc_in : in STD_LOGIC_VECTOR (B_int_in - 1 downto 0);
                  casc_out : out STD_LOGIC_VECTOR (B_int_out - 1 downto 0));
end int_cascade;

architecture struc of int_cascade is

  -- component declarations
  component integrator
    generic ( B_int : integer);
    port    ( clk : in STD_LOGIC;
                  rst_h : in STD_LOGIC;
                  int_in : in STD_LOGIC_VECTOR (B_int - 1 downto 0);
                  int_out : out STD_LOGIC_VECTOR (B_int - 1 downto 0));
  end component integrator;

  component signextend
    generic ( Bin : integer;
                  Bout : integer);
    port    ( input : in STD_LOGIC_VECTOR (Bin - 1 downto 0);
                  output : out STD_LOGIC_VECTOR (Bout - 1 downto 0));
  end component signextend;

  -- internal signals
  subtype intrn_bus is STD_LOGIC_VECTOR(B_int_out - 1 downto 0);
  type intrn_arr is array (0 to N - 2) of intrn_bus;
  signal intrn_sig : intrn_arr;
  signal sgn_ext_in : intrn_bus;

begin   -- arch. struc of int_cascade
    -- use the "if generate" form of generate statement to generate N stages of integrators
  casc_gen : for J in 0 to N - 1 generate

    first_int : if J = 0 generate
      sgn_ext1 : signextend     -- need to sign extend the input
        generic map (Bin => B_int_in, Bout => B_int_out)
        port map (input => casc_in, output => sgn_ext_in);
      integrator1 : integrator  -- instantiate an integrator to operate on sign extended input
        generic map (B_int => B_int_out)
        port map (clk => clk, rst_h => rst_h, int_in => sgn_ext_in, int_out => intrn_sig(J));
    end generate first_int;

    mid_ints : if J > 0 and J < N - 1 generate
      ints_mid : integrator -- cascade integrators together
        generic map (B_int => B_int_out)
        port map (clk => clk, rst_h => rst_h, int_in => intrn_sig(J - 1), int_out => intrn_sig(J));
    end generate mid_ints;

    last_int : if J = N - 1 generate
      int_last : integrator -- assign output of last integrator to output of block
        generic map (B_int => B_int_out)
        port map (clk => clk, rst_h => rst_h, int_in => intrn_sig(J - 1), int_out => casc_out);
    end generate last_int;

  end generate casc_gen;
end struc;

-- configuration declaration for int_cascade entity
configuration int_cascade_con of int_cascade is
  for struc

    for casc_gen(0)
      for first_int
        for sgn_ext1 : signextend
          use entity work.signextend(behv);
        end for;

        for integrator1 : integrator
          use configuration work.integrator_con;
        end for;
      end for;
    end for;

    for casc_gen(1 to N - 2)
      for mid_ints
        for ints_mid : integrator
          use configuration work.integrator_con;
        end for;
      end for;
    end for;

    for casc_gen(N - 1)
      for last_int
        for int_last : integrator
          use configuration work.integrator_con;
        end for;
      end for;
    end for;

  end for;
end configuration int_cascade_con;
```

downsample.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    13:37:51 08/02/2007
-- Design Name:
-- Module Name:    downsample - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a structural description of the downsample block.
--      The downsample block consists of two controllers/counters.  The first controller
--      (start_ctrl) that is used is needed to grab the correct sample from the integrator
--      cascade upon a reset.  Since there are multiple integrator stages, the first sample
--      that the downsampler needs to keep is not the first sample that comes in after a reset.
--      Start_ctrl simply counts the number of cycles equivalent to the number of integrator stages
--      of the incoming clock and then outputs an   enable signal to the clk_ctrl block.
--      The second controller in the downsample block is the clk_ctrl block.  This
--      block counts the number of samples equivalent to D_factor and outputs a clock enable
--      signal to enable the registers after the downsampler when the counter wraps to zero.
--      With these two controllers, downsampling can be achieved by simply registering the
--      input to the downsampler block with a clock enabled register that is driven by the
--      enable output of the clk_ctrl controller.  Below is a description of the generics,
--      ports, and internal signals of the downsample block.
--
--  Generics
--      B_downsample : # of bits on input and output of downsample block.
--      D_factor : Factor that downsample block is to decimate by.
--      N : # of stages in integrator and comb cascade.
--
--  Ports
--      clk : Global clock, used to generate clock enable signal based on D_factor.
--      rst_h : Global synchronous, asserted high reset signal.
--      data_in : Data coming from integrator stages, only every D_factor-th sample is
--            passed on to the comb stages.
--      clk_en : clock enable output to drive comb cascade registers.  These registers
--           operate at the incoming clock rate divided by D_factor.
--      data_out : Data output from the downsample block.  This is simply every D_factor-th
--             input sample.
--
-- Internal Signals
--      fsm_en : Output of start_ctrl, this signal tells the clk_ctrl module when to
--           start counting cycles of the fast, global clock.
--      tmp_en : Output of the clk_ctrl module. This is the clk_en output of the
--           downsample block.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
```

```
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity downsample is
  generic    ( B_downsample : integer := 52;
                   D_factor : integer := 125;
                   N : integer := 5);
  port       ( clk : in STD_LOGIC;
                   rst_h : in STD_LOGIC;
                   data_in : in STD_LOGIC_VECTOR (B_downsample - 1 downto 0);
                   clk_en : out STD_LOGIC;
                   data_out : out STD_LOGIC_VECTOR (B_downsample - 1 downto 0));
end downsample;

architecture struc of downsample is

  -- component declarations
  component clk_ctrl
    generic ( D : integer);
    port    ( clk : in STD_LOGIC;
                   rst_h : in STD_LOGIC;
                   en_in : in STD_LOGIC;
                   en_out : out STD_LOGIC);
  end component clk_ctrl;

  component start_ctrl
    generic ( N : integer);
    port    ( clk : in STD_LOGIC;
                   rst_h : in STD_LOGIC;
                   en : out STD_LOGIC);
  end component start_ctrl;

  component ce_reg
    generic ( B_ce_reg : integer);
    port    ( clk : in STD_LOGIC;
                   rst_h : in STD_LOGIC;
                   ce : in STD_LOGIC;
                   inp : in STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0);
                   outp : out STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0));
  end component ce_reg;

  -- internal signals
  signal fsm_en : STD_LOGIC;        -- enable signal for the clock enable state machine
  signal tmp_en : STD_LOGIC;        -- clock enable output for divided clock

begin
```

```
  -- instantiate a startup counter to enable clock divider state machine
  start_cnt : start_ctrl
    generic map (N => N)
    port map (clk => clk, rst_h => rst_h, en => fsm_en);

  -- instantiate an enabled clock divider state machine.  This module outputs
  -- the clock enable signal for all registers in the comb cascade.
  dsampler : clk_ctrl
    generic map (D => D_factor)
    port map (clk => clk, rst_h => rst_h, en_in => fsm_en, en_out => tmp_en);

  -- register the output of the downsample block with a clock enabled register,
  -- effectively downsampling the signal by D_factor
  outreg : ce_reg
    generic map (B_ce_reg => B_downsample)
    port map (clk => clk, rst_h => rst_h, ce => tmp_en, inp => data_in, outp => data_out);

  -- assign output clock enable signal to internal signal tmp_en
  clk_en <= tmp_en;

end struc;

-- configuration declaration for downsample entity
configuration downsample_con of downsample is
  for struc

    for start_cnt : start_ctrl
      use entity work.start_ctrl(behv);
    end for;

    for dsampler : clk_ctrl
      use entity work.clk_ctrl(behv);
    end for;

    for outreg : ce_reg
      use entity work.ce_reg(behv);
    end for;

  end for;
end configuration downsample_con;
```

