5-1-1995

ASIC design of an IIR digital filter: Using Mentor Graphics DSP Station Tools

Robert Panek

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritsscholarworks@rit.edu.
ASIC Design of an IIR Digital Filter

Using Mentor Graphics DSP Station Tools

by

Robert C. Panek

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by: _____________________________________________
Graduate Advisor - Kenneth W. Hsu, Associate Professor

________________________________________
George A. Brown, Professor

___________________________
Roy S. Czernikowski, Professor and Department Head

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York

May 1995
Title: ASIC Design of an IIR Digital Filter Using Mentor Graphics DSP Station Tools

I, Robert C. Panek, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Signature: ________________________________

Date: June 7, 1995
Abstract

Automation in VLSI design is a powerful way to simplify the VLSI layout process and will allow for faster time to market for integrated circuit designs. One means of automation is VHDL, a hardware description language for integrated circuit designs. A structured VHDL description can be used to describe the hardware design at the logic-gate level, and automated software is available that will use this gate-level design to generate the VLSI layout. A more recent type of automation occurs at a level above this. The Mentor Graphics DSP Station tools use a high-level algorithmic description to generate the gate-level VHDL description. These tools are especially intended for applications in digital signal processing (DSP), providing simulation tools particularly geared toward DSP algorithms. One application of digital signal processing is an infinite impulse response (IIR) filter. With the use of the Mentor Graphics tools, a digital filter was designed from a set of original specifications down to the silicon level. N-well 1.2 micron CMOS technology with two metal layers and one polysilicon layer was used to implement the filter layout. Using the 1.2 micron CMOSN standard cell library, the final VLSI layout measured 7.315 mm x 7.213 mm, containing approximately 25,700 transistors.
Table of Contents

Abstract............................................................................................................. iii

Table of Contents............................................................................................ iv

List of Figures .................................................................................................... vi

List of Tables ...................................................................................................... vii

Glossary .............................................................................................................. viii

Chapter 1: Filter Design Concepts ................................................................. 1
  1.1 Introduction to Digital Filter Design ....................................................... 1
  1.2 Digital Filter Representations ................................................................ 4
    1.2.1 Difference Equation ....................................................................... 5
    1.2.2 Block Diagram ............................................................................... 5
    1.2.3 Transfer Function $H(z)$ ................................................................. 6
    1.2.4 Impulse Response $h(n)$ ................................................................. 7
    1.2.5 Pole/Zero Plot ................................................................................ 9
  1.3 Derivation of the Bilinear Z-Transform ................................................. 10

Chapter 2: IIR Digital Filter Design .............................................................. 15
  2.1 Filter Specifications .................................................................................. 15
    2.1.1 Original Design Specifications ....................................................... 15
    2.1.2 Filter Order Complexity ................................................................. 17
    2.1.3 Prewarping Filter Frequencies to Correct for BZT ....................... 22
  2.2 Designs Based on Normalized Low-Pass Filter ..................................... 26
    2.2.1 Transformations Based on Normalized Low-Pass Filter ............... 27
    2.2.2 Converting $H(s) \rightarrow H(z)$ Using the Bilinear Z-Transform .......... 28
    2.2.3 Biquad-Section Design .................................................................. 29

Chapter 3: Mentor Graphics Automated Design Tools .............................. 32
  3.1 Introduction to Tools .............................................................................. 32
  3.2 FILlab: High-Level Filter Design and Simulation ............................... 32
  3.3 Design Architect: DFL and Schematic Editor ....................................... 34
  3.4 DSPlab: Simulation Tools for DFL ......................................................... 34
  3.5 Mistrall/HDL: VHDL Generation/Compilation Tools ......................... 35
  3.6 AutoLogic: Synthesizing VHDL -> Schematic .................................... 37
  3.7 DVE: Design Viewpoint Checking and Validation .............................. 38
  3.8 QuickSim II: VHDL Simulation/Verification ......................................... 38
Chapter 4: IIR Band-Reject Filter Design Example .............. 40

4.1 Filter Specifications and Applications ........................................ 40
4.2 DFL Design and High-Level Simulation ...................................... 41
  4.2.1 Derivation of Filter Coefficients ....................................... 41
  4.2.2 DFL Description of Filter ............................................... 44
  4.2.3 Final DFL Simulation Results .......................................... 45
4.3 VHDL Generation and Compilation .......................................... 49
4.4 AutoLogic: Synthesizing VHDL -> Schematic ............................... 51
4.5 Design Architect: Refining Schematic .................................... 52
4.6 DVE: Schematic Checks and ERC Validation ................................. 52
4.7 QuickSim II: VHDL Simulation/Verification ................................ 52
4.8 IC Station: Schematic -> VLSI Layout .................................... 59
4.9 Testing the Filter ASIC Chip .................................................. 60

Chapter 5: Conclusions .......................................................... 62

References ................................................................................. 64

Appendix A: MathCad Derivation of Filter Coefficients .......... A-1
Appendix B: Mistral1 Generated VHDL Code ......................... B-1
Appendix C: I/O Schematic For VLSI Chip ............................ C-1
Appendix D: Tutorial .................................................................. D-1
List of Figures

Figure 1.1 Ideal Filter Frequency Response.............................................................. 1
Figure 1.2 Digital Filter Block Diagram ........................................................................ 6
Figure 1.3 Digital Filter Pole/Zero Plot ........................................................................ 10
Figure 1.4 Graph of Function g(α) For BZT Derivation ............................................. 11
Figure 1.5 Integration Approximation for the Function g(α).......................................... 13
Figure 2.1 Low-Pass Filter Prototypes for the Four Filter Types ................................. 20
Figure 2.2 Graphical Representation of BZT Mapping .................................................. 23
Figure 2.3 Graph of Nonlinearity Between s-Domain and z-Domain ............................. 24
Figure 2.4 Frequency Warping in Filter Design ........................................................... 25
Figure 2.5 Block Diagram for a Single Biquad-Section ................................................. 30
Figure 2.6 Simplified Block Diagram for Biquad-Section ............................................. 31
Figure 3.1 Road Map Through Mentor Graphics Tools ............................................... 33
Figure 4.1 Filter Input Test Signal.................................................................................. 46
Figure 4.2 Filter Output Response To Test Signal......................................................... 47
Figure 4.3 Filter Design Example Magnitude Response ............................................... 47
Figure 4.4 Filter Design Example Phase Response....................................................... 48
Figure 4.5 Chip I/O Interfacing Timing Diagram .......................................................... 51
List of Tables

Table 4.1 Filter Design Example Specifications .................................................. 40
Table 4.2 Filter Design Example Simulation Results ........................................... 48
Table 4.3 I/O Pin Description for Filter Design Example IC .................................. 50
Table 4.4 Bit-Serial Inputs For Digital Simulation ............................................. 53
Table 4.5 ms1_in Force Statements For Digital Simulation ................................. 55
Table 4.6 Digital Simulation Results ................................................................ 58
Table 4.7 Chip Output Delays From Silicon to Pin ............................................ 59
## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit (p. 32).</td>
</tr>
<tr>
<td>Biquad-Section</td>
<td>A two-pole/two-zero IIR filter. Two or more can be cascaded together to implement a higher-order filter with less error due to coefficient quantization (p. 29).</td>
</tr>
<tr>
<td>Bit-Parallel Design</td>
<td>Separate hardware is used to process entire word-sizes at once (p. 36).</td>
</tr>
<tr>
<td>Bit-Serial Design</td>
<td>The same hardware is reused when processing the bits of individual words in a design. To accomplish this, the bits of the word must be shifted in one at a time (p. 36).</td>
</tr>
<tr>
<td>Butterworth Filter</td>
<td>A type of analog filter whose spacing between the poles is known to be $360\degree/2n$ for a low-pass filter, where $n$ indicates the number of poles. A Butterworth filter does not have any zeroes in the s-domain (p. 18).</td>
</tr>
<tr>
<td>BZT</td>
<td>Bilinear Z-Transform; used as a transformation between the s-domain and the z-domain (p. 10).</td>
</tr>
<tr>
<td>DFL</td>
<td>Design Flow Logic; the high-level language that is used to describe the digital filter within the DSP Station tools (p. 32).</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing (p. 3).</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DVE</td>
<td>Design Viewpoint Editor (p. 38).</td>
</tr>
<tr>
<td>ERC</td>
<td>Electrical Rules Checking (p. 38).</td>
</tr>
<tr>
<td>Filter Gain</td>
<td>The ratio of the magnitude of the output to the magnitude of the input. The filter gain will vary for different frequencies in a digital filter (p. 2).</td>
</tr>
<tr>
<td>Filter Order</td>
<td>The number of poles in a digital filter. The order is an indication of the complexity of the filter (p. 17).</td>
</tr>
<tr>
<td>FIR Filter</td>
<td>Finite Impulse Response Filter (p. 3).</td>
</tr>
<tr>
<td>IIR Filter</td>
<td>Infinite Impulse Response Filter (p. 3).</td>
</tr>
<tr>
<td>Impulse Response h(n)</td>
<td>The output of a digital filter when the impulse function is used as the input to the filter (p. 7).</td>
</tr>
<tr>
<td>Intermediate Low-Pass Filter</td>
<td>A filter $H_{ai}(s)$ derived from the normalized low-pass filter. This is an intermediate filter while transforming the normalized low-pass filter $H_{ai}(s)$ into a more complicated denormalized parent filter $H_{ap}(s)$. The intermediate filter will have its passband cutoff frequency at $\Omega_c$ (p. 18).</td>
</tr>
<tr>
<td>LSB</td>
<td>Least-Significant Bit (p. 49).</td>
</tr>
<tr>
<td>Magnitude Response</td>
<td>A graph of the filter gain for the valid range of input frequencies (p. 1).</td>
</tr>
</tbody>
</table>
MSB

Most-Significant Bit (p. 53).

Normalized Low-Pass Filter

A simplified low-pass filter $H_{an}(s)$ with a passband cutoff frequency of 1 rad/sec. This can be used as a prototype filter to derive a more complicated denormalized parent filter. In the discussion of Chapter 2, this filter will be converted to the denormalized parent filter $H_{ap}(s)$ in two steps, resulting in an intermediate filter $H_{a}(s)$. The normalized filter is also referred to as a one-radian low-pass filter (p. 18).

Nyquist Frequency

Half the sampling frequency. This frequency represents the maximum frequency that can be distinguished accurately with digital sampling (p. 16).

Parent Analog Filter

The denormalized analog filter $H_{ap}(s)$ that a digital filter will be derived from (p. 3).

Passband

The range of frequencies which are allowed to pass through the filter without being suppressed (p. 2).

Phase Response

A graph of the filter phase shift that occurs between the output and the input for the valid range of input frequencies (p. 4).

Poles of $H(z)$

The values of $z$ for which the transfer function $H(z)$ equals $\infty$. These are the roots of the denominator polynomial of $H(z)$ (p. 4).

s-Domain

The frequency domain, where $s = j\omega$ (p. 10).
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stopband</td>
<td>The range of frequencies which are suppressed and not allowed to appear on the output of the filter (p. 1).</td>
</tr>
<tr>
<td>Transfer Function $H(z)$</td>
<td>The digital z-domain ratio $Y(z)/X(z)$ describing the relationship between the output $Y(z)$ and the input $X(z)$ (p. 6).</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language (p. 32).</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit (p. 32).</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration (p. 32).</td>
</tr>
<tr>
<td>z-Domain</td>
<td>The digital domain, where $z = e^{j\theta}$ (p. 4).</td>
</tr>
<tr>
<td>Zeroes of $H(z)$</td>
<td>The values of $z$ for which the transfer function $H(z)$ equals 0. These are the roots of the numerator polynomial of $H(z)$ (p. 9).</td>
</tr>
</tbody>
</table>
Chapter 1: Filter Design Concepts

1.1 Introduction to Digital Filter Design

There are four main types of filters: the low-pass filter, the high-pass filter, the band-pass filter, and the band-stop filter. An ideal low-pass filter is one that will pass input frequencies less than oct to the output but block the portions of the input signal with higher frequencies from appearing at the output. High-pass filters, on the other hand, block out the lower frequencies and pass the higher ones. A band-pass filter blocks both low and high frequencies and passes only those frequencies within the range of the filter specifications, while the band-stop filter, sometimes called a band-reject filter or a notch filter, blocks only a select range of input frequencies while passing frequencies both below and above the stopband frequencies. The ideal filter magnitude frequency response is shown in Figure 1.1 below.
This indicates that the gain of the filter is equal to one for frequencies that are passed by the filter and zero for the frequencies that are rejected.