comb_cascade.vhd

```
---------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    07:11:28 08/03/2007
-- Design Name:
-- Module Name:    comb_cascade - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a structural description of the comb cascade
--      of the CIC filter. The comb_cascade block uses the "if genererate" form of the
--      generate statement to cascade N stages of comb sections together.  Note that
--      these comb stages are implemented with a constant differential delay of 2.
--      This comb cascade has pipeline registers at the output to reduce the combinational
--      path of the cascade.  Below are descriptions of the generics, ports, and Internal
--      signals that are used in this entity.
--
--  Generics
--      B_comb_cascade : # of bits on input and output of comb_cascade block.
--      N : # of comb stages to cascade together.
--
--  Ports
--      clk : Global clock signal.  The comb stages only have to operate at the rate
--            of this clock divided by D_factor.
--      rst_h : Global synchronous, asserted high reset signal
--      ce : Clock enable signal.  This is output from the downsample block and drives
--           all registers in the comb cascade.
--      casc_in : Input to the comb cascade, output from the downsample block.
--      casc_out : Output of the comb cascade, this is the output of the entire CIC
--             filter.
--
--  Internal Signals
--      intrn_sig : This signal is an array of standard logic vectors. Each row
--              of this array is used as an output of a comb entity.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity comb_cascade is
  generic   ( B_comb_cascade : integer := 52;
                  N : integer := 5);
  port      ( clk : in STD_LOGIC;
                  rst_h : in STD_LOGIC;
                  ce : in STD_LOGIC;
                  casc_in : in STD_LOGIC_VECTOR (B_comb_cascade - 1 downto 0);
                  casc_out : out STD_LOGIC_VECTOR (B_comb_cascade - 1 downto 0));
end comb_cascade;
```

```
architecture struc of comb_cascade is

  -- component declarations
  component comb
    generic ( B_comb : integer);
    port    ( clk  : in STD_LOGIC;
              rst_h  : in STD_LOGIC;
              ce  : in STD_LOGIC;
              comb_in  : in STD_LOGIC_VECTOR (B_comb - 1 downto 0);
              comb_out  : out STD_LOGIC_VECTOR (B_comb - 1 downto 0));
  end component comb;

  -- internal signals
  subtype intrn_bus is STD_LOGIC_VECTOR(B_comb_cascade - 1 downto 0);
  type intrn_arr is array (0 to N - 2) of intrn_bus;
  signal intrn_sig : intrn_arr;

begin   -- arch. struc of comb_cascade

  comb_gen : for K in 0 to N - 1 generate

    cmb1 : if K = 0 generate
      first_cmb  : comb
        generic map (B_comb => B_comb_cascade)
        port map (clk => clk, rst_h => rst_h, ce => ce,
                  comb_in => casc_in, comb_out => intrn_sig(K));
    end generate cmb1;

    mid_cmbs : if K > 0 and K < N - 1 generate
      cmb_mids : comb
        generic map (B_comb => B_comb_cascade)
        port map (clk => clk, rst_h => rst_h, ce => ce,
                  comb_in => intrn_sig(K - 1), comb_out => intrn_sig(K));
    end generate mid_cmbs;

    last_cmb  : if K = N - 1 generate
      cmb_last  : comb
        generic map (B_comb => B_comb_cascade)
        port map (clk => clk, rst_h => rst_h, ce => ce,
                  comb_in => intrn_sig(K - 1), comb_out => casc_out);
    end generate last_cmb;

  end generate comb_gen;

end struc;

-- configuration declaration for comb_cascade entity
configuration comb_cascade_con of comb_cascade is
  for struc

    for comb_gen(0)
      for cmb1
        for first_cmb  : comb
          use configuration work.comb_con;
        end for;
      end for;
    end for;

    for comb_gen(1 to N - 1)
      for mid_cmbs
        for cmb_mids : comb
          use configuration work.comb_con;
        end for;
      end for;
    end for;

    for comb_gen(N - 1)
      for last_cmb
        for cmb_last : comb
          use configuration work.comb_con;
        end for;
      end for;
    end for;

  end for;
end configuration comb_cascade_con;
```

fir_taps.vhd

```
-----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    14:36:40 08/08/2007
-- Design Name:
-- Module Name:    fir_taps - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This entity is a structural description of a transposed direct
--      form FIR filter. To make this a polyphase decimating filter, tap ROMs
--      containing D_factor coefficients are inserted and the output of the tap
--      multipliers are accumulated for D_factor cycles. The coefficients are
--      stored in the tap ROMs in such a way that for each cycle, the equivalent
--      FIR filter is a polyphase representation of the prototype filter. The
--      accumulators make it so the filter outputs at a decimated rate. To store the
--      coefficients in the tap ROMs correctly, a MATLAB script (polyfilt.m) deals
--      the prototype filter coefficients out correctly and then writes these
--      coefficient values to several text files that are then used to initialize the
--      tap ROMs. Below are descriptions of the generics, ports, and internal
--      internal signals used in this entity.
--
--  Generics
--      N_taps : The number of taps in the FIR filter.
--      D_factor : The factor at which the filter will decimate.
--      B_coeffs : The bit width of the coefficients stored in the tap ROMs.
--      B_input : The bit width of the input to the polyphase FIR filter.
--      B_output : The bit width of the output of the polyphase FIR filter. This
--             is usually made to keep full precision through the multipliers
--             and the accumulators. This is also the bit width of the
--             accumulators.
--      B_rom_addr : The number of bits in the tap ROM address inputs.
--
--  Ports
--      clk : The global clock signal.
--      rst_h : The global, synchronous, asserted high reset signal
--      ce_fast : The clock enable signal that drives flip flops operating
--            at the CIC filter output rate.
--      ce_slow : The clock enable signal that drives flip flops operating
--            at the polyphase FIR filter output rate.
--      fir_in : The input samples to the polyphase FIR filter.
--      rom_addr : The address used to fetch data from the tap ROMs.
--      fir_out : The output of the polyphase FIR filter. This is usually kept
--            at full precision.
--
--  Internal Signals
--      intrn_rom_sig : This signal is an array of standard logic vectors (SLVs) that
--              are used in the generate for loop as outputs of the
--              tap ROMs. These are input to the tap multipliers along with the
--              input signal, fir_in.
--      intrn_mult_sig : This signal is an array of SLVs that are used in the generate
--               for loop as outputs of the tap multipliers. These are then
--               input to the accumulators.
--      intrn_accD_sig : This signal is an array of SLVs that are used in the generate
--               for loop as outputs of the accumulators. These are then input
--               to the tap adders along with the register outputs from the
--               previous tap.
--      intrn_reg_sig : This signal is an array of SLVs used as outputs of the tap
--              registers used in the generate for loop. These are input
--              to the tap adders along with the accumulator outputs.
--      intrn_add_sig : This signal is an array of SLVs used as outputs of the
--              tap adders used in the generate for loop. These are then
--              input to the tap registers.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity fir_taps is
  generic   ( N_taps : integer := 30;
                D_factor : integer := 8;
                B_coeffs : integer := 14;
                B_input : integer := 26;
                B_output : integer := 40;
                B_rom_addr : integer := 3);
  port      ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                ce_fast : in STD_LOGIC;
                ce_slow : in STD_LOGIC;
                fir_in : in STD_LOGIC_VECTOR (B_input - 1 downto 0);
                rom_addr : in STD_LOGIC_VECTOR (B_rom_addr - 1 downto 0);
                fir_out : out STD_LOGIC_VECTOR (B_output - 1 downto 0));
end fir_taps;

architecture struc of fir_taps is

  -- component declarations
  component coeffrom
    generic ( N_elems : integer := 8;
                B_data : integer := 14;
                B_addr : integer := 3;
                fname : string := "rom0.txt");
    port    ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                en : in STD_LOGIC;
                addr : in STD_LOGIC_VECTOR (B_addr - 1 downto 0);
                rom_out : out STD_LOGIC_VECTOR (B_data - 1 downto 0));
  end component coeffrom;

  component gen_mult
    generic ( B_mult_in1 : integer;
                B_mult_in2 : integer);
    port    ( in1 : in STD_LOGIC_VECTOR (B_mult_in1 - 1 downto 0);
                in2 : in STD_LOGIC_VECTOR (B_mult_in2 - 1 downto 0);
                prod : out STD_LOGIC_VECTOR ((B_mult_in1 + B_mult_in2) - 1 downto 0));
  end component gen_mult;

  component accum_D
    generic ( D_factor : integer;
                B_acc_in : integer;
                B_acc_out : integer);
    port    ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                ce_in : in STD_LOGIC;
                acc_in : in STD_LOGIC_VECTOR (B_acc_in - 1 downto 0);
                acc_out : out STD_LOGIC_VECTOR (B_acc_out - 1 downto 0));
  end component accum_D;

  component ce_reg
    generic ( B_ce_reg : integer);
    port    ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                ce : in STD_LOGIC;
                inp : in STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0);
                outp : out STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0));
  end component ce_reg;

  component gen_adder
    generic ( B_adder : integer);
    port    ( add1 : in STD_LOGIC_VECTOR (B_adder - 1 downto 0);
                add2 : in STD_LOGIC_VECTOR (B_adder - 1 downto 0);
                sum : out STD_LOGIC_VECTOR (B_adder - 1 downto 0));
  end component gen_adder;

  -- internal signals
  subtype rom_bus is STD_LOGIC_VECTOR (B_coeffs - 1 downto 0);
  subtype mult_out is STD_LOGIC_VECTOR ((B_coeffs + B_input) - 1 downto 0);
  subtype accD_out is STD_LOGIC_VECTOR (B_output - 1 downto 0);

  type rom_sig is array (0 to N_taps - 1) of rom_bus;
  type mult_sig is array (0 to N_taps - 1) of mult_out;
  type accD_sig is array (0 to N_taps - 1) of accD_out;
  type reg_sig is array (0 to N_taps - 2) of accD_out;
  type add_sig is array (1 to N_taps - 1) of accD_out;

  signal intrn_rom_sig : rom_sig;
  signal intrn_mult_sig : mult_sig;
  signal intrn_accD_sig : accD_sig;
  signal intrn_reg_sig : reg_sig;
  signal intrn_add_sig : add_sig;

begin   -- arch. struc of fir_taps

  tap_gen : for J in 0 to N_taps - 1 generate
  begin

    first_tap : if J = 0 generate
    begin
      rom_0 : coeffrom
        generic map ( N_elems => D_factor, B_data => B_coeffs, B_addr => B_rom_addr,
                      fname => ("rom" & integer'image(N_taps - J - 1) & ".txt"))
        port map ( clk => clk, rst_h => rst_h, en => ce_fast, addr => rom_addr,
                   rom_out => intrn_rom_sig(J));

      mult_0 : gen_mult
        generic map ( B_mult_in1 => B_input, B_mult_in2 => B_coeffs)
        port map ( in1 => fir_in, in2 => intrn_rom_sig(J), prod => intrn_mult_sig(J));

      accD_0 : accum_D
        generic map ( D_factor => D_factor, B_acc_in => (B_coeffs + B_input),
                      B_acc_out => B_output)
        port map ( clk => clk, rst_h => rst_h, ce_in => ce_fast,
                   acc_in => intrn_mult_sig(J), acc_out => intrn_accD_sig(J));

      reg_0 : ce_reg
        generic map ( B_ce_reg => B_output)
        port map ( clk => clk, rst_h => rst_h, ce => ce_slow,
                   inp => intrn_accD_sig(J), outp => intrn_reg_sig(J));
    end generate first_tap;

    mid_taps : if J > 0 and J < N_taps - 1 generate
    begin
      rom_mid : coeffrom
        generic map ( N_elems => D_factor, B_data => B_coeffs, B_addr => B_rom_addr,
                      fname => ("rom" & integer'image(N_taps - J - 1) & ".txt"))
        port map ( clk => clk, rst_h => rst_h, en => ce_fast, addr => rom_addr,
                   rom_out => intrn_rom_sig(J));

      mult_mid : gen_mult
        generic map ( B_mult_in1 => B_input, B_mult_in2 => B_coeffs)
        port map ( in1 => fir_in, in2 => intrn_rom_sig(J), prod => intrn_mult_sig(J));

      accD_mid : accum_D
        generic map ( D_factor => D_factor, B_acc_in => (B_coeffs + B_input),
                      B_acc_out => B_output)
        port map ( clk => clk, rst_h => rst_h, ce_in => ce_fast,
                   acc_in => intrn_mult_sig(J), acc_out => intrn_accD_sig(J));

      add_mid : gen_adder
        generic map ( B_adder => B_output)
        port map ( add1 => intrn_accD_sig(J), add2 => intrn_reg_sig(J - 1),
                   sum => intrn_add_sig(J));

      reg_mid : ce_reg
        generic map ( B_ce_reg => B_output)
        port map ( clk => clk, rst_h => rst_h, ce => ce_slow, inp => intrn_add_sig(J),
                   outp => intrn_reg_sig(J));
    end generate mid_taps;

    last_tap : if J = N_taps - 1 generate
    begin
      rom_last : coeffrom
        generic map ( N_elems => D_factor, B_data => B_coeffs, B_addr => B_rom_addr,
                      fname => ("rom" & integer'image(N_taps - J - 1) & ".txt"))
        port map ( clk => clk, rst_h => rst_h, en => ce_fast, addr => rom_addr,
                   rom_out => intrn_rom_sig(J));

      mult_last : gen_mult
        generic map ( B_mult_in1 => B_input, B_mult_in2 => B_coeffs)
        port map ( in1 => fir_in, in2 => intrn_rom_sig(J), prod => intrn_mult_sig(J));

      accD_last : accum_D
        generic map ( D_factor => D_factor, B_acc_in => (B_coeffs + B_input),
                      B_acc_out => B_output)
        port map ( clk => clk, rst_h => rst_h, ce_in => ce_fast,
                   acc_in => intrn_mult_sig(J), acc_out => intrn_accD_sig(J));

      add_last : gen_adder
        generic map ( B_adder => B_output)
        port map ( add1 => intrn_accD_sig(J), add2 => intrn_reg_sig(J - 1),
                   sum => intrn_add_sig(J));
```

```
      reg_last : ce_reg
        generic map ( B_ce_reg => B_output)
        port    map ( clk => clk, rst_h => rst_h, ce => ce_slow, inp => intrn_add_sig(J),
                      outp => fir_out);
    end generate last_tap;

  end generate tap_gen;

end struc;

configuration fir_taps_con of fir_taps is
  for struc

    for tap_gen(0)
      for first_tap
        for rom_0 : coeffrom
          use entity work.coeffrom(behv);
        end for;

        for mult_0 : gen_mult
          use entity work.gen_mult(behv);
        end for;

        for accD_0 : accum_D
          use entity work.accum_D(behv);
        end for;

        for reg_0 : ce_reg
          use entity work.ce_reg(behv);
        end for;
      end for;
    end for;

    for tap_gen(1 to N_taps - 2)
      for mid_taps
        for rom_mid : coeffrom
          use entity work.coeffrom(behv);
        end for;

        for mult_mid : gen_mult
          use entity work.gen_mult(behv);
        end for;

        for accD_mid : accum_D
          use entity work.accum_D(behv);
        end for;

        for add_mid : gen_adder
          use entity work.gen_adder(behv);
        end for;

        for reg_mid : ce_reg
          use entity work.ce_reg(behv);
        end for;
      end for;
    end for;

    for tap_gen(N_taps - 1)
      for last_tap
        for rom_last : coeffrom
          use entity work.coeffrom(behv);
        end for;

        for mult_last : gen_mult
          use entity work.gen_mult(behv);
        end for;

        for accD_last : accum_D
          use entity work.accum_D(behv);
        end for;

        for add_last : gen_adder
          use entity work.gen_adder(behv);
        end for;

        for reg_last : ce_reg
          use entity work.ce_reg(behv);
        end for;
      end for;
    end for;

  end for;
end configuration fir_taps_con;
```