Analog filters are useful in filtering the input signal before it is digitized using an analog-to-digital (A/D) converter. The digitized signal is often processed by a microprocessor. If instead, a digital filter can be designed inside the microprocessor, then there is no need for the analog filter. The noisy signal can be sent directly to the A/D converter, and the digitized result can be filtered by the microprocessor using a digital filter design that is analogous to the original analog filter.

The power of digital filters in circuits comes from their flexibility. If tomorrow, it is decided that a band-pass filter would have worked better than a low-pass filter, it is easier to modify the firmware code within the microprocessor than it is to remove hardwired circuit parts from the circuit boards and replace them. This is useful both while developing a product and enhancing it. Perhaps, instead of changing to a band-pass filter, it is better to widen the passband of the original filter. With current technology, many products have remote communications ability built-in. If this is so, then the microprocessor firmware can often be updated remotely without requiring a hardware update.

Another advantage of digital filters is their reliability. The results of a digital filter are always correct for the given input samples. There is no drift due to temperature or humidity. Because the operations are performed within a microprocessor, fewer analog components, which tend to be affected by environmental
changes, are needed. Of course, a smaller number of analog components translates into lower-cost designs also.

Digital filters will not replace analog filters completely. There are cases where it is not practical for the microprocessor to implement the digital filter. For example, if the microprocessor is not powerful enough to both meet the product specifications and implement a digital filter, then it is probably more expensive to use a more-powerful microprocessor than it is to use an analog filter. Also, there are many applications that require a filter but have no microprocessor available to perform the digital signal processing.

Digital filters is a major topic in the field of digital signal processing (DSP). There are two main types of digital filters: infinite impulse response (IIR) filters and finite impulse response (FIR) filters. As the name suggests, the response of an infinite impulse response filter to an impulse input is of infinite duration. Conversely, the response of a finite impulse response filter is zero outside of some finite time interval.

IIR filters can be derived from “parent” analog filters. A transformation can be performed to transform the analog filter into a digital filter equivalent. Since analog filter design has been researched in detail, this provides a solid background from which to design IIR filters. It will be shown later that the output of an IIR filter depends both on the input signal and feedback from the output signal. IIR filters are sometimes referred to as recursive filters since the output result is also used as a feedback signal as will be shown in the filter block diagram (Figure 1.2).
The FIR filter is derived purely by DSP theory and does not relate to analog filter design. The output of an FIR filter depends only on the input signal, and this results in some key advantages. An FIR filter is always stable and can be designed to have a linear phase response. IIR filters, on the other hand, can never have a linear-phase response, and if designed so that the poles of the IIR filter are outside the unit circle in the z-domain, the IIR filter would also be unstable. The stability of FIR filters is a key to adaptive filters which change their own coefficients in an intelligent way because it is not possible to create an unstable filter. When the signal shape is important to the application, then linear phase must be obtained in the filter. Data communications is an example where linear phase response is a requirement. However, for voice communications applications, the linear phase response is not a concern because the human ear responds to energy in the frequency domain.

IIR filter design will be discussed in this research paper in a fair amount of detail. However, FIR filter design is beyond the scope of this paper. Both types of digital filters are useful for various applications. Next, some basic concepts to aid in the understanding of digital filters will be presented to prepare the reader for the discussion of IIR filter design. These concepts are more applicable to IIR filters than FIR filters, but they will provide insight into the field of digital signal processing.

1.2 Digital Filter Representations

There are five main ways to describe a digital filter: create a difference equation, a block diagram, a transfer function \( H(z) \), an impulse response \( h(n) \), or a pole/zero plot. Each description can be used to fully specify a digital filter. It is
necessary to understand how to interpret these representations and how to convert between the different forms. This section will discuss these five different representations that are referred to in other digital signal processing sources.

1.2.1 Difference Equation

The first means to describe a digital filter is through the use of a difference equation. The difference equation specifies the output $y(n)$ as a function of the input $x(n)$ and the previous outputs as shown below:

$$y(n) = a_0 x(n) + a_1 x(n-1) + \ldots + a_M x(n-M) - b_1 y(n-1) - b_2 y(n-2) \ldots - b_N y(n-N)$$

If all the $b_k$ coefficients are zero, then we are describing an FIR filter, with no feedback from the output. If at least one of the $b_k$ coefficients is non-zero, then an IIR filter is being represented. Note that the $n$ represents the individual digitized samples as a function of time, assumed to be sampled periodically so that the time between samples is consistent. All digital filters can be represented using a difference equation; the coefficients $a_k$ and $b_k$ determine the type of filter that is implemented and the response of the filter.

1.2.2 Block Diagram

The logical next step to describing a filter is to graphically represent the difference equation for the given filter. This will allow for a more intuitive method of
understanding filters. Following is the block diagram for the difference equation presented above:

![Diagram of the block diagram for the difference equation]

Figure 1.2 Digital Filter Block Diagram

The block diagram illustrates the recursiveness of the IIR filter since it is easily seen that the current output depends on both the input and the previous outputs. If all the $b_k$ terms were zero, then the block diagram would represent an FIR filter with no feedback from the output signal.

### 1.2.3 Transfer Function $H(z)$

To determine the transfer function of a digital filter, a simple technique is to take the Z-transform of the difference equation. Taking the Z-transform yields

\[
Y(z) = X(z) \left[ a_0 + a_1 z^{-1} + \ldots + a_M z^{-M} \right] Y(z) \left[ b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^{-N} \right]
\]

\[
Y(z) \left[ 1 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^{-N} \right] = X(z) \left[ a_0 + a_1 z^{-1} + \ldots + a_M z^{-M} \right]
\]

The transfer function $H(z)$ will be defined as the ratio of the output to the input:

---

6
Y(z) = a_0 + a_1 z^{-1} + ... + a_M z^{-M} \\
X(z) = 1 + b_1 z^{-1} + b_2 z^{-2} + ... + b_N z^{-N}

Rewriting the equation to create positive exponents for the z-terms yields:

H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 z^M + a_1 z^{M-1} + ... + a_M}{z^N + b_1 z^{N-1} + b_2 z^{N-2} + ... + b_N}

Note that often, the factor [z^{N-M}] will not be shown because most IIR filters have the same number of delay stages for the input side as for the output side of the block diagram. Thus, since M=N, [z^{N-M}] is equal to one and does not need to be shown in the transfer function.

Once the transfer function has been derived, the output Y(z) can be found for any input X(z) by simple multiplication in the z-domain as shown:

Y(z) = X(z) H(z)

Next, the impulse response in the time-domain will be discussed and correlated with the transfer function in the z-domain.

**1.2.4 Impulse Response h(n)**

The impulse response h(n) is defined as the inverse Z-transform of the transfer function. It is called the impulse response because h(n) describes what the output y(n) would look like if the input x(n) was a unit impulse function. A unit impulse function p(n) is defined as follows:
\[
\begin{align*}
p(n) &= \begin{cases} 
0 & \text{for } n < 0 \\
1 & \text{for } n = 0 \\
0 & \text{for } n > 0 
\end{cases}
\end{align*}
\]

Since the impulse response is simply the filter response to a unit impulse function input, \( h(n) = y(n) \) when the input \( x(n) = p(n) \).

As described previously, the output \( Y(z) \) can be computed in the \( z \)-domain by simply multiplying the transfer function \( H(z) \) by the input \( X(z) \). In general, multiplication in the \( z \)-domain is equivalent to convolution (represented by \( * \)) in the time-domain. Thus, as for any linear time-invariant system, the discrete output response \( y(n) \) in the time-domain is given by the equation below, where \( x(n) \) represents the discrete input and \( h(n) \) represents the impulse response:

\[
y(n) = x(n) * h(n)
\]

Using the definition of discrete convolution results in the following two modified forms of the equation above:

\[
y(n) = \sum_{k = -\infty}^{\infty} x(k) h(n-k) = \sum_{k = -\infty}^{\infty} h(k) x(n-k)
\]

Thus, the impulse response \( h(n) \) can be used to determine the output response \( y(n) \) in the time-domain in the same manner as the transfer function \( H(z) \) is used to determine the output response \( Y(z) \) in the \( z \)-domain. For a causal IIR filter, where the impulse response \( h(n) \) is zero for \( n < 0 \) (since practical filters cannot predict that an impulse input will occur before it actually does), the limits of \( k \) for the summation are \( 0 \) to \( \infty \).
1.2.5 Pole/Zero Plot

A final method of representing a filter is through a pole/zero plot in the z-domain. The transfer function \( H(z) \) can be factored as shown:

\[
H(z) = \frac{a_0 z^M + a_1 z^{M-1} + \ldots + a_M}{z^N + b_1 z^{N-1} + b_2 z^{N-2} + \ldots + b_N} = K \frac{(z-z_1)(z-z_2) \ldots (z-z_M)}{(z-p_1)(z-p_2) \ldots (z-p_N)}
\]

The parameters \( z_k \) indicate the location of the zeroes of the filter while the \( p_k \) parameters indicate the location of the poles of the filter. Note that \( K \) is considered to be a gain constant. This constant will not be represented in the pole/zero plot. However, the filter functionality is described accurately using the pole/zero plot representation. The conversion between representations of the digital filters would vary by a gain constant, but because we are dealing with digital filters, the gain can be adjusted to whatever is desirable without changing the operation of the filter. The filter will still pass and reject the same frequencies as it would when modeled using one of the other four representations.

Suppose that our transfer function was

\[
H(z) = \frac{[z - (0.2 + j0.8)] [z - (0.2 - j0.8)] [z - (-1)]}{[z - 0.6] [z - (0.9 + j0.1)] [z - (0.9 - j0.1)]}
\]

Then, the filter would have three zeroes and three poles. The zeroes would be at the complex-conjugate locations \( z_1 = (0.2 + j0.8) \) and \( z_2 = (0.2 - j0.8) \) as well as at \( z_3 = -1 \). The poles would be at \( p_1 = 0.6 \) and at the complex pole locations \( p_2 = (0.9 + j0.1) \) and
\( p_3 = (0.9 - j0.1) \). A pole/zero plot is made by simply plotting the complex points in the z-plane as shown below:

![Figure 1.3 Digital Filter Pole/Zero Plot](image)

As mentioned earlier, the pole/zero plot does not indicate a gain constant. However, all five methods of representing digital filters are equivalent filter descriptions within a gain constant.

### 1.3 Derivation of the Bilinear Z-Transform

Next, the derivation for the Bilinear Z-Transform (BZT) will be explained. It is included here to help the reader to understand what the BZT does. Having understood this background theory, the reader will understand why the filter specification frequencies must be prewarped using the BZT.

The BZT is a transformation between the continuous s-domain and the discrete z-domain. To derive the BZT, let us begin by attempting to create a discrete
approximation of an integrator. Computing the integration of a function, f(t), in the
time-domain is equivalent to multiplying the Laplace transform of that function, F(s),
by a factor 1/s in the s-domain. But what is this multiplication factor in the z-domain?
To derive this, a definition of integration in the time domain is useful. From calculus,
it is well known that the integration of a function is equivalent to determining the area
under the curve of the function with respect to the x-axis. Suppose we have a function
g(α) that is to be integrated between negative infinity and t.