C.4 Fourth Level of the Hierarchy

integrator.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    11:11:48 07/30/2007
-- Design Name:
-- Module Name:    integrator - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a structural description of a single integrator
--      stage. The structure of this entity is such that there is only one adder between each
--      accumulator register of the cascade.  This greatly increases the design
--      performance.  Below are descriptions of the generics, ports, and internal signals
--      of this block.
--
```

```
--  Generics
--      B_int : # of bits on input and output of integrator stage.
--
--  Ports
--      clk : Global clock signal.
--      rst_h : Global synchronous, asserted high reset signal.
--      int_in : Input to integrator stage.
--      int_out : Output to integrator stage.
--
-- Internal Signals
--      reg_out : This is the output of the accumulator register. This signal
--            is the output of the integrator, but it also feeds back into
--            the second input of the adder.
--      add_out : This is the output of the adder in the integrator.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity integrator is
  generic   ( B_int : integer := 52);
  port      ( clk : in STD_LOGIC;
                 rst_h : in STD_LOGIC;
                 int_in : in STD_LOGIC_VECTOR (B_int - 1 downto 0);
                 int_out : out STD_LOGIC_VECTOR (B_int - 1 downto 0));
end integrator;

architecture struc of integrator is

  -- component declarations
  component gen_reg
    generic ( B : integer);
    port    ( clk : in STD_LOGIC;
                 rst_h : in STD_LOGIC;
                 inp : in STD_LOGIC_VECTOR (B - 1 downto 0);
                 outp : out STD_LOGIC_VECTOR (B - 1 downto 0));
  end component gen_reg;

  component gen_adder
    generic ( B_adder : integer);
    port    ( add1 : in STD_LOGIC_VECTOR (B_adder - 1 downto 0);
                 add2 : in STD_LOGIC_VECTOR (B_adder - 1 downto 0);
```

```
                  sum : out STD_LOGIC_VECTOR (B_adder - 1 downto 0));
  end component gen_adder;

  -- internal signals
  signal reg_out : STD_LOGIC_VECTOR(B_int - 1 downto 0);
  signal add_out : STD_LOGIC_VECTOR(B_int - 1 downto 0);

begin   -- arch. struc of integrator

  -- instantiate adder to perform accumulation
  adder : gen_adder
    generic map (B_adder => B_int)
    port map (add1 => int_in , add2 => reg_out , sum => add_out);

  -- instantiate accumulator register
  acc_reg : gen_reg
    generic map (B => B_int)
    port map (clk => clk , rst_h => rst_h , inp => add_out , outp => reg_out);

  -- assign output of block to internal signal reg_out
  int_out <= reg_out;

end struc;

-- configuration declaration of integrator entity
configuration integrator_con of integrator is
  for struc
    for adder : gen_adder
      use entity work.gen_adder(behv);
    end for;

    for acc_reg : gen_reg
      use entity work.gen_reg(behv);
    end for;
  end for;
end configuration integrator_con;
```

comb.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    16:13:49 08/02/2007
-- Design Name:
-- Module Name:    comb - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a structural description of a single comb stage
--       of the comb cascade of the CIC filter. This entity is pipelined with output
--      registers to reduce the combinational path of the cascade.  With the pipeline
--      registers, there is only one subtractor in between registered nodes.  This comb stage
--      has a fixed delay by two shift register to implement the fixed differential delay
--      value of two.  Below are descriptions of the generics, ports, and internal signals
--      that are used in this entity.
--
--  Generics
--      B_comb : # of bits on input and output of comb stages.
--
--  Ports
--      clk : Global clock signal.
--      rst_h : Global synchronous, asserted high reset signal.
--      ce : Clock enable signal, all registers in comb cascade are driven by this
--              because they have to operate only at 1/D_factor of the input rate.
--      comb_in : Input to comb stage.
--      comb_out : Output of comb stage.
--
--  Internal Signals
--      delayout : This signal is the output of the delayby2 shift register. This
--             is the signal that is subtracted from the input to the comb stage.
--      subout : This is the output signal of the subtractor.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity comb is
  generic   ( B_comb : integer := 52);
  port      ( clk : in STD_LOGIC;
                 rst_h : in STD_LOGIC;
                 ce : in STD_LOGIC;
                 comb_in : in STD_LOGIC_VECTOR (B_comb - 1 downto 0);
                 comb_out : out STD_LOGIC_VECTOR (B_comb - 1 downto 0));
end comb;

architecture struc of comb is

  -- component declarations
  component delayby2
    generic ( B_delayby2 : integer);
    port    ( clk : in STD_LOGIC;
                 rst_h : in STD_LOGIC;
                 ce : in STD_LOGIC;
                 del_in : in STD_LOGIC_VECTOR (B_delayby2 - 1 downto 0);
                 del_out : out STD_LOGIC_VECTOR (B_delayby2 - 1 downto 0));
  end component delayby2;

  component gen_subtractor
    generic ( B_subtractor : integer);
    port    ( sub1 : in STD_LOGIC_VECTOR (B_subtractor - 1 downto 0);
              sub2 : in STD_LOGIC_VECTOR (B_subtractor - 1 downto 0);
              diff : out STD_LOGIC_VECTOR (B_subtractor - 1 downto 0));
  end component gen_subtractor;

  component ce_reg
    generic ( B_ce_reg : integer := 52);
    port    ( clk : in STD_LOGIC;
              rst_h : in STD_LOGIC;
              ce : in STD_LOGIC;
              inp : in STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0);
              outp : out STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0));
  end component ce_reg;

  -- internal signals
  signal delayout : STD_LOGIC_VECTOR(B_comb - 1 downto 0);  -- output of delay block
  signal subout : STD_LOGIC_VECTOR(B_comb - 1 downto 0);    -- output of subtractor block

begin   -- arch. struc of comb

  -- instantiate a block to delay input signal by two samples
  delay : delayby2
    generic map (B_delayby2 => B_comb)
    port map (clk => clk, rst_h => rst_h, ce => ce, del_in => comb_in, del_out => delayout);

  subtractor : gen_subtractor
    generic map (B_subtractor => B_comb)
    port map (sub1 => comb_in, sub2 => delayout, diff => subout);

  -- register the output to reduce the combinational path
  outreg : ce_reg
    generic map (B_ce_reg => B_comb)
    port map (clk => clk, rst_h => rst_h, ce => ce, inp => subout, outp => comb_out);

end struc;

-- configuration declaration for comb entity
configuration comb_con of comb is
  for struc

    for delay : delayby2
      use configuration work.delayby2_con;
    end for;

    for subtractor : gen_subtractor
      use entity work.gen_subtractor(behv);
    end for;
    for outreg : ce_reg
      use entity work.ce_reg(behv);
    end for;

  end for;
end configuration comb_con;
```

C.5 Fifth Level of the Hierarchy

delayby2.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    16:06:20 08/02/2007
-- Design Name:
-- Module Name:    delayby2 - struc
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This entity is simply a cascade of two clock enabled registers
--      This entity is used to delay the input of each comb stage by two samples
--      to implement the fixed differential delay of 2 that this design uses.
--      Below are descriptions of the generics, ports, and internal signals that
--      are used in this entity.
--
--  Generics
--      B_delayby2 : # of bits in each register of this entity.
--
--  Ports
--      clk : A global clock signal.
--      rst_h : A global syncronous, asserted high reset.
--      ce : The clock enable input that is used to drive the registers.
--      del_in : The input sample to the block.
--      del_out : The delayed input sample.
--
--  Internal Signals
--      delayby1 : This signal is the output of the first register, input to the
--             second register.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity delayby2 is
  generic   ( B_delayby2 : integer := 52);
  port      ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                ce : in STD_LOGIC;
                del_in : in STD_LOGIC_VECTOR (B_delayby2 - 1 downto 0);
                del_out : out STD_LOGIC_VECTOR (B_delayby2 - 1 downto 0));
end delayby2;

architecture struc of delayby2 is

  -- component declarations
  component ce_reg
    generic ( B_ce_reg : integer);
    port    ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                ce : in STD_LOGIC;
                inp : in STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0);
                outp : out STD_LOGIC_VECTOR   (B_ce_reg - 1 downto 0));
  end component ce_reg;

  -- internal signals
  signal delayby1 : STD_LOGIC_VECTOR(B_delayby2 - 1 downto 0);

begin

  -- create a register to delay sample by one cycle
  delayreg1 : ce_reg
    generic map (B_ce_reg => B_delayby2)
    port map (clk => clk, rst_h => rst_h, ce => ce, inp => del_in, outp => delayby1);

  -- cascade a second register to delay sample by two cycles
  delayreg2 : ce_reg
    generic map (B_ce_reg => B_delayby2)
    port map (clk => clk, rst_h => rst_h, ce => ce, inp => delayby1, outp => del_out);

end struc;

-- configuration declaration for delayby2 entity
configuration delayby2_con of delayby2 is
  for struc
    for all : ce_reg
      use entity work.ce_reg(behv);
    end for;
  end for;
end configuration delayby2_con;
```

C.6 Leaf Nodes

ce_ctrl.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    18:07:41 08/07/2007
-- Design Name:
-- Module Name:    ce_ctrl - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is used to generate the clock enable signal
--      that drives flip flops operating at the polyphase FIR filter output rate.
--      This entity uses the combined decimation factor (D_cic * D_fir) and the
--      global clock instead of trying to count the clock enable signal that is
--      generated by the downsample entity of the CIC filter. This entity has an
--      enable input to account for the latency due to the pipelined structure of
--      the comb cascade of the CIC filter. This entity does not start counting
--      clock cycles until the enable signal is asserted.
--
--  Generics
--      D_cic : The decimation factor of the CIC filter.
--      D_fir : The decimation factor of the polyphase FIR filter.
--
--  Ports
--      clk : The global clock signal
--      rst_h : The global, synchronous, asserted high reset signal
--      en_in : The asserted high enable signal that tells the entity when to
--          start counting clock cycles.
--      ce_out : The clock enable signal that is used to drive flip flops that
--           operate at the polyphase FIR filter output rate.
--
--  Internal Signals
--      count : The counter signal that is used to count global clock cycles. When
--          this signal gets reset to zero, the ce_out output gets asserted.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ce_ctrl is
  generic   ( D_cic : integer := 10;
                D_fir : integer := 8);
  port      ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                en_in : in STD_LOGIC;
                ce_out : out STD_LOGIC);
end ce_ctrl;

architecture behv of ce_ctrl is
  -- constant value determining combined decimation factor
  constant D_factor : integer := (D_cic * D_fir);
  -- internal signal used to count global clock cycles
  signal count : integer range 0 to D_factor;
begin
  ce_count : process(clk)
  begin
    if clk'event and clk = '1' then -- do everything on posedge of clock
      if rst_h = '1' then   -- create a synchronous reset
        count <= 0;
        ce_out <= '0';
      elsif en_in = '1' then    -- start counting when en_in gets asserted
        if count <= (D_factor - 1) then  -- keep counting until count = D_factor-1
          if count = 0 then -- when count wraps to zero, assert ce_out
            ce_out <= '1';
            count <= count + 1;
          else  -- keep counting until count = D_factor-1
            ce_out <= '0';
            count <= count + 1;
          end if;   -- count = 0
        else
          count <= 0;
          ce_out <= '0';
        end if; -- count <= (D_factor-1)
      end if;   -- en_in = '1'
    end if; -- clk'event and clk = '1'
  end process ce_count;
end behv;
```

addr_ctrl.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    16:35:43 08/16/2007
-- Design Name:
-- Module Name:    addr_ctrl - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity generates the address used to fetch coefficient
--      values out of the tap ROMs in the fir_taps entity. This generated address
--      goes to all of the tap ROMs. As you can see, this entity is simply a down
--      counter that gets set to the maximum value when it is reset. It is
--      important to note that this entity is driven by an enable signal as well
--      as a clock enable signal. This entity is enabled for the same reason that
--      the ce_ctrl entity is enabled, to account for the latency in the combs of
--      the CIC filter. This entity is also drive by a clock enable because it needs
--      to operate at the CIC filter output rate.
--
--  Generics
--      B_addr : The number of bits in the address of the tap ROMs.
--
--  Ports
--      clk : The global clock signal.
--      rst_h : The global, synchronous, asserted high reset signal.
--      en_in : An enable signal telling the counter when to decrement. The counter
--                  decrements when this input and the ce_in input are both high.
--      ce_in : The clock enable signal that makes the counter decrement at the
--          CIC filter output rate.
--      adddr_out : The generated address that is used to fetch data from the
--              tap ROMs in the fir_taps entity.
--
--  Internal Signals
--      count : A counter signal that decrements when this entity is enabled and
--          at the CIC filter output rate. This internal signal is actually the
--          addr_out output.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity addr_ctrl is
  generic   ( B_addr : integer := 3);
  port      ( clk : in STD_LOGIC;
                  rst_h : in STD_LOGIC;
                  en_in : in STD_LOGIC;
                  ce_in : in STD_LOGIC;
                  addr_out : out STD_LOGIC_VECTOR (B_addr - 1 downto 0));
end addr_ctrl;