![Figure 1.4 Graph of Function g(α) For BZT Derivation](image)

Integration can be modeled by the following simple block diagram.

\[ v(t) = \int_{-\infty}^{t} g(\alpha) \, d\alpha \]

When sampling discretely, as in the z-domain, one can only sample at discrete points
throughout the function. Thus, a sampling frequency, T, must be chosen. Thus, if k is
chosen to be an integer value representing the individual discrete samples taken, then
$kT$ can be used as an approximation for the continuous variable $t$ in the previous equation.

$$v(kT) = \int_{-\infty}^{kT} g(\alpha) \, d\alpha$$

This equation can be broken down into two pieces: the contribution of all the previous samples taken up to the $k-1^{st}$ sample and the contribution that the $k^{th}$ sample has added.

$$v(kT) = \int_{-\infty}^{kT} g(\alpha) \, d\alpha = \int_{-\infty}^{(k-1)T} g(\alpha) \, d\alpha + \int_{(k-1)T}^{kT} g(\alpha) \, d\alpha$$

Notice that the first term is equivalent to the function $v[(k-1)T]$, resulting in a recursive function:

$$v(kT) = v[(k-1)T] + \int_{(k-1)T}^{kT} g(\alpha) \, d\alpha$$

Note additionally that the second term is simply the area under the curve $g(\alpha)$ between the points $g[(k-1)T]$ and $g(kT)$. Assuming that the sampling period $T$ is short, then the area under the curve can be approximated by the trapezoidal area as shown below:
Thus, the function \( v(kT) \) can be rewritten as

\[
v(kT) = v[(k-1)T] + T \frac{g(kT) + g[(k-1)T]}{2}
\]

Often in discrete systems, the sampling period \( T \) is understood to exist, and \( kT \) is replaced simply by \( k \).

\[
v(k) = v(k-1) + T \frac{g(k) + g(k-1)}{2}
\]

Thus, the discrete integration operation has been derived. If the integration operation is discretely applied to the sampled function \( g(k) \), then the result is the discrete output function \( v(k) \).
If the Z-transform is applied to the previous equation, the transfer function for integration can be obtained.

\[ V(z) = z^{-1} V(z) + \frac{T}{2} G(z) \left[ 1 + z^{-1} \right] \]

\[ H_{\text{INTEGRATION}}(z) = \frac{V(z)}{G(z)} = \frac{T}{2} \left[ \frac{1 + z^{-1}}{1 - z^{-1}} \right] = \frac{T}{2} \left[ \frac{z + 1}{z - 1} \right] \]

As mentioned previously, the transfer function for integration in the s-domain is simply equal to \((1/s)\). Thus, if the transfer functions are equated, then the Bilinear Z-Transform between the s-domain and z-domain is obtained.

\[ \frac{1}{s} \quad \rightarrow \quad \frac{T}{2} \left[ \frac{z + 1}{z - 1} \right] \quad \text{OR} \quad s \quad \rightarrow \quad \frac{2}{T} \left[ \frac{z - 1}{z + 1} \right] \]
Chapter 2: IIR Digital Filter Design

2.1 Filter Specifications

The following analysis will be taken from reference [24] and previous knowledge of filter design.

2.1.1 Original Design Specifications

To design a filter, one must know certain specifications that the filter must meet to suit the application. Note that these specifications may be refined to be consistent with other goals that the user may have, such as complexity. To begin, the filter that is to be designed should be selected from the four main types of filters, choosing a low-pass filter, high-pass filter, band-pass filter, or band-stop filter. The ideal filter response for each type of filter was presented graphically in Figure 1.1.

Figure 1.1 showed what specifications are required for ideal filters. Practical filters cannot achieve a perfect cutoff frequency such that the magnitude response changes from zero to one at a single frequency. In reality, a range must be specified in which the filters change from a passband state to a stopband state and vice-versa. Also, the magnitude response of the passband is usually specified (in dB) as the maximum amount of ripple that is acceptable. Thus, if the maximum ripple of the passband is specified to be 0.75 dB (minimum gain $= \log^{-1} (-0.75dB / 20) = 0.92$), then the minimum gain and maximum gain of the output with respect to the input signal for frequencies within the passband must be between 0.92 and 1.00.
Similarly, the magnitude response of the filter for the stopband is usually specified (in dB) as the minimum attenuation for frequencies within the stopband. Thus, if the minimum attenuation in the stopband is set to 36 dB (\(\text{maximum gain} = \log^1(-36\,\text{dB} / 20) = 0.016\)), then the maximum gain of the output with respect to the input signal for frequencies within the stopband must be 0.016. Thus, the realizable filter can block out those frequencies of the stopband such that their effects are limited. Note that as the filter parameters get more stringent and the desired response is closer to ideal, the complexity of the filter design increases.

Once a filter type is selected, the frequencies that are relevant to the filter of choice should be selected as well as the desired magnitude response. For example, the band-stop filter can be described by four frequencies and two desired magnitude response gains. The four frequencies of interest include the lower passband frequency (where the first passband ends), the lower stopband frequency (where the stopband begins), the upper stopband frequency (where the stopband ends), and the upper passband frequency (where the second passband begins). As described above, the two desired magnitude response gain parameters are the maximum ripple in the passband and the minimum attenuation in the stopband. Note that in the passband of the filter, it is often desirable to have a gain of one. Thus, all frequencies that are meant to pass through the filter do so such that the magnitude of the output is equal to that of the input to the filter.

Another parameter of interest when designing a digital filter is the sampling frequency. From the Nyquist principle, it is well known that to accurately sample
frequencies, the maximum frequency in the input signal to a digital circuit should be no greater than half the sampling frequency. Aliasing occurs when a signal at one frequency appears to look like a slower frequency signal as a result of the system sampling rate being too slow to accurately reproduce the input frequency. This aliasing is usually unwanted. To prevent this, the sampling frequency of the filter should be chosen to be at least twice the highest frequency contained in the input signal. If the maximum frequency of the input signal is unknown, it may be possible to use an analog low-pass filter to limit the frequencies of the input to those which are less than half the chosen sampling rate.

To summarize, the filter specifications that are required usually are specified as passband and stopband gains as well as the frequency ranges through which the filter is allowed to change states between passing frequencies and blocking them. The next topic will include a discussion on how the specifications of the filter will affect the complexity of the filter.

2.1.2 Filter Order Complexity

To design a digital filter, the Bilinear Z-Transform will be used to transform an analog filter into a digital filter. Other types of transformations that could have been used include impulse-invariance (to create a digital filter with the same impulse response as the parent analog filter) and step-invariance (to create a digital filter with the same step response as the parent analog filter) designs. The BZT method of design creates a digital filter whose frequency response matches the parent analog
filter. This results in a much better filter than can be created using the other two methods.

The parent analog filter that will be used in the following derivations is a Butterworth filter. A Butterworth filter is a type of analog filter whose spacing between the poles is known to be $360^\circ/2n$ for low-pass filters, where $n$ indicates the number of poles (or the order of the filter). A Butterworth low-pass filter does not have any zeroes in the s-domain.

Given the filter parameters, the order of the Butterworth analog filter needed to meet the specifications must be computed. To begin, the frequency parameters must be converted from Hertz to rad/sec by multiplying the frequencies by $2\pi$ (assuming the frequencies were originally specified in Hertz). To design a low-pass, high-pass, band-pass, or band-reject analog filter, it is sometimes easier to work backwards to find the equivalent complexity one-radian low-pass filter. The order of this normalized Butterworth low-pass counterpart will be computed, having a passband that ends at 1 rad/sec and a stopband beginning at $\omega_4$ as calculated in Figure 2.1(a)-(d). The result $\omega_4$ will be used to derive an intermediate analog filter $H_{ai}(s)$ that can be transformed into the more complicated parent analog filter $H_{ap}(s)$. This parent analog filter will then be transformed into the digital filter $H(z)$. Figure 2.1(a)-(d) shows the four types of analog-to-analog transformations. Note once again that the normalized prototype filter $H_{ap}(s)$ will be transformed into a parent analog filter $H_{ap}(s)$ through an intermediate step, creating the intermediate filter $H_{ai}(s)$. In the figure below, the “Filter Response Desired” refers to the response of the analog filter having
the same frequency specifications as the digital filter to be designed. Thus, if a low-pass digital filter was being designed, $\omega_s$ would indicate the end of the passband and $\omega'_s$ would indicate the start of the stopband for the digital filter.

Case 1: Low-Pass Filter Design

Low-Pass Filter Response Desired

Low-Pass Prototype Response

$$\omega_k = \frac{\omega'_s}{\omega_s}$$

Figure 2.1(a) Low-Pass Filter Prototype Parameters

Case 2: High-Pass Filter Design

High-Pass Filter Response Desired

Low-Pass Prototype Response

$$\omega_k = \frac{\omega_s}{\omega'_s}$$

Figure 2.1(b) High-Pass Filter Prototype Parameters
Case 3: Band-Pass Filter Design

Band-Pass Filter Response

Desired Low-Pass Prototype Response

Magnitude

\[ \cot \theta = \text{minimum} \]

\[ \left\{ \begin{array}{c}
\cos \omega_c = \cos \left( \omega_{\text{lower}} \right)
+ \cos(\omega_{\text{upper}} - \omega_{\text{lower}})
+ \cos(\omega_{\text{upper}} - \omega_{\text{lower}})
\end{array} \right\} \]

Figure 2.1(c) Band-Pass Filter Prototype Parameters

Case 4: Band-Stop Filter Design

Band-Stop Filter Response

Desired Low-Pass Prototype Response

Magnitude

\[ \cot \theta = \text{minimum} \]

\[ \left\{ \begin{array}{c}
\cos \omega_c = \cos(\omega_{\text{upper}} - \omega_{\text{lower}})
+ \cos(\omega_{\text{upper}} - \omega_{\text{lower}})
+ \cos(\omega_{\text{upper}} - \omega_{\text{lower}})
\end{array} \right\} \]

Figure 2.1(d) Band-Stop Filter Prototype Parameters

Figure 2.1 Low-Pass Filter Prototypes for the Four Filter Types
Once the stopband frequency \( \omega_s \) has been calculated for the desired low-pass, high-pass, band-pass, or band-stop filter as indicated by the four calculations above, then the order, \( n \), of the filter can be computed using the following equation:

\[
 n = \text{ceil} \left\{ \frac{\log \frac{10^{(\alpha / 10)} - 1}{10^{(\beta / 10)} - 1}}{2 \log \frac{1}{\omega_s}} \right\}
\]

Note that the ceiling function indicates that the result must be rounded up to an integer value. The order, \( n \), that the above equation provides is the order that is necessary for the prototype low-pass filter. For low-pass and high-pass filter designs, the order of the filter will be the same as \( n \) when the prototype filter is transformed to the desired filter. However, for the band-pass and band-stop filter designs, the order of the filter will be twice the value of \( n \) when the prototype low-pass filter is transformed into the more complicated band-pass or band-stop filter.

Based on the filter order, passband attenuation, and stopband attenuation, the cutoff frequency \( \Omega_c \) can be found that will be used to transform the normalized low-pass analog filter into an intermediate analog filter that can be used to meet the desired attenuation requirements. The intermediate filter will have its passband and stopband altered by a factor of \( \Omega_c \), resulting with the passband ending at \( \Omega_c \) and the stopband beginning at \( \Omega_c \Omega_c \). Note that if the result of \( n \) before the ceiling function is non-integer, the filter of order \( n \) will exceed the design specifications. If it is desirable to
satisfy the requirement of the passband attenuation and exceed the minimum stopband attenuation, the following equation should be used to compute $\Omega_c$:

$$\Omega_{c1} = \frac{1}{[10^{(\omega/10)^{(1/2)n}} - 1]}$$

The 1 in the numerator is the end of the passband for the normalized low-pass filter that the intermediate filter will be derived from. This indicates that the end of the passband was at 1 rad/sec.

However, if one had wanted to meet the minimum stopband attenuation exactly and exceed the passband attenuation requirement, then the following could have been used instead:

$$\Omega_{c2} = \frac{\omega}{[10^{(\beta/10)^{(1/2)n}} - 1]}$$

To exceed both requirements, one could choose a value for $\Omega_c$ between $\Omega_{c1}$ and $\Omega_{c2}$.

$$\Omega_{c1} \leq \Omega_c \leq \Omega_{c2} \quad \text{if } \Omega_c \leq \Omega_{c2}$$

$$\Omega_{c2} \leq \Omega_c \leq \Omega_{c1} \quad \text{if } \Omega_c > \Omega_{c2}$$

### 2.1.3 Prewarping Filter Frequencies to Correct for BZT

Next, the filter frequencies for the filter that is to be designed must be prewarped to correct for the nonlinearities of the Bilinear Z-Transform. The BZT is used to map the $s$-plane (entire plane left of the $j\omega$ (vertical) axis) into the unit circle of the $z$-domain. Mapping an infinite plane into a finite circle will cause warping that cannot be avoided. However, the warping of the frequencies from the analog filter $H_a(s)$ to the digital filter $H(z)$ that will occur is understood, and if corrections are made before the BZT is applied, then the results after the BZT will yield the desired filter.
BZT Mapping:

\[ H(z) = H_a(s) \quad \text{s} \rightarrow \frac{2}{T} \left[ \frac{z - 1}{z + 1} \right] \]

The BZT mapping equation above can be illustrated graphically below.