architecture behv of addr_ctrl is
  -- declare a counter signal used for address output
  signal count : STD_LOGIC_VECTOR (B_addr - 1 downto 0);
begin
  counter : process(clk)
  begin
    if clk'event and clk = '1' then -- do everything on posedge of clock
      if rst_h = '1' then   -- make a synchronous reset
        count <= (others => '1');    -- set address to max. value on a reset
      elsif en_in = '1' and ce_in = '1' then    -- decrement when en_in and ce_in are '1'
        count <= count - 1;
      end if;   -- en_in = '1' and ce_in = '1'
      addr_out <= count; -- assign count to output
    end if; -- clk'event and clk = '1'
  end process counter;
end behv;
```

en_ctrl.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    13:43:31 09/01/2007
-- Design Name:
-- Module Name:    en_ctrl - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is used to account for the latency of the
--      comb cascade in the CIC filter. It simply counts the number of clock enable
--      signals that are generated by the downsample entity in the CIC filter and
--      outputs an enable signal based on the number of comb stages that exist. This
--      enable signal is then used to tell the ce_ctrl entity when to start counting
--      cycles of the global clock. Below are descriptions of the generics, ports,
--      and internal signals that are used in this entity.
--
--  Generics
--      N_cic_stage : The number of stages in the CIC filter that precedes this
--                  polyphase FIR filter.
--
--  Ports
--      clk : The global clock signal.
--      rst_h : A global, synchronous, asserted high reset signal.
--      ce_in : The clock enable signal generated in the downsample block of the
--          CIC filter. This is what gets counted.
--      en_out : The enable output that tells the ce_ctrl module when to start
--                  counting global clock cycles.
--
--  Internal Signals
--      count : This signal is used to hold the count of the ce_in input. The en_out
--              output is asserted based on this signal.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity en_ctrl is
  generic   ( N_cic_stages : integer := 5);
  port      ( clk : in STD_LOGIC;
                rst_h : in STD_LOGIC;
                ce_in : in STD_LOGIC;
                en_out : out STD_LOGIC);
end en_ctrl;

architecture behv of en_ctrl is
  -- declare a signal to count ce_in pulses of the appropriate size
  signal count : integer range 0 to N_cic_stages;
begin

  cic_out_rdy : process(clk)
  begin
    if clk'event and clk = '1' then -- do everything on a posedge of clk
      if rst_h = '1' then   -- make a synchronous, asserted high reset
        count <= 0;  -- reset the counter
        en_out <= '0';   -- disable the ce_ctrl entity
      elsif ce_in = '1' then    -- count ce_in pulses
        if count <= (N_cic_stages - 1) then  -- count N_cic_stages-1 pulses
          count <= count + 1;
          en_out <= '0';
        else    -- stop counting and assert en_out when N_cic_stages-1 pulses are counted
          count <= count;
          en_out <= '1';
        end if; -- count <= (N_cic_stages-1)
      end if;   -- rst_h = '1'
    end if; -- clk'event and clk = '1'
  end process cic_out_rdy;
end behv;
```

signextend.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    10:16:16 07/30/2007
-- Design Name:
-- Module Name:    signextend - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a simple sign extender.  It is used to sign
--      extend the input of the CIC to the accumulator bit width of the first
--      integrator stage.  To do this, it uses the "for generate" form of the generate
--      loop.  Below are descriptions of the generics, ports, and internal signals that
--      are used in this module.
--
--  Generics
--      Bin : # of bits on the input port of module.
--      Bout : # of bits on the output port of module.
--
--  Ports
--      input : Non-sign extended value from ADC.
--      output : Sign extended value from ADC.
--
--  Internal Signals
--      This module has no internal signals.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity signextend is
  generic   ( Bin : integer := 13;
                  Bout : integer := 52);
  port      ( input : in STD_LOGIC_VECTOR (Bin - 1 downto 0);
                  output : out STD_LOGIC_VECTOR (Bout - 1 downto 0));
end signextend;
```

```
architecture behv of signextend is
begin
  output(Bin − 1 downto 0) ≤ input;
  sgn_ext : for j in Bin to Bout − 1 generate
    output(j) ≤ input(Bin − 1);
  end generate sgn_ext;
end behv;
```

clk_ctrl.vhd

```
————————————————————————————————————————————————————————————————————————————————————
−− Company:
−− Engineer:
−−
−− Create Date:    11:48:33 08/02/2007
−− Design Name:
−− Module Name:    clk_ctrl − behv
−− Project Name:
−− Target Devices:
−− Tool versions:
−− Description: This VHDL entity is a leaf node of the downsample block. This entity
−−      creates a clock enable signal based on the incoming, global clock signal.
−−      This clock enable signal is then used to drive clock enabled registers later
−−      on in the processing chain. Clk_ctrl counts a certain number of incoming, fast
−−      clock cycles and outputs this clock enable signal based on the decimation factor.
−−      Below are descriptions of the generics, ports, and internal signals used in
−−      this module.
−−
−−  Generics
−−      D : The decimation factor of the CIC filter.
−−
−−  Ports
−−      clk : Global clock signal.
−−      rst_h : Global, synchronous, asserted high reset signal.
−−      en_in : Input enable signal. The state machine does not start counting cycles
−−              until this signal is asserted. This input comes from the start_ctrl
−−          module in this downsample entity.
−−      en_out : Clock enable signal that is used to drive all clock enabled registers
−−           following this block in the processing chain.  This output is a
−−           registered output.
−−
−−  Internal Signals
−−      count : This signal is used to count the number of the global clock signal
−−              and assert the output at appropriate times.
−−
−− Dependencies:
−−
−− Revision:
−− Revision 0.01 − File Created
−− Additional Comments:
−−
```

```
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity clk_ctrl is
  generic   ( D : integer := 125);
  port  ( clk  : in STD_LOGIC;
          rst_h  : in STD_LOGIC;
          en_in  : in STD_LOGIC;
          en_out  : out STD_LOGIC);
end clk_ctrl;

architecture behv of clk_ctrl is
  signal count : integer range 0 to D;  -- create an integer signal big enough for given D
begin
  ctrl : process(clk)
  begin
    if clk'event and clk = '1' then -- count positive edges of input clock signal
      if rst_h = '1' then       -- generate a syncronous, asserted high reset
        count <= 0;
        en_out <= '0';
      elsif en_in = '1' then        -- don't start counting unless FSM is enabled
        if count <= (D - 2) then -- count up to D - 2 because output is registered
          if count = 0 then     -- if counter has wrapped to zero, keep next sample
            en_out <= '1';
            count <= count + 1;
          else                -- if counter is less than D - 2, keep counting
            en_out <= '0';
            count <= count + 1;
          end if; -- count = 0
        else                  -- reset counter when it reaches D - 1
          count <= 0;
          en_out <= '0';
        end if; -- count <= (D - 2)
      end if;   -- en_in = '1'
    end if; -- clk'event and clk = '1'
  end process ctrl;
end behv;
```

start_ctrl.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
```

```
--
-- Create Date:    12:00:00 08/02/2007
-- Design Name:
-- Module Name:    start_ctrl - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a leaf node of the downsample block. This job creates
--      an enable signal for the counter/state machine clk_ctrl.  Since there are
--      multiple integrator stages, the first good sample from the point of view of the
--      downsample block is not the first sample input after a reset.  It takes N cycles
--      of the incoming clock for the first good sample to be output from the integrator
--      stages.  This block counts N cycles of the incoming clock and then enables the clk_ctrl
--      counter/state machine telling it that the sample coming is the first good sample to keep.
--      Below are descriptions of the generics, ports, and internal signals used in this entity.
--
--  Generics
--      N : # of stages in integrator cascade.  This is the number of cycles that need
--          to be counted until the first good sample appears at the input of the
--          downsample block.
--
--  Ports
--      clk : A global clock signal.
--      rst_h : A global synchronous, asserted high reset signal.
--      en : Output coming from this block telling which sample for clk_ctrl to keep.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 1 - November 4, 2007 - changed how long the counter waits until it
--      creates the enable signal.  In other words, line 68 was changed from ...≤ N-3
--      to ...≤ N - 2.  This change was necessary because I added the demodulator
--      to the design.  The demodulator added an extra cycle of latency that needed to
--      be taken into account.
--
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity start_ctrl is
  generic    ( N : integer := 5);
  port       ( clk : in  STD_LOGIC;
                rst_h : in  STD_LOGIC;
                en : out  STD_LOGIC);
end start_ctrl;
```

```
architecture behv of start_ctrl is
  signal count : integer range 0 to N;  -- create an integer signal of correct size
begin

  startup_count : process(clk)
  begin
    if clk'event and clk = '1' then -- count positive edges of incoming clock
      if rst_h = '1' then       -- create an asserted high, synchronous reset
        count ≤ 0;
        en ≤ '0';
      else
        if count ≤ N − 2 then   -- if count is ≤ N − 2, keep counting
          count ≤ count + 1;
          en ≤ '0';
        else            -- as soon as count reaches N − 1, stop counting and assert en
          en ≤ '1';
        end if; -- count ≤ N − 3
      end if;   -- rst_h = '1'
    end if; -- clk'event and clk = '1'
  end process startup_count;

end behv;
```