As shown above, mapping an infinitely long line to a finite semicircle length is performed by the BZT. The BZT equation presented earlier on page 14 is solved for \( s \) in terms of \( z \). This relationship is bidirectional, and if it is solved for \( z \) in terms of \( s \), the following BZT equation is achieved:

\[ z = \frac{(2/T) + s}{(2/T) - s} \]

In the s-domain, \( j\omega \) can be substituted for \( s \) without changing the equation. Solving for \( z \) in terms of polar notation yields the following:

\[ z = \frac{(2/T) + j\omega}{(2/T) - j\omega} = \frac{e^{j\tan^{-1}(\omega T/2)}}{\sqrt{(2/T)^2 + \omega^2}} \cdot \frac{e^{j\tan^{-1}(\omega T/2)}}{e^{-j\tan^{-1}(\omega T/2)}} = e^{j\tan^{-1}(\omega T/2)} = e^{j\theta} \]
where $\theta = 2 \tan^{-1}(\omega T/2)$. Thus, values for $\omega$ in the s-domain that range from zero to positive infinity are mapped to an angle $\theta$ in the z-domain between 0 and $\pi$. Likewise, values for $\omega$ in the s-domain that range from zero to negative infinity are mapped to an angle $\theta$ in the z-domain between 0 and -$\pi$. The nonlinearity in this relationship is illustrated by the graph of the mapping equation as shown below:

![Graph of Nonlinearity Between s-Domain and z-Domain](image)

Figure 2.3 Graph of Nonlinearity Between s-Domain and z-Domain

Now, that the nonlinear relationship between the s-domain and z-domain is understood, the next step is to look at what can be done to correct for this before applying the BZT equation.

Suppose that we intend to design a low-pass filter that passes all frequencies less than $\omega_{ce}$ (band-edge frequency), indicating the low-pass cutoff frequency for the desired digital filter as represented by $\omega_c$ for Case 1 of Figure 2.1 on page 19. It will also be assumed that the maximum frequency $\omega_{max}$ will be less than or equal to half the digital filter sampling frequency (Nyquist frequency). The analog input function $x_a(t)$ is to be sampled with a period $T$, and the sampled output $x(n)$ is the result of the
sampling process. Figure 2.4 shows the frequency warping relationship for filter designs.

![Diagram of Frequency Spectrum and Z-domain](image)

**Original Analog Filter Specification**

**Apply BZT to parent analog filter**

Intuitively, one would choose \( \omega_c \), the cutoff frequency for the parent analog filter, to be equal to \( \omega_{be} \). When the parent analog filter is transformed into the desired digital filter using the BZT equation, frequency warping results that creates an inaccurate digital filter. Instead, the parent analog filter should be adjusted by choosing a value for \( \omega_c \) that will result in the desired digital filter after the BZT is applied.

![Diagram of Frequency Warping](image)

Figure 2.4 Frequency Warping in Filter Design

It is desirable to design the parent analog filter so that \( \theta_c = \theta_{be} \). Substituting in this equation and solving for \( \omega_c \) yields the following:
2 \tan^{-1}(\omega_c T/2) = \omega_c T

\omega_c = \frac{2}{T} \tan \left( \frac{\omega_c T}{2} \right)

Thus, when transforming analog filters into digital filters using the BZT technique, the
frequencies of the parent analog filter (\omega_c) should be modified using the equation
above to adjust for the nonlinearities inherent to the transformation. The resulting
frequencies \omega_c should be used in place of the frequencies \omega_c. The required
conversions for the four main types are filters are given below:

Low-Pass and High-Pass Filters:

\Omega_u = \frac{2}{T} \tan \left( \frac{\omega_u T}{2} \right)

Band-Pass and Band-Stop Filters:

\Omega_{lower} = \frac{2}{T} \tan \left( \frac{\omega_{lower} T}{2} \right)

\Omega_{upper} = \frac{2}{T} \tan \left( \frac{\omega_{upper} T}{2} \right)

2.2 Designs Based on Normalized Low-Pass Filter

When calculating the filter order in Section 2.1.2, the normalized low-pass
filter was described as having its passband end at 1 rad/sec and its stopband begin at
\omega_c. This filter will be transformed into an intermediate filter having its passband end at
\Omega_c and stopband begin at \Omega_c. For a filter of order n, there are n number of poles.
The poles of a filter must be in the left half of the s-plane (inside the unit circle for the
z-domain) for the filter to be stable. For a Butterworth low-pass filter, the angular
spacing between the poles in the s-domain is 180 degrees divided by n, with the poles
being centered around the negative $\sigma$ axis. Thus, for a filter of order three, the poles are separated by 60 degrees. The first pole is at $s_1 = -1$, while the other two poles, $s_2$ and $s_3$, are at a radius one (because the normalized filter cutoff frequency is 1 rad/sec) from the origin at $\pm 60$ degrees from the negative $\sigma$ axis. The general equation for the normalized Butterworth low-pass filter (passband ending at 1 rad/sec) is shown below:

$$H_{an}(s) = \frac{1}{(s - s_1)(s - s_2)(s - s_3)}$$

To transform the normalized Butterworth low-pass analog filter to the intermediate analog filter that can be used to meet the attenuation requirements, the following transformation must take place in $H_{an}(s)$ using a value between $\Omega_{c1}$ and $\Omega_{c2}$ for $\Omega_c$ (from page 22):

$$H_{ai}(s) = H_{an}(s) \left| \begin{array}{c} s \rightarrow \frac{s}{\Omega_c} \end{array} \right.$$  

This transforms the normalized Butterworth analog filter into an intermediate analog filter that will guarantee that the passband and stopband attenuation specifications are met. The next step is to take the new intermediate analog filter $H_{ai}(s)$ and convert it to the desired filter type as shown in the next section.

**2.2.1 Transformations Based on Normalized Low-Pass Filter**

From Section 2.1.2, the four types of filters that can be designed were presented along with the abbreviations for the filter specifications. In this section, it is necessary to transform the intermediate filter $H_{ai}(s)$ into a parent analog filter $H_{ap}(s)$ so that the BZT can then be applied, yielding the desired digital transfer function $H(z)$.  

---

27
2.2.1.1 Intermediate Low-Pass -> Denormalized Low-Pass Transform

\[ H_{ap}(s) = H_{ai}(s) \]
\[ s \rightarrow \frac{s}{\Omega_u} \]

2.2.1.2 Intermediate Low-Pass -> Denormalized High-Pass Transform

\[ H_{ap}(s) = H_{ai}(s) \]
\[ s \rightarrow \frac{\Omega_u}{s} \]

2.2.1.3 Intermediate Low-Pass -> Denormalized Band-Pass Transform

\[ H_{ap}(s) = H_{ai}(s) \]
\[ s \rightarrow \frac{s^2 + \Omega_{lower} \Omega_{upper}}{s(\Omega_{upper} - \Omega_{lower})} \]

2.2.1.4 Intermediate Low-Pass -> Denormalized Band-Stop Transform

\[ H_{ap}(s) = H_{ai}(s) \]
\[ s \rightarrow \frac{s(\Omega_{upper} - \Omega_{lower})}{s^2 + \Omega_{lower} \Omega_{upper}} \]

Note that the transformation between \( H_{an}(s) \) and \( H_{ap}(s) \) could have been done in one step if the transformation for converting \( H_{an}(s) \) to \( H_{ai}(s) \) was combined with the above transformation. The parent analog filter described by the transfer function \( H_{ap}(s) \) will next be transformed into the digital filter described by \( H(z) \) using the BZT equation.

2.2.2 Converting \( H(s) \) -> \( H(z) \) Using the Bilinear \( Z \)-Transform

To convert the analog transfer function into a digital filter transfer function, the BZT technique will be used as shown once again below:

\[ H(z) = H_{ap}(s) \]
\[ s \rightarrow \frac{2}{T} \left[ \frac{z - 1}{z + 1} \right] \]
The resulting transfer function $H(z)$ can be converted to any one of the five digital filter representations described in Chapter 1.

### 2.2.3 Biquad-Section Design

Using methods described in Chapter 1, the transfer function $H(z)$ can be converted to a listing of the poles and zeroes of the filter. A biquad-section uses two poles and two zeroes. Thus, if one was designing a band-reject filter and originally calculated $n$ to be 3, then the actual order of the filter is 6 (since the conversion from normalized low-pass to denormalized band-reject doubled the order of the digital filter). This filter can then be divided into three cascaded biquad-sections, with each biquad-section using two poles and two zeroes from the transfer function $H(z)$. It is important to pair the complex-conjugate poles together and the complex-conjugate zeroes together when creating a biquad-section so that the coefficients to the filter are real-valued.

Before going into detail, it is important to understand why biquad-section designs are used. Biquad-section designs are used in IIR filter design because less hardware can be used to create more accuracy. Suppose a 10-pole, 10-zero digital filter was being designed. The transfer function would be as follows:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 z^{10} + a_1 z^9 + \ldots + a_{10}}{z^{10} + b_1 z^9 + b_2 z^8 + \ldots + b_{10}} = \frac{(z-z_1)(z-z_2)\ldots(z-z_{10})}{(z-p_1)(z-p_2)\ldots(z-p_{10})}$$

The accuracy of the ten coefficients $b_k$ affect the location of the ten poles $p_j$. The pole locations change according to the following relationship:
\[ \Delta p_j = K \Delta b_k \quad \text{where } K = 10^{(\text{number of poles})} \]

Thus, when building a 10-pole filter as a single stage, a slight change in coefficients (as a result of quantization of coefficients) results in a much greater shift \((10^{10} \text{ instead of } 10^2)\) in the pole locations than if the 10-pole filter is built in five cascading stages of biquad-sections.

A single biquad-section contains two poles and two zeroes and can be represented by the following transfer function and block diagram:

\[
H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 z^2 + a_1 z + a_2}{z^2 + b_1 z + b_2}
\]

`Figure 2.5 Block Diagram for a Single Biquad-Section`

Note that each time-delay element is actually a register in hardware that stores the previous reading. The number of time-delay elements can be reduced if the block diagram is rearranged as shown below:
Figure 2.6 Simplified Block Diagram for Biquad-Section

This block diagram is an equivalent representation that uses two less registers per biquad stage. A complete design example will be presented in Chapter 4.
Chapter 3: Mentor Graphics Automated Design Tools

3.1 Introduction to Tools

In creating an ASIC (Application Specific Integrated Circuit) geared toward digital signal processing (DSP) applications, Mentor Graphics has developed some automated tools referred to as the DSP Station tools. These tools provide a means for creating a high-level description of a digital filter application, referred to as a DFL (design-flow logic) description. This high-level description can be simulated to determine how many bits of accuracy are required within the filter to achieve the desired results. Once the filter DFL has been successfully simulated, another tool can generate the VHDL (VHSIC Hardware Description Language, where VHSIC is an abbreviation for Very High Speed Integrated Circuit) for the digital filter. The VHDL can be simulated, and a schematic can be generated from the VHDL model. The schematic can be integrated into a chip-level schematic, and this can be used to automatically generate the VLSI (Very Large Scale Integration) layout. To perform these steps, various Mentor Graphics tools were used as shown on the following page. A complete design description is detailed in Chapter 4.

3.2 FILlab: High-Level Filter Design and Simulation

The program FILlab can be used as a starting point to designing a digital filter. Both IIR and FIR filters can be designed at a high level, and this package can save the design as a DFL description. This package is useful in estimating the order of the filter
Figure 3.1 Road Map Through Mentor Graphics Tools
that will be required to achieve certain filter specifications. The DFL generated may easily be modified to change the filter coefficients, if desired.

3.3 **Design Architect: DFL and Schematic Editor**

The DFL that is generated from FILlab may be modified using the Design Architect program. Design Architect can be used to both create new DFL’s and edit existing ones. This also will compile the DFL so that any syntax errors are resolved before simulation.

Design Architect can also be used as a schematic editor. The schematic that will be generated from the VHDL can be easily be represented as a symbol in Design Architect. This symbol can be linked to symbols representing the VLSI chip pads. Design Architect is used to create and edit schematics which can be compiled and then used to generate the VLSI layout.

3.4 **DSPlab: Simulation Tools for DFL**

The DSPlab tools allow the user to simulate the DFL that is created by FILlab or Design Architect. Both time-domain and frequency-domain simulations can be performed using the DSPlab tools. This simulation software provides graphs for the results for both a high-level simulation model (full resolution of coefficients used with maximum accuracy for both multiplications and additions) and a bit-true simulation model (resolution of coefficients, multiplications, and additions are quantized to the specified number of bits and simulated accordingly).
For the time-domain simulation, DSPlab allows the user to select an input waveform. The time-domain simulation presents the input and output waveforms graphically as a function of time. Input waveforms may be generated using the built-in waveform editor, allowing the user to enter a sine wave easily by specifying the magnitude, DC offset, and frequency. This input waveform will be connected to the filter input, and the result of the filter can be displayed graphically for both the high-level and bit-true simulation. This also allows the user to overlay the input waveform on the output waveform to observe the effects of the filter.

The frequency-domain simulation allows the user to select a range of frequencies to sweep through the digital filter under test. DSPlab allows the user to select the step size between consecutive frequencies tested. The result from DSPlab is the magnitude response of the filter. This graph illustrates the frequency response of the digital filter and can be used to verify that the passband attenuation and stopband attenuations will be met with the current DFL design.