ce_reg.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    11:42:25 08/02/2007
-- Design Name:
-- Module Name:    ce_reg − behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a general bit width, positive edge triggered,
--      clock enabled D−type register.  This is used for all registers after the
--      downsample block.  Below are descriptions of the ports and generics used in
--      this entity.
--
--  Generics
--      B_ce_reg : # of bits in the register.
--
--  Ports
--      clk : A global clock signal.
--      rst_h : A global synchronous, asserted high reset signal.
--      ce : A clock enable signal.  This signal is created from the downsample block
--            and used to drive any register that needs to operate at the divided rate.
--      inp : The input to this register.
--      outp : The output of this register.
```

```
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ce_reg is
  generic   ( B_ce_reg : integer := 52);
  port      ( clk : in STD_LOGIC;
                  rst_h : in STD_LOGIC;
                  ce : in STD_LOGIC;
                  inp : in STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0);
                  outp : out STD_LOGIC_VECTOR (B_ce_reg - 1 downto 0));
end ce_reg;

architecture behv of ce_reg is
begin
  en_reg : process (clk)
  begin
    if clk'event and clk = '1' then -- create a positive edge triggered register
      if rst_h = '1' then            -- create a synchronous, asserted high resettable register
        outp <= (others => '0');
      elsif ce = '1' then            -- create a clock enabled register
        outp <= inp;
      end if;   -- rst_h = '1'
    end if; -- clk'event and clk = '1'
  end process en_reg;
end behv;
```

coeffrom.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    16:33:41 08/13/2007
-- Design Name:
-- Module Name:    coeffrom - behv
-- Project Name:
```

```
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a ROM module that is initialized with the
--      contents of a file. This file contains what each element should be
--      initialized to in bit vector format. These files are created by the
--      MATLAB script polyfilt.m. The entity uses an impure function to initialize
--      the rom array to the values contained in the file. Below are descriptions
--      of the generics, ports, and internal signals used in this entity.
--
--  Generics
--      N_elems : The number of elements in the ROM.
--      B_data : The bit width of the elements stored in the ROM.
--      B_addr : The bit width of the address input to the ROM.
--      fname : The file name that is used to initialize the ROM. This file must
--                be in the same directory as the entity.
--
--  Ports
--      clk : The global clock signal.
--      rst_h : The global synchronous, asserted high reset signal.
--      en : The enable signal that is used to output new data at the CIC filter
--           output rate.
--      addr : The address input to the ROM.
--  rom_out : The output data from the ROM.
--
--  Internal Signals
--      rom : This is a constant signal that is an (N_elems) by (B_data) arra
--            that stores the coefficients of the filter.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use STD.TEXTIO.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity coeffrom is
  generic   ( N_elems : integer := 8;
                  B_data : integer := 14;
                  B_addr : integer := 3;
                  fname : string := "rom0.txt");
  port      ( clk : in STD_LOGIC;
                  rst_h : in STD_LOGIC;
                  en : in STD_LOGIC;
                  addr : in STD_LOGIC_VECTOR (B_addr - 1 downto 0);
                  rom_out : out STD_LOGIC_VECTOR (B_data - 1 downto 0));
end coeffrom;

architecture behv of coeffrom is
  -- create an internal rom array of bit_vector type so it can be initialized with a file
  type rom_type is array (0 to N_elems - 1) of bit_vector (B_data - 1 downto 0);
  -- create an impure function to load the ram with a file
  impure function load_rom (load_file : in string) return rom_type is
    file rom_file : text open read_mode is load_file;   -- open a file in read mode
    variable line_in : line;        -- declare a line variable for reading from file
    variable tmp_rom : rom_type;    -- declare a rom array to initialize and then return
  begin
    for I in rom_type'range loop
      readline(rom_file, line_in);  -- read from file to line variable
      read(line_in, tmp_rom(I));        -- initialize rom element to value from file
    end loop;
    return tmp_rom;
  end function load_rom;
  -- create a constant rom array and initialize it with contents of file fname
  constant rom : rom_type := load_rom(fname);
begin   -- arch. behv of coeffrom
  read_rom : process(clk)
  begin
    if clk'event and clk = '1' then -- read on a posedge of clock
      if rst_h = '1' then   -- output last value of ROM on a reset
        rom_out <= to_stdlogicvector(rom(N_elems - 1));
      elsif en = '1' then   -- output data according to addr when en is asserted
        rom_out <= to_stdlogicvector(rom(conv_integer(addr)));
      end if;   -- rst_h = '1'
    end if; -- clk'event and clk = '1'
  end process read_rom;
end behv;
```

gen_mult.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    17:01:47 08/07/2007
-- Design Name:
-- Module Name:    gen_mult - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a generic, signed, full precision multiplier.
--      Below are descriptions of the generics, ports, and internal signals of this
--      entity.
--
--  Generics
--       B_mult_in1 : The bit width of the first input to the multiplier. In this
--                design, this is the input to the polyphase FIR filter.
--       B_mult_in2 : The bit width of the second input to the multiplier. In this
--                design, this is the output of the tap ROMs.
--
--   Ports
--       in1 : The first input to the multiplier.
--       in2 : The second input to the multiplier.
--       prod : The full precision, signed output of the multiplier.
--
--   Internal Signals
--       There are no internal signals in this entity.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity gen_mult is
  generic   ( B_mult_in1 : integer := 14;
                  B_mult_in2 : integer := 16);
  port          ( in1 : in STD_LOGIC_VECTOR (B_mult_in1 - 1 downto 0);
                  in2 : in STD_LOGIC_VECTOR (B_mult_in2 - 1 downto 0);
                  prod : out STD_LOGIC_VECTOR ((B_mult_in1 + B_mult_in2) - 1 downto 0));
end gen_mult;