3.5 **Mistral1/HDL: VHDL Generation/Compilation Tools**

Mistral1 allows the user to generate structural VHDL (as well as some other types of netlist models) from the compiled DFL. Mistral1 will create VHDL that uses the number of bits specified in the DFL that was simulated in DSPlab. Using the Mistral1 analysis tools, the number of delay stages between the input and output is given so that the timing of the VHDL is understood. This information is displayed when analyzing the results of the Mistral1 run.
The Mistral1 tools use a concept referred to as bit-serial design to implement the DFL design. In a bit-serial design, the same hardware is used for computing all the bits of a 12-bit operation, for example. In a bit-parallel design, there is hardware to handle the entire word size (12-bits). When values are shifted in a bit-parallel design, the entire word size is shifted at the same time to separate hardware. However, for the bit-serial design, the least significant bit of the values are shifted in first. Then, the bits are shifted in one at a time until the most-significant bit of the word is shifted in. The same hardware is reused as the individual bits are shifted in. This results in reuse of hardware and is more efficient, particularly for filter designs where the speed of the results of the bit-serial design is acceptable.

The VHDL that is generated by Mistral1 can be compiled using the HDL compiler. This compiler will compile the VHDL from the command line if the Mistral1 library location is specified. This library is then linked in, and the compiled VHDL can be used by AutoLogic to generate the schematic for the filter.

Note that the Mentor Graphics Mistral1 tools used have not currently implemented full functionality that the DFL syntax provides. Thus, there are certain limitations in the current software which restrict certain types of automation. For example, the operators / (division), div (integer division), mod (integer remainder), and ** (power function) have not been implemented. Also, pure floating-point operations are not supported, and thus, fixed-point representations must be used. Addition and subtraction operands must be fixed-point representations with the same wordlength.
and fractional length. In Mistral1, delay initializations are ignored, so the chip must be initialized by preloading the scan chains with known values.

Another limitation in the Mistral1 tools is that it does not provide the flexibility for filter coefficients to be scanned in and modified so that the filter operation can be modified. On the other hand, Mistral1 allows for minimizing the hardware used to implement a specific filter with a specific set of filter coefficients. The filter coefficients are represented in signed digit notation (prefixed by 0sd) as a series of ternary digits (0, 1, or -1, with -1 represented by a T) to minimize the amount of hardware necessary to multiply filter values by the coefficients. For example, 0.875 can be represented as 0sd1.00T (1 0.125, since T in the third decimal position indicates to subtract $2^{-3}$) instead of being represented in binary decimal form 0.111 (0.5 + 0.25 + 0.125, where only addition of $2^n$ factors is allowed). This results in one less operation that is needed to multiply 0.875 by another number. Thus, the power of Mistral1 lies with application-specific designs where the filter coefficients are known, and it is desirable to minimize the hardware for use with these coefficients. Although the result does not provide full flexibility to load other filter coefficients, it uses much less hardware than would be required to implement that added flexibility.

3.6 AutoLogic: Synthesizing VHDL -> Schematic

AutoLogic can be used to generate a digital schematic for a VHDL design. The compiled VHDL is loaded into AutoLogic, and the user can instruct AutoLogic to optimize for minimizing the area required for the layout. In AutoLogic, the user must select the standard cell library that will be used by IC Station when generating the
layout. For the design example of Chapter 4, the CMOSN 1.2 micron standard cell library was used. The schematic that results can be saved as a design viewpoint. This schematic can be used as a symbol in Design Architect and can be connected to symbols of pads for a chip design. The schematic from Design Architect can then be used by the Design Viewpoint Editor to complete the next step.

3.7 DVE: Design Viewpoint Checking and Validation

The Design Viewpoint Editor (DVE) can be used to compile and check the schematic for ERC (Electrical Rules Checking) errors. If no errors are found, a design viewpoint is generated for the schematic. This design viewpoint can be used to simulate the design at a bit level and for automatically generating the VLSI layout.

3.8 QuickSim II: VHDL Simulation/Verification

QuickSim II allows the user to simulate the design viewpoint that has been generated from the VHDL. This allows the user to control the signals on the input pads and observe the signals on the output pads of the chip. The input signals, however, must be specified as forces, either low or high, and the times that the signals change state. Simulating a filter at the bit-level for a bit-serial design is tedious, but this was performed in the design example in Chapter 4 to ensure that the timing of the chip was understood.

3.9 IC Station: Schematic -> VLSI Layout

Once there is confidence that the design is correct, IC Station can be used to automatically generate the VLSI layout for the chip. The user must instruct the
program to generate a floorplan for the chip, place the ports (pads) for the chip, place the standard cells from the library chosen in AutoLogic, and complete the routing for the chip. These steps are highly automated.
Chapter 4: IIR Band-Reject Filter Design Example

4.1 Filter Specifications and Applications

In this chapter, a band-reject filter that can block out 60 Hz frequencies will be designed so that the VLSI layout can be generated. The chosen sampling frequency is 3000 Hz. The parameters have been chosen such that a six-pole, six-zero filter will be required. The design specifications are summarized below:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Design Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling Frequency</td>
<td>3000 Hz</td>
</tr>
<tr>
<td>Lower Passband Frequency</td>
<td>50 Hz</td>
</tr>
<tr>
<td>Lower Stopband Frequency</td>
<td>58 Hz</td>
</tr>
<tr>
<td>Upper Stopband Frequency</td>
<td>62 Hz</td>
</tr>
<tr>
<td>Upper Passband Frequency</td>
<td>72 Hz</td>
</tr>
<tr>
<td>Maximum Passband Ripple</td>
<td>0.75 dB</td>
</tr>
<tr>
<td>Minimum Stopband Attenuation</td>
<td>36 dB</td>
</tr>
<tr>
<td>Input and Output Representations</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

Table 4.1 Filter Design Example Specifications

This filter presents some unique applications. In the United States, A/C power is delivered from utilities using 60 Hz waveforms. The 60 Hz frequencies that are the source of power in so many applications also cause noise within these systems. It is sometimes useful to block out the 60 Hz noise, especially if the application is concerned at looking at a signal that is not supposed to contain any 60 Hz components.
In most power-quality applications, harmonic distortion is considered to be undesirable. If the voltage signal for a residence is being monitored for power-quality, then one would desire that the waveform contain only a 60 Hz component ideally. There are ways to measure total harmonic distortion, but one application might be to feed the digitized voltage signal through a 60-Hz band reject filter, and whatever signal is left contains mostly the harmonic distortion which can be analyzed further.

4.2 DFL Design and High-Level Simulation

To begin initially, FILlab was used to get a general idea of the complexity of the filter that is required to meet these specifications. FILlab generated a DFL description, which was modified to contain the coefficients that will be derived next.

4.2.1 Derivation of Filter Coefficients

A complete derivation of the filter coefficients is presented in Appendix A, containing the MathCad printout of the design steps described in Chapter 2. The coefficients for the three biquad-sections were derived using the steps from Chapter 2. In addition, Appendix A describes some quantization issues which will also be discussed next.

After the biquad-section coefficients were derived, the effects of quantizing the coefficients needed to be analyzed. Quantizing the coefficients can severely affect the filter performance. In a digital filter, the coefficients need to be quantized, but as mentioned in Section 2.2.3 concerning the biquad-section filter design, slight variations in the coefficients can cause large variations in the pole locations, even with the
biquad-section approach. The gain of the filter with full precision was designed to be unity. However, after quantizing the coefficients, the gain of the filter is usually greater than one as a result of the change in pole locations of the filter. If the filter gain is too much greater than one, then it may be a requirement that the output contain more bits than the input. In this particular design, the input and output both will be represented by 8-bits, and thus, it is important to keep the filter gain less than or equal to one. In Appendix A, the filter was set up to meet the stopband attenuation and exceed the passband attenuation. This was done so that the gain could be adjusted by a factor slightly less than one. The lower gain adjustment would cause the stopband attenuation requirement to be exceeded while maintaining the other filter specifications.

The final goal was to use 18-bit quantized coefficients that meet the specifications and for the gain of the filter to be less than or equal to one. In an attempt to determine what the multiplication adjustment factor should be, the coefficients were first quantized to 20 bits (using truncation to the smallest integer), and then the DC gain was computed for the quantized filter. The 20-bit quantized filter was used as an estimate of the 18-bit quantized filter because some of the coefficients can be represented the same whether 18-bits or 20-bits are used. Thus, the intermediate filter is a closer approximation to what the 18-bit coefficients will achieve. The gain was then adjusted by a factor such that when multiplied by the 20-bit coefficient-filter, a DC gain of unity would be approximated. The 20-bit quantization was an intermediate step to help determine the factor to adjust the gain.
Note that the coefficients can be represented by \( n \) bits, where one bit is used for the sign bit, two bits for the numbers to the left of the decimal point, and \((n-3)\) bits for the actual fixed point representation to the right of the decimal point for the coefficients.

There are two sets of filter coefficients presented in Appendix A which meet the specifications. The first set of coefficients used the quantization method described above and will be used throughout the remainder of the design. However, a second, more simpler approach was also taken to obtain the quantized coefficients. The results were quite good, but the filter frequency response was a little less desirable than the filter achieved using the previous method.

This second method involves a single quantization step, without using the 20-bit intermediate quantization approach. Here, the original coefficients are quantized to 18-bits using a truncation method which results in smaller absolute values for the coefficients for both positive and negative coefficients. Previously, the coefficients were truncated with the "floor" function, which truncates the coefficient to the smallest integer. For negative coefficients, the smaller integer is the more negative of the two choices, and thus, the absolute value of the coefficient will be higher when the "floor" function is used with the negative coefficients. This second method will truncate such that the absolute value of the coefficient will always be less than or equal to the absolute value of the unquantized coefficient.

Upon simulation, this method created a filter whose gain was less than one for all frequencies less than 93.17 Hz. However, the gain for frequencies above 93.17 Hz rose to a peak of 0.00171 dB (magnitude gain of 1.0002) at 108.50 Hz and then
decreased exponentially, asymptotic to the positive side of the 0 dB line. This gain is so small that it is essentially no greater than one since when the maximum 8-bit number is multiplied by a factor such as 1.0002, then same 8-bit quantized value would result.

Thus, this second design also seemed to give acceptable results. However, since the results of the first approach achieved a digital filter meeting all the specifications with a gain less than one for all frequencies, the first method was chosen to be the filter to be implemented.

### 4.2.2 DFL Description of Filter

The DFL description of the band-stop IIR filter is given below. It describes the IIR filter in C-like syntax.

```c
#define IOW fix<8,0> /* Word Length of User I/O */
#define IW fix<20,3> /* Word Length of Input to Filter */
#define CW fix<18,15> /* Word Length of Filter Coefficients */
#define AW fix<20,3> /* Word Length of Accumulators */
```

---

44
Implement the three-biquad sections of the IIR filter.

```go
func main(IN : IOW) OUT : IOW =
```

/\* Definition of the coefficient arrays of the second-order sections \/
/\* METHOD 1 USED: 20-bit quantization first, then adjust by DC gain, then 10-bit quantization; floor for all \*\/

```go
C1[] = CW({
  1.000000000000000E+00,
  -1.984252929687500E+00,
  1.000000000000000E+00,
  1.951141357421875E+00,
  0.966613769531250E+00
});

C2[] = CW({
  1.000000000000000E+00,
  -1.984252929687500E+00,
  1.000000000000000E+00,
  1.97247802734375E+00,
  -0.985107421875000E+00
});

C3[] = CW({
  0.965820312500000E+00,
  -1.916473388671875E+00,
  0.965820312500000E+00,
  1.961547851562500E+00,
  -0.981231689453125E+00
});
```

/\* Description of the series of second-order sections \/
\*/

```go
Y1 = second_order_section(IN, C1[]);
Y2 = second_order_section(Y1, C2[]);
OUT = second_order_section(Y2, C3[]);
```

/\* Describe a single biquad section of an IIR filter. \*/

```go
func second_order_section(IN : IW; C : CW|5|) OUT : OW =
begin

Mn1 = AW(C[0] * A1);
Mn2 = AW(C[1] * A101);
Mn3 = AW(C[2] * A102);
Md1 = AW(C[3] * A101);
Md2 = AW(C[4] * A102);

OUT = Mn1 + Mn2 + Mn3;
A1 = Md1 + Md2 + IN;
```

The DFL above can be edited and compiled in Design Architect.

### 4.2.3 Final DFL Simulation Results

After the DFL is compiled, DSPlab can be used to simulate the DFL design at both a high-level and a bit-true level. To run the time-domain simulation for the filter, a waveform was created which consisted of two frequencies. The input waveform
contained a 60 Hz waveform with amplitude 32 added to a 40 Hz waveform of amplitude 96. The input waveform is shown below:

![Input (40 Hz sine wave + 60 Hz sine wave)]

Figure 4.1 Filter Input Test Signal

Using this input, the filter should block out the 60 Hz component, but allow the 40 Hz signal of amplitude 96 to pass through unchanged. The following waveform resulted:
The results verify the operation of the filter. Next, the frequency-domain simulation was performed, and the results are indicated below in Figures 4.3 and 4.4:

Figure 4.2 Filter Output Response To Test Signal

Figure 4.3 Filter Design Example Magnitude Response
The frequency response of the designed filter was acceptable. A summary of the results are given below in Table 4.2:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Expected Result</th>
<th>Simulated Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>End of Lower Passband</td>
<td>≥ 50 Hz</td>
<td>50.14 Hz</td>
</tr>
<tr>
<td>Beginning of Stopband</td>
<td>≤ 58 Hz</td>
<td>57.92 Hz</td>
</tr>
<tr>
<td>End of Stopband</td>
<td>≥ 62 Hz</td>
<td>62.00 Hz</td>
</tr>
<tr>
<td>Beginning of Upper Passband</td>
<td>≤ 72 Hz</td>
<td>71.62 Hz</td>
</tr>
</tbody>
</table>

Table 4.2 Filter Design Example Simulation Results

For this filter, the gain of the filter was less than one for all frequencies between DC and 1500 Hz. Note once again that the phase response of IIR filters is nonlinear, as illustrated in the phase response graph above.
4.3 VHDL Generation and Compilation

Next, Mistral1 is used to generate a bit-serial VHDL design from the DFL description. The simple DFL was expanded into structural VHDL code (a net list of the design) which is presented in Appendix B for comparison of the DFL complexity with the net list complexity. Note that VHDL is the input language of choice for AutoLogic. It is obvious that a high-level description, whether it is a DFL description or a behavioral VHDL description, is much simpler to write than the functional net list equivalent. The HDL compiler was used to compile the generated VHDL code.

The VHDL generated by Mistral1 has 7 inputs and 4 outputs. The pins can be summarized as follows:

<table>
<thead>
<tr>
<th>Pin Type</th>
<th>Pin Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>tc</td>
<td>Test control pin. When ‘1’, indicates to scan in the scanin_1, scanin_2, and scanin_3 data into the three scan chains and to scan out the results (Note that 1 bit at a time is scanned in and out for each clock cycle of phi). When ‘0’, indicates to operate normally using msl_in and msl_out. (Note that phi and blc_in are used to control the I/O timing).</td>
</tr>
<tr>
<td>Input</td>
<td>phi</td>
<td>System clock. This clock should be set to the sample rate multiplied by the wordlength, or 60 KHz (3 KHz * 20).</td>
</tr>
<tr>
<td>Input</td>
<td>msl_in</td>
<td>Filter input when tc = ‘0’. The input is scanned in LSB first, with one bit per clock cycle scanned in. Note that blc_in = ‘1’ for one clock cycle while the LSB of the input is read in. Then, blc_in will be low for the next 19 (wordlength - 1) clock cycles. Unused when tc = ‘1’.</td>
</tr>
<tr>
<td>Input</td>
<td>blc_in</td>
<td>Start of sample when tc = ‘0’. When blc_in = ‘1’, this indicates that the LSB of the input is ready to be read in. For the next 19 clock cycles, blc_in should equal ‘0’. For the first 7 of the 19 clock cycles that blc_in is low, the remainder of the input word will be read in one bit at a time. Unused when tc = ‘1’.</td>
</tr>
</tbody>
</table>
| Input    | scanin_1 | Bit to scan into scan chain #1 when tc = ‘1’. Scan chain #1
contains 195 cascading flip-flops. Unused when tc = ‘0’.

Bit to scan into scan chain #2 when tc = ‘1’. Scan chain #2 contains 195 cascading flip-flops. Unused when tc = ‘0’.

Bit to scan into scan chain #3 when tc = ‘1’. Scan chain #3 contains 189 cascading flip-flops. Unused when tc = ‘0’.

Output port when tc = ‘0’. The LSB of the output is valid 25 clock cycles after the input LSB is scanned in. Thus, under normal operation, there is a valid output starting every 5 clock cycles after the LSB of the input is scanned in (since the wordlength is 20). Unused when tc = ‘1’.

Bit that is scanned out of scan chain #1 when tc = ‘1’. Unused when tc = ‘0’.

Bit that is scanned out of scan chain #2 when tc = ‘1’. Unused when tc = ‘0’.

Bit that is scanned out of scan chain #3 when tc = ‘1’. Unused when tc = ‘0’.

| Input  | scanin_2  | Bit to scan into scan chain #2 when tc = ‘1’. Scan chain #2 contains 195 cascading flip-flops. Unused when tc = ‘0’.
|--------|-----------|
| Input  | scanin_3  | Bit to scan into scan chain #3 when tc = ‘1’. Scan chain #3 contains 189 cascading flip-flops. Unused when tc = ‘0’.
| Output | msl_out   | Output port when tc = ‘0’. The LSB of the output is valid 25 clock cycles after the input LSB is scanned in. Thus, under normal operation, there is a valid output starting every 5 clock cycles after the LSB of the input is scanned in (since the wordlength is 20). Unused when tc = ‘1’.
| Output | scanout_1 | Bit that is scanned out of scan chain #1 when tc = ‘1’. Unused when tc = ‘0’.
| Output | scanout_2 | Bit that is scanned out of scan chain #2 when tc = ‘1’. Unused when tc = ‘0’.
| Output | scanout_3 | Bit that is scanned out of scan chain #3 when tc = ‘1’. Unused when tc = ‘0’.

Table 4.3 I/O Pin Description for Filter Design Example IC

Using the Mistrall analysis tools, the user can determine the timing of the inputs and outputs for the bit-serial design. For the DFL design example, the internal wordlength of the digital filter is 20 bits. Thus, a new input can be scanned in every 20 clock cycles. Note that since the sampling frequency of the filter is 3 KHz, the clock phi must be chosen to be 20 times faster, or 60 KHz. The input and output are only 8 bits each, and the input and output are always scanned in and out least-significant bit (LSB) first. Thus, for the input pin, the input word is scanned in for 8 clock cycles, and then the input pin performs no operation for the next 12 clock cycles; thereafter, the cycle repeats. The output LSB is valid five clock cycles after the LSB of the input is scanned in. The output that comes out is caused by the previous input, however,
since the total delay is 25 clock cycles from the input to the output of the filter. This I/O timing is illustrated in Figure 4.5 below.

![Figure 4.5 Chip I/O Interfacing Timing Diagram](image)

Note that the hardware in this design is set up to constantly scan in inputs every 20 clock cycles and that the results from the previous input are still propagating through the hardware while the next input is being scanned in. The VHDL generated is somewhat cryptic and difficult to completely understand. However, the analysis tools that Mistral1 provides assist the user in determining the timing of the circuit. Specific examples which illustrate the timing in more detail were simulated using the QuickSim II tool. This will be the topic later in this chapter.

### 4.4 AutoLogic: Synthesizing VHDL -> Schematic

Using AutoLogic, a digital schematic was synthesized from the structural VHDL. This schematic detailed the VHDL connections between the standard cells
that will ultimately be used in the layout. AutoLogic generated a design viewpoint that could be loaded as a symbol into Design Architect and connected to the pads of the chip at the schematic level. The 1.2 micron CMOS worst-case standard cell library was specified in AutoLogic.

4.5 Design Architect: Refining Schematic

Design Architect is used to load the AutoLogic symbol for the VHDL and interface to the ports of the chip. Appendix C contains the schematic which links the filter symbol to the pads. This high-level representation will be used to instruct the IC Station package to automatically route the filter inputs and outputs to the pads of the chip.

4.6 DVE: Schematic Checks and ERC Validation

The Design Viewpoint Editor is used to check the schematic for errors. This includes an electrical rules check to verify that there are no electrical faults that will occur as a result of the schematic layout.

4.7 QuickSim II: VHDL Simulation/Verification

To verify that the circuit timing for scanning in and out data was interpreted correctly from the Mistral1 analysis tools, QuickSim II was used to provide a low-level simulation for 11 input words, with the results being analyzed. The input used to generate the 11 input words was the following function:

\[
\text{input}(n) = \sin(2\pi 60t) + \sin(2\pi 40t) \quad \text{for } t = nT \quad \text{for } n = 0,1,\ldots,10
\]
The sampling frequency was assumed to be 3000 Hz, so the sampling period $T$ is $1/3000$, or 333 microseconds. The following table of inputs were computed by hand and scaled by a multiplication factor of 50 so that the input could be quantized to integer 8-bit representations:

<table>
<thead>
<tr>
<th>$n$</th>
<th>input(n)</th>
<th>Scale by 50, truncate (-128 to 127)</th>
<th>Hex Input</th>
<th>8-bit Binary Input (MSB first)</th>
<th>8-bit Binary Input (LSB first)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>0.2090</td>
<td>10</td>
<td>A</td>
<td>00001010</td>
<td>01010000</td>
</tr>
<tr>
<td>2</td>
<td>0.4155</td>
<td>20</td>
<td>14</td>
<td>00010100</td>
<td>00101000</td>
</tr>
<tr>
<td>3</td>
<td>0.6168</td>
<td>30</td>
<td>1E</td>
<td>00011110</td>
<td>01110000</td>
</tr>
<tr>
<td>4</td>
<td>0.8315</td>
<td>41</td>
<td>29</td>
<td>00101001</td>
<td>10010100</td>
</tr>
<tr>
<td>5</td>
<td>0.9945</td>
<td>49</td>
<td>31</td>
<td>00110001</td>
<td>10001100</td>
</tr>
<tr>
<td>6</td>
<td>1.1663</td>
<td>58</td>
<td>3A</td>
<td>00111010</td>
<td>01011100</td>
</tr>
<tr>
<td>7</td>
<td>1.3239</td>
<td>66</td>
<td>42</td>
<td>0.000010</td>
<td>01000010</td>
</tr>
<tr>
<td>8</td>
<td>1.4655</td>
<td>73</td>
<td>49</td>
<td>01001001</td>
<td>10010010</td>
</tr>
<tr>
<td>9</td>
<td>1.5894</td>
<td>79</td>
<td>4F</td>
<td>01001111</td>
<td>11110010</td>
</tr>
<tr>
<td>10</td>
<td>1.6942</td>
<td>84</td>
<td>54</td>
<td>01010100</td>
<td>00101010</td>
</tr>
</tbody>
</table>

Table 4.4 Bit-Serial Inputs For Digital Simulation

The filter scan chains were tested before the input was applied. This was also done to initialize the state of the filter chip by clearing all the flip-flops. First, all three scan chains were loaded with ‘1’ s, then, the scan chains were loaded with ‘0’ s, and finally, the 11-input test was performed. The input signals were simulated as follows:
**phi:** An active high clock (≈) with a 50% duty cycle and period 16.667 μs (approx. 60 KHz). Note that the simulator time units are nanoseconds, so the clock was set to 16667 ns, and the timing of all input signals have been adjusted using a 16667 ns clock as opposed to a 16.667 μs clock.

**tc:** ‘1’ for 400 clock cycles (6666.800 μs), then ‘0’ forever. Note that first, the scan chains will be loaded with ‘1’s for 200 clock cycles, and then the scan chains will be loaded with ‘0’s for the next 200 clock cycles. Note also that the maximum scan chain length is 195, so after 195 cycles with the scanin_x signals held constant, all of the scan chain flip-flops have been initialized to their respective scanin_x constants.

**scanin_1, scanin_2, scanin_3:** ‘1’ for 200 clock cycles (3333.400 μs), then ‘0’ forever.

**blc_in:** A clock signal with period 333.340 μs (20 clock cycles of phi, or approx. 3000 Hz, the sampling frequency). The blc_in signal is ‘1’ for the first 16.667 μs (1 clock cycles of phi), then ‘0’ for the remainder of the clock period (19 clock cycles of phi).