architecture behv of gen_mult is
begin
  mult : process (in1, in2)
  begin
    prod <= signed(in1) * signed(in2);
  end process mult;
end behv;
```

accum_D.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
```

```
--
-- Create Date:    14:48:26 08/07/2007
-- Design Name:
-- Module Name:    accum_D - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity accumulates the input signal for D_factor
--      cycles. This entity uses signed arithmetic to do this. Also, this entity has
--      a clock enable signal because it needs to accumulate at the CIC filter output
--      rate, not the global clock rate. This entity keeps full precision in the
--      accumulator, in other words, there are several guard bits in the accumulator
--      to prevent overflow. Below are descriptions of the generics, ports, and
--      internal signals used in this entity.
--
--  Generics
--      D_factor : The decimation factor of the polyphase FIR filter. This is
--             the number of cycles that the input needs to be accumulated.
--      B_acc_in : The bit width of the input to the accumulator.
--      B_acc_out : The bit width of the output of the accumulator. This is also
--              the bit width of the accumulator. This is usually larger than
--              the input by several bits so that there are some guard bits
--              in the accumulator to prevent overflow.
--
--  Ports
--      clk : The global clock signal.
--      rst_h : The global, synchronous, asserted high reset signal.
--      ce_in : The clock enable signal used to operate the accumulator at the
--          CIC filter output rate instead of the faster global clock rate.
--      acc_in : The input to the accumulator. In this design, this is the output
--           of the tap multiplier.
--      acc_out : The output of the accumulator. This is output at the polyphase
--            FIR filter output rate.
--
--  Internal Signals
--      accumulator : This signal is used to keep the value of the accumulated
--                input.
--      tmp : This signal is a temporary register that is used to hold the value
--            of the input when the counter overflows. This is necessary because
--            without it, the input that occurs on the cycle that the counter
--            overflows does not get accumulated. This register stores the value
--            of the input that occurs on the cycle that the counter overflows.
--            This value is then added to the accumulator on the next cycle.
--      count : This signal is a counter that is used to keep track of how many
--          cycles the input has been accumulated.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity accum_D is
  generic   ( D_factor : integer := 8;
                 B_acc_in : integer := 18;
                 B_acc_out : integer := 24);
  port      ( clk  : in STD_LOGIC;
                 rst_h  : in STD_LOGIC;
                 ce_in  : in STD_LOGIC;
                 acc_in  : in STD_LOGIC_VECTOR (B_acc_in - 1 downto 0);
                 acc_out  : out STD_LOGIC_VECTOR (B_acc_out - 1 downto 0));
end accum_D;

architecture behv of accum_D is
  -- function declaration for a sign extender. This is needed to
  -- sign extend the tmp register output to the accumulator bit
  -- bit width when it is added to the accumulator.
  function signextend ( num_short : in STD_LOGIC_VECTOR (B_acc_in - 1 downto 0))
    return STD_LOGIC_VECTOR is
    variable tmp_num : STD_LOGIC_VECTOR (B_acc_out - 1 downto 0);
  begin
    tmp_num(B_acc_in - 1 downto 0) := num_short;
    sign_ext : for i in B_acc_in to B_acc_out - 1 loop
      tmp_num(i) := num_short(B_acc_in - 1);
    end loop sign_ext;
    return tmp_num;
  end function signextend;
  -- internal signals
  signal accumulator : STD_LOGIC_VECTOR (B_acc_out - 1 downto 0);
  signal tmp : STD_LOGIC_VECTOR (B_acc_in - 1 downto 0);
  signal count : integer range 0 to D_factor;
begin
  accum : process(clk)
  begin
    if clk'event and clk = '1' then -- do everything on posedge of clock
      if rst_h = '1' then   -- create a synchronous, asserted high reset
        accumulator <= (others => '0');
        count <= 0;
        acc_out <= (others => '0');
        tmp <= (others => '0');
      elsif ce_in = '1' then    -- count when ce_in is asserted
        if count <= (D_factor - 2) then  -- count for D_factor-2 cycles
          count <= count + 1;     -- increment counter
          if count = 0 then -- when counter overflows, add tmp to the input
            accumulator <= signed(signextend(tmp)) + signed(signextend(acc_in));
          else  -- when counter hasn't overflowed, add the input to the accumulator
            accumulator <= signed(accumulator) + signed(signextend(acc_in));
          end if;   -- count = 0
        else    -- reset counter when count = D_factor-1 and output accumulator
```

```
          count <= 0;
          acc_out <= accumulator;
          accumulator <= (others => '0');
          tmp <= acc_in;
        end if; -- count <= (D_factor-2)
      end if;   -- rst_h = '1'
    end if; -- clk'event and clk = '1'
  end process accum;
end behv;
```

gen_adder.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    13:20:48 08/06/2007
-- Design Name:
-- Module Name:    gen_adder - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a generic signed adder.  This is
--      a leaf node of the integrator entity.  This adder is used in the
--              accumulators of the integrator stages.  It is important to note that this
--              adder has no carry in or carry out ports.  For this design, it is assumed
--      that the accumulator bit width is sufficient to not include these ports.
--              Below are descriptions of the generics, ports, and internal signals that
--      are used in this entity.
--
--  Generics
--      B_adder : # of bits on input and output of generic adder.
--
--  Ports
--      add1 : The first input to be added together.
--      add2 : The second input to be added together.
--      sum : The result of adding add1 and add2.
--
--  Internal Signals
--  This module has no internal signals.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity gen_adder is
  generic   ( B_adder : integer := 52);
  port      ( add1 : in STD_LOGIC_VECTOR (B_adder - 1 downto 0);
                add2 : in STD_LOGIC_VECTOR (B_adder - 1 downto 0);
                sum : out STD_LOGIC_VECTOR (B_adder - 1 downto 0));
end gen_adder;

architecture behv of gen_adder is
begin

  add : process(add1, add2)
  begin
    sum <= signed(add1) + signed(add2);
  end process add;

end behv;
```

gen_subtractor.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    13:23:00 08/06/2007
-- Design Name:
-- Module Name:    gen_subtractor - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a generic B_subtractor - bit signed subtractor.
--       This is a leaf node in the comb structure of the CIC filter.  This entity
--       resides in the fourth level of hierarchy in the design.  Gen_subtractor
--       subtracts sub2 from sub1 to create diff.  Below are descriptions of the
--       generics and ports used in this module.
--
--   Generics
--       B_subtractor : # of bits on inputs and output of module
--
--   Ports
--       sub1 : Number that sub2 gets subtracted from.
--       sub2 : Number that is subtracted from sub1.
--       diff : The difference of sub1 and sub2.
--
-- Dependencies:
```

```
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity gen_subtractor is
    generic ( B_subtractor : integer := 52);
    port       ( sub1 : in STD_LOGIC_VECTOR (B_subtractor - 1 downto 0);
                 sub2 : in STD_LOGIC_VECTOR (B_subtractor - 1 downto 0);
                 diff : out STD_LOGIC_VECTOR (B_subtractor - 1 downto 0));
end gen_subtractor;

architecture behv of gen_subtractor is
begin

    sub : process(sub1, sub2)
    begin
        diff <= signed(sub1) - signed(sub2);
    end process sub;

end behv;
```

gen_reg.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    17:44:10 07/03/2007
-- Design Name:
-- Module Name:    gen_reg - behv
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: This VHDL entity is a B-bit positive edge triggered register with an
--  asynchronous, asserted high reset signal.  This is used for the accumulator
--     registers in the integrator cascade of the CIC filter.  Below are descriptions
--     of the generics, ports, and internal signals that are used in this entity.
--
--  Generics
--      B : # of bits in register.
--
--  Ports
--      clk  : A global clock.
--      rst_h  : A global, synchronous, asserted high reset signal
--      inp : Input to the register.
--      outp : Output of the register.
--
-- Internal Signals
--      There are no internal signals in this entity.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
--              0.02 - changed from asynchronous reset to synchronous reset, still
--                       asserted high.
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity gen_reg is
    generic ( B : integer := 52);
    port        ( clk  : in STD_LOGIC;
                  rst_h  : in STD_LOGIC;
                  inp : in STD_LOGIC_VECTOR(B - 1 downto 0);
                  outp : out STD_LOGIC_VECTOR(B - 1 downto 0));
end gen_reg;

architecture behv of gen_reg is
begin
    reg : process(clk)
    begin
        if clk'event and clk = '1' then -- create a positive edge triggered register
            if rst_h = '1' then             -- create a synchronous, asserted high reset
                outp <= (others => '0');
            else
                outp <= inp;                 -- copy input to output on a positive edge
            end if; -- rst_h = '1'
        end if; -- clk'event and clk = '1'
    end process reg;
end behv;
```