**ms1_in:** The input signal which scans in the data after 400 clock cycles of testing and preloading the scan chains. To simulate the input words, it was necessary to force the input to change states at specified times. The LSB of the input was set properly at the same time that the blc_in signal was set high. Both signals remained constant for 1 clock cycle of phi. Then, every clock cycle of phi, the next input was updated (scanned in) during the falling edge of the clock and held at a constant state for 1 entire clock cycle of phi. The forces are presented in a table on the following page as an illustration of how tedious low-level filter simulations with bit-serial designs can be.
<table>
<thead>
<tr>
<th>msl_in</th>
<th>Time (μs)</th>
<th>msl_in</th>
<th>Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8716.841</td>
</tr>
<tr>
<td>1</td>
<td>7016.807</td>
<td>0</td>
<td>8766.842</td>
</tr>
<tr>
<td>0</td>
<td>7033.474</td>
<td>1</td>
<td>9016.847</td>
</tr>
<tr>
<td>1</td>
<td>7050.141</td>
<td>0</td>
<td>9033.514</td>
</tr>
<tr>
<td>0</td>
<td>7066.808</td>
<td>1</td>
<td>9100.182</td>
</tr>
<tr>
<td>1</td>
<td>7366.814</td>
<td>0</td>
<td>9116.849</td>
</tr>
<tr>
<td>0</td>
<td>7383.481</td>
<td>1</td>
<td>9333.520</td>
</tr>
<tr>
<td>1</td>
<td>7400.148</td>
<td>0</td>
<td>9350.187</td>
</tr>
<tr>
<td>0</td>
<td>7416.815</td>
<td>1</td>
<td>9383.521</td>
</tr>
<tr>
<td>1</td>
<td>7683.487</td>
<td>0</td>
<td>9400.188</td>
</tr>
<tr>
<td>0</td>
<td>7750.155</td>
<td>1</td>
<td>9433.522</td>
</tr>
<tr>
<td>1</td>
<td>8000.160</td>
<td>0</td>
<td>9450.189</td>
</tr>
<tr>
<td>0</td>
<td>8016.827</td>
<td>1</td>
<td>9666.860</td>
</tr>
<tr>
<td>1</td>
<td>8050.161</td>
<td>0</td>
<td>9733.528</td>
</tr>
<tr>
<td>0</td>
<td>8066.828</td>
<td>1</td>
<td>9766.862</td>
</tr>
<tr>
<td>1</td>
<td>8083.495</td>
<td>0</td>
<td>9783.529</td>
</tr>
<tr>
<td>0</td>
<td>8100.162</td>
<td>1</td>
<td>10033.534</td>
</tr>
<tr>
<td>1</td>
<td>8333.500</td>
<td>0</td>
<td>10050.201</td>
</tr>
<tr>
<td>0</td>
<td>8350.167</td>
<td>1</td>
<td>10066.868</td>
</tr>
<tr>
<td>1</td>
<td>8400.168</td>
<td>0</td>
<td>10083.535</td>
</tr>
<tr>
<td>0</td>
<td>8433.502</td>
<td>1</td>
<td>10100.202</td>
</tr>
<tr>
<td>1</td>
<td>8683.507</td>
<td>0</td>
<td>10116.869</td>
</tr>
<tr>
<td>0</td>
<td>8700.174</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5 msl_in Force Statements For Digital Simulation
The answers for the current input become available 25 clock cycles after the input. On the rising clock edge (previous to scanning out the data), the result msl_out is latched, with a simulated delay of 0.4 ns before the output is valid. The user, however, will not read msl_out until the falling edge, after it has had 8333.5 ns to settle.

To summarize, the user must change the inputs and read the outputs during the falling edge of the clock phi, and the filter chip will latch the inputs and outputs during the rising edge. Since the clock period is very long relative to the propagation delay within the filter chip, the I/O timing should be very reliable.

The filter results for the 11 integer inputs were estimated using a C-program. The program used double precision floating-point operations (64-bit representations).

The C-program used to simulate the filter results is given below:

```c
#define NUM 11 /* Number of inputs and outputs to monitor */

double adln[NUM] = { 0.0, 10.0, 20.0, 30.0, 41.0, 49.0, 58.0, 66.0, 73.0, 79.0, 84.0 }; /* List of inputs */
double adYl[NUM]; /* List of outputs from first biquad-section */
double adY2[NUM]; /* List of outputs from second biquad-section */
double adOut[NUM]; /* List of outputs from third biquad-section */
double adC1[5] = { 1.0, 
   -1.9842529296875, 
   1.0, 
   1.951141357421875, 
   -0.9661376953125 }; /* Coefficients for Biquad 1 */

double adC2[5] = { 1.0, 
   -1.9842529296875, 
   1.0, 
   1.972747802734375, 
   -0.985107421875 }; /* Coefficients for Biquad 2 */

double adC3[5] = { 0.9658203125, 
   -1.916473388671875, 
   0.9658203125, 
   1.9615478515625, 
   -0.981231689453125 }; /* Coefficients for Biquad 3 */
```
/ * Biquad-section processing for entire array of inputs */
void biquad_section(double *in, double *coeff, double *out)
{
    int k;
    double A1[NUM+2];
    double Mdl[NUM];
    double Md2[NUM];
    double Mnl[NUM];
    double Mn2[NUM];
    double Mn3[NUM];

    /* Initialize previous inputs to zero */
    A1[0] = 0.0;
    A1[1] = 0.0;

    /* Process the inputs to the biquad section */
    for (k=0; k<NUM; k++)
    {
        Mdl[k] = coeff[3] * A1[k+1];
        A1[k+2] = Mdl[k] + Md2[k] + in[k];
        Mnl[k] = coeff[0] * A1[k+2];
        Mn2[k] = coeff[1] * A1[k+1];
        Mn3[k] = coeff[2] * A1[k];
        out[k] = Mnl[k] + Mn2[k] + Mn3[k];
    }
}

int main()
{
    int i;
    /* Set up the IIR filter as three biquad-sections in series */
    biquad_section( adln, adCl, adY1 );
    biquad_section( adY1, adC2, adY2 );
    biquad_section( adY2, adC3, adOut );

    printf("The simulation results for the IIR filter are as follows:\n");

    /* Print out the results from the IIR filter */
    for (i=0; i<NUM; i++)
    {
        printf("i: %d; input: %d; output: %d\n",
                i, (int)(adIn[i]), (int)(adOut[i]) );
    }
    return 1;
}

The expected answers to the 11 inputs are compared with the simulated results below.

The time indicated refers to the time that the LSB of the result was valid.
The simulation showed that using 20-bit internal accumulators and 8-bit truncated results yielded the same value as the expected results that were accumulated with 64-bit floating point accumulators, with all results within 1 count for the 11 consecutive inputs simulated. This simulation was meant to prove that the timing of the inputs and outputs has been interpreted correctly. Given that the simulation results coincide with the expected results, the Mistral1 analysis tools must have been interpreted correctly.

Upon successful completion of the QuickSim II testing, a Mentor Graphics tool referred to as QuickPath was used to analyze the approximate delay times involved for the output signals to propagate from the filter layout to the chip pads.
The worst-case delay from the output of the VLSI layout to the chip pin output was computed to be 5.56 ns for a falling edge. The results for all the output delays are presented below in order of descending delays:

<table>
<thead>
<tr>
<th>Type of Edge</th>
<th>Output Pin</th>
<th>Delay From Silicon to Pad (in ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Falling edge</td>
<td>scanout_3_pad</td>
<td>5.56</td>
</tr>
<tr>
<td>Rising edge</td>
<td>scanout_3_pad</td>
<td>5.41</td>
</tr>
<tr>
<td>Falling edge</td>
<td>scanout_2_pad</td>
<td>4.32</td>
</tr>
<tr>
<td>Falling edge</td>
<td>scanout_1_pad</td>
<td>4.22</td>
</tr>
<tr>
<td>Rising edge</td>
<td>scanout_2_pad</td>
<td>4.18</td>
</tr>
<tr>
<td>Rising edge</td>
<td>scanout_1_pad</td>
<td>4.07</td>
</tr>
<tr>
<td>Falling edge</td>
<td>msl_out_pad</td>
<td>3.49</td>
</tr>
<tr>
<td>Rising edge</td>
<td>msl_out_pad</td>
<td>3.32</td>
</tr>
</tbody>
</table>

Table 4.7 Chip Output Delays From Silicon to Pin

The speed of the chip is fast relative to the sampling frequency of 3000 Hz. Delays can easily be compensated for since the chip controller latches its I/O on the rising clock edges and the user latches his I/O on the falling clock edges. This allows for 8333.5 ns settling time between the user’s I/O and the chip controller’s I/O, which is more than enough time for the signals to propagate through to the outputs.

4.8 IC Station: Schematic -> VLSI Layout

After the schematic simulation was verified, IC Station was used to perform the VLSI layout process. N-well 1.2 micron CMOS technology with two metal layers and one polysilicon layer was used to implement the filter layout. Using the 1.2
micron CMOSN standard cell library, the layout that was generated is 7.315 mm x 7.213 mm. The layout was generated in accordance with the CMOSN rules file.

Taking a small area that is representative of the chip, the number of transistors in the design was estimated. An area 0.552 mm by 0.353 mm was examined and contained 154 transistors. The dimensions of the chip containing transistors of approximately the same density was estimated to be 5.412 mm by 6.008 mm. Assuming that the transistors per unit area is held constant for the entire area of the chip, the estimated number of transistors in the layout is 25,700.

4.9 Testing the Filter ASIC Chip

After fabrication, the chip may be tested using a digital hardware design that can control the timing for the I/O ports to the chip. The timing of the digital design must reflect the timing that was discussed when simulating the chip using QuickSim II. It is important to note that this application was meant to be an ASIC design, that is, an application-specific integrated circuit. Normally, this type of design would occur as a single component within a larger VLSI design. Thus, this design was not meant to solve the world’s 60-Hz noise problem, but it was meant to illustrate the power that automation can provide in VLSI designs.

In general, the easiest method for analyzing a filter’s performance is to connect the filter input to a function generator and the filter output to an oscilloscope. The user can sweep through different input frequencies and analyze the attenuation on the output signal. For the band-reject filter, the output should be close to zero for input
frequencies in the stopband region near 60 Hz. At lower and higher frequencies (not greater than 1500 Hz, the Nyquist rate, to prevent aliasing) of the passband, the output should be approximately the same size signal as the input.

Since this is a filter chip with a digital interface, the function generator must be connected to an A/D (analog-to-digital) converter with the digital A/D output connected to the filter chip. Similarly, the output of the chip must be connected to a D/A (digital-to-analog) converter with the analog D/A output connected to the oscilloscope. The filter chip is a bit-serial design, with the least-significant bit always shifted in first. Thus, the simplest test fixture design would use an 8-bit serial A/D and 8-bit serial D/A. Recall that the chip controller latches its I/O on rising clock edges and the user latches his I/O on falling clock edges. The timing of the test fixture must be set up properly to enforce proper communication with the filter chip as shown in the timing simulation using QuickSim II. The test fixture must also recognize that the DFL design was implemented for signed inputs and signed outputs.
Chapter 5: Conclusions

Automation is not yet to the point where it applies to every type of design, but the DSP Station tools provided by Mentor Graphics are a big step forward in design automation. VHDL has been recognized in the design automation field as the high-level replacement for the layout of the VLSI chip. The DSP Station tools provide a similar type of automation, by automatically generating a structured VHDL net list from a high-level DFL description. The DSP Station tools additionally provide a means to simulate high-level DSP algorithms that often require sinusoidal inputs. This high-level graphical simulation is very useful in determining the quantization effects by providing the bit-true simulation option.

There are certain limitations in the current tools that do not allow for complex designs. Mistral 1 is designed to be used for creating filter designs where the filter functionality is known and where the coefficients can be hardwired in without a problem. Thus, the current application for these tools is for use in a static design where filter programmability is not a requirement. Signal initializations are ignored in Mistral 1, and thus, for the IIR bit-serial filter, the individual scan-chains must be preloaded with initial zero values. This is required since an IIR filter uses feedback from the output, which will be in an unknown state on power-up. Although there are limitations present in the DSP Station tools, this is a big step in the field of design automation. The tools provide for simplifying the VLSI design process, and with this
simplification, it is possible to create certain designs faster than with conventional methods.
References


Appendix A: MathCad Derivation of Filter Coefficients

The following analysis will be taken from Fundamentals of Digital Signal Processing, by Lonnie C. Ludeman, and previous knowledge of filter design. Using Mathcad, an infinite impulse response (IIR) band-reject filter of order 6 will be designed. The quadratic formula will be designed below as a matrix, and j will be defined to indicate imaginary numbers.

\[
\text{quad}(a, b, c) := \begin{bmatrix}
-b + \sqrt{b^2 - 4 \cdot a \cdot c} \\
2 \cdot a \\
b - \sqrt{b^2 - 4 \cdot a \cdot c} \\
2 \cdot a
\end{bmatrix}
\]

\[j := \sqrt{-1}\]

\[s := 0\]

These definitions for s and z are used as a matter of convenience in Mathcad so there are no undefined variables.

Next, the sampling frequency and the band-reject filter parameters will be set up. The band-reject filter will have both a passband (the frequencies for which the filter will pass the input) and a stopband (the frequencies for which the filter will NOT pass the input). Ideally, signals in the passband should be passed through with a gain of 1. However, some small range of error must be accepted. This error is called the passband ripple. For the stopband, the ideal case is for no signals of these frequencies to be passed through, thus having a gain of 0. Realistically, one must settle for a minimum stopband attenuation.

The band-reject filter has four distinct frequencies which must be specified. These frequencies specify the passband and stopband frequencies. Note that a perfect filter would have the stopband begin at the exact point where the passband ends, and vice-versa. In reality, some room must be left between where the passband ends and the stopband starts, and vice-versa. If a very strict filter is required, then the order of the filter will be higher, resulting in a more complex design. Below, the filter parameters will be set up, and it will be shown that the filter can be designed with order 6.

\[F_s := 3000\]  (Sampling frequency) \[T := \frac{1}{F_s}\]  (Sampling period)

Filter frequency parameters:

\[f_{\text{lower}} := 50\]  (lower passband frequency - where passband ends)

\[f_1 := 58\]  (lower stopband frequency - where stopband begins)

\[f_2 := 62\]  (upper stopband frequency - where stopband ends)

\[f_{\text{upper}} := 72\]  (upper passband frequency - where passband begins again)

Filter passband and stopband attenuations

\[\alpha := 0.75\]  (maximum ripple in the passband)

\[\beta := 36\]  (minimum attenuation in the stopband)
To design our digital filter, the Bilinear Z-Transform will be used, which transforms an analog filter into a digital filter. Given the filter parameters, the order of the analog filter needed to meet the specifications must be computed. To begin, the frequency parameters will be converted from Hertz to rad/sec.

\[
\begin{align*}
\omega_{\text{lower}} & := 2\pi f_{\text{lower}} \quad \omega_{\text{lower}} = 314.1592653589793 \\
\omega_{1} & := 2\pi f_{1} \quad \omega_{1} = 364.424747816416 \\
\omega_{2} & := 2\pi f_{2} \quad \omega_{2} = 389.5574890451343 \\
\omega_{\text{upper}} & := 2\pi f_{\text{upper}} \quad \omega_{\text{upper}} = 452.3893421169302
\end{align*}
\]

To design a band-reject analog filter, it is sometimes easier to work backwards to find the parameters for a simple analog low-pass filter which can be transformed into a more complicated band-reject filter. Thus, a low-pass counterpart will be designed having a passband that ends at 1 rad/sec and a stopband beginning as calculated below.

\[
\begin{align*}
\omega_{r} := \min \left[ \begin{array}{c}
\frac{\omega_{1} \left( \omega_{\text{upper}} - \omega_{\text{lower}} \right)}{-\omega_{2}^{2} + \omega_{\text{lower}} \cdot \omega_{\text{upper}}} \\
\frac{\omega_{2} \left( \omega_{\text{upper}} - \omega_{\text{lower}} \right)}{-\omega_{2}^{2} + \omega_{\text{lower}} \cdot \omega_{\text{upper}}} 
\end{array} \right] \\
\omega_{r} = 5.406779661016958
\end{align*}
\]

\[
\begin{align*}
n_{\text{real}} := \log \left[ \frac{a}{10^{10} - 1} \right] \\
n_{\text{real}} := \log \left[ \frac{b}{10^{10} - 1} \right] \\
n := \text{ceil} \left( n_{\text{real}} \right) \quad n = 3
\end{align*}
\]

Thus, a low-pass filter of order 3 is required. When transforming the low-pass filter to a band-reject filter, the order of the filter will double. Thus, a band-reject filter that will be of order 6 has been found.

Next, the filter frequencies for the band-reject filter will be prewarped to correct for the nonlinearities of the Bilinear Z-Transform.
\[ \Omega_{\text{lower}} := \frac{2}{T} \tan \left( \frac{2 \cdot \pi f_{\text{lower}} T}{2} \right) \quad \Omega_{\text{lower}} = 314.4466756982472 \]

\[ \Omega_{1} := \frac{2}{T} \tan \left( \frac{2 \cdot \pi f_{1} T}{2} \right) \quad \Omega_{1} = 364.8735357446796 \]

\[ \Omega_{2} := \frac{2}{T} \tan \left( \frac{2 \cdot \pi f_{2} T}{2} \right) \quad \Omega_{2} = 390.1057961127768 \]

\[ \Omega_{\text{upper}} := \frac{2}{T} \tan \left( \frac{2 \cdot \pi f_{\text{upper}} T}{2} \right) \quad \Omega_{\text{upper}} = 453.2485575262613 \]

To satisfy our requirement of the passband attenuation and exceed the minimum stopband attenuation, the following is used:

\[ \Omega_{c1} := \frac{1}{\left( \frac{\alpha}{10^{10} - 1} \right)^{2 \cdot n}} \quad \text{The 1 in the numerator is the end of the passband for the normalized low-pass filter that the denormalized band-reject filter will be derived from. This indicates that the end of the passband was at 1 rad/sec.} \]

\[ \Omega_{c1} = 1.32062825736801 \]

However, if one had wanted to meet the minimum stopband attenuation exactly and exceed the passband attenuation requirement, then the following could have been used instead:

\[ \Omega_{c2} := \frac{\omega_{r}}{\left( \frac{\beta}{10^{10} - 1} \right)^{2 \cdot n}} \quad \Omega_{c2} = 1.358178512655312 \]

To exceed both requirements, one could choose a value in between the values calculated above. Since a digital filter is being created that requires a gain less than or equal to one (so that the same number of bits can be used for the output as for the input), we will meet the minimum stopband attenuation and exceed the passband attenuation. In one of the final steps, the gain will be adjusted to guarantee that at all frequencies, unity gain is not exceeded.

\[ \Omega_{c} := \Omega_{c2} \quad \Omega_{c} = 1.358178512655312 \]
The poles of a filter must be in the left-half of the s-plane for the filter to be stable. The angular spacing between the poles is 180 degrees divided by the filter order n. Thus, for a filter of order 3, the poles are separated by 60 degrees. The first pole is at s=-1, while the other two poles are at a radius one from the origin at ±60 degrees from the NEGATIVE s-axis. Thus:

\[
\begin{align*}
    s_1 &= -1 \\
    s_2 &= \frac{-1}{2} + \frac{\sqrt{3}}{2} \cdot j \\
    s_3 &= \frac{-1}{2} - \frac{\sqrt{3}}{2} \cdot j
\end{align*}
\]

The general equation for a normalized Butterworth low-pass filter of order 3 is shown below:

\[
(1) \quad H_a(s) := \frac{1}{(s - s_1)(s - s_2)(s - s_3)}
\]

To transform the normalized low-pass analog filter to one which can be used to meet the desired attenuation requirements, the following intermediate transformation must take place:

\[
(2) \quad s := \frac{s}{\Omega_c}
\]

That transformation is known as a low-pass to low-pass transformation. Next, a low-pass to band-reject transformation will be done as shown:

\[
(3) \quad s := \frac{s \cdot (\Omega_{\text{upper}} - \Omega_{\text{lower}})}{s^2 + \Omega_{\text{lower}} \cdot \Omega_{\text{upper}}}
\]

Combining transformations (2) and (3), transformation (4) is obtained which can represent both:

\[
(4) \quad s := \frac{s \cdot (\Omega_{\text{upper}} - \Omega_{\text{lower}})}{s^2 + \Omega_{\text{lower}} \cdot \Omega_{\text{upper}}}
\]

For simplicity in the equations that follow, some variables will be created:

\[
(5) \quad c_1 := \frac{\Omega_{\text{upper}} - \Omega_{\text{lower}}}{\Omega_c} \quad c_1 = 102.1970827359424
\]

\[
(6) \quad c_2 := \Omega_{\text{lower}} \cdot \Omega_{\text{upper}} \quad c_2 = 142522.5021791586
\]

A-4
The Bilinear Z-Transform is used to convert the transfer function in the s-domain to a transfer function in the z-domain. This is where it becomes important to use the frequencies which have been prewarped to correct for the nonlinearities in the Bilinear Z-Transform technique. The BZT transformation is given by:

\[
(7) \quad s := \frac{2}{T} \cdot \left( \frac{z - 1}{z + 1} \right)
\]

For simplicity, a variable will be set up to represent \(2/T\) also.

\[
(8) \quad F := \frac{2}{T} \quad F = 6000
\]

Thus, by combining equations (4), (5), (6), (7), and (8), one transformation can be created which will convert equation (1) into the desired z-domain transfer function \(H(z)\) for the band-reject filter. The transformation is given below:

\[
(9) \quad s := \frac{\frac{z - 1}{z + 1} \cdot F \cdot c_1}{\left(\frac{z - 1}{z + 1} \cdot F\right)^2 + c_2}
\]

This can be rearranged as such:

\[
(10) \quad s := \frac{(z - 1) \cdot (z + 1) \cdot F \cdot c_1}{(z - 1)^2 \cdot F^2 + (z + 1) \cdot c_2}
\]

Substituting (10) into (1) by replacing each "s" in (1) by its equivalent from (10) will yield the z-domain transfer function \(H(z)\) for the desired band-reject filter. After simplification, the following equations are obtained:

\[
n_z := 2 \cdot F^2 - 2 \cdot c_2
\]

\[
n_{\text{const}} := \begin{bmatrix}
-F^2 + \frac{c_1}{s_1} \cdot F - c_2 \\
-F^2 + \frac{c_1}{s_2} \cdot F - c_2 \\
-F^2 + \frac{c_1}{s_3} \cdot F - c_2
\end{bmatrix}
\]

\[
d_{\text{const}} := \begin{bmatrix}
-F^2 + \frac{c_1}{s_1} \cdot F - c_2 \\
-F^2 + \frac{c_1}{s_2} \cdot F - c_2 \\
-F^2 + \frac{c_1}{s_3} \cdot F - c_2
\end{bmatrix}
\]
\[
\text{gain} := \frac{\left( F^2 + c_2 \right)^3}{s \cdot 1 \cdot s \cdot 2 \cdot s \cdot 3 \cdot d \cdot \text{const}_0 \cdot d \cdot \text{const}_1 \cdot d \cdot \text{const}_2}
\]

\[
gain = 0.966639423963699
\]

\[
a := \begin{pmatrix}
1 & -\frac{n_z}{F^2 + c_2} & 1
\end{pmatrix}
\]

\[
b := \begin{pmatrix}
1 & \frac{n_z}{d \cdot \text{const}_0} & \frac{n \cdot \text{const}_0}{d \cdot \text{const}_0} \\
1 & \frac{n_z}{d \cdot \text{const}_1} & \frac{n \cdot \text{const}_1}{d \cdot \text{const}_1} \\
1 & \frac{n_z}{d \cdot \text{const}_2} & \frac{n \cdot \text{const}_2}{d \cdot \text{const}_2}
\end{pmatrix}
\]

Thus, the transfer function \( H(z) \) can be written as:

\[
H(z) := \text{gain} \cdot \frac{\left( a_{0,0} \cdot z^2 + a_{0,1} \cdot z + a_{0,2} \right)^3}{\left( z^2 + b_{0,1} \cdot z + b_{0,2} \right) \cdot \left( z^2 + b_{1,1} \cdot z + b_{1,2} \right) \cdot \left( z^2 + b_{2,1} \cdot z + b_{2,2} \right)}
\]

\[
gain = 0.966639423963699
\]

\[
a_{0,0} = 1
\]

\[
a_{0,1} = -1.984226612608949
\]

\[
a_{0,2} = 1
\]

\[
b_{0,1} = -1.951124458050346
\]

\[
b_{0,2} = 0.96663470356838
\]

\[
b_{1,1} = -1.967118776818208 + 0.028659195284552j
\]

\[
b_{1,2} = 0.982756167383274 - 0.028887018350056j
\]

\[
b_{2,1} = -1.967118776818208 - 0.028659195284552j
\]

\[
b_{2,2} = 0.982756167383274 + 0.028887018350056j
\]
Next, the poles and zeros of the transfer function can be solved for:

\[
\text{zeroes} := \begin{bmatrix}
\text{quad}(a_{0,0}, a_{0,1}, a_{0,2})_0 \\
\text{quad}(a_{0,0}, a_{0,1}, a_{0,2})_1 \\
\text{quad}(a_{0,0}, a_{0,1}, a_{0,2})_0 \\
\text{quad}(a_{0,0}, a_{0,1}, a_{0,2})_1 \\
\text{quad}(a_{0,0}, a_{0,1}, a_{0,2})_0 \\
\text{quad}(a_{0,0}, a_{0,1}, a_{0,2})_1 \\
\end{bmatrix}
\]

\[
\text{poles} := \begin{bmatrix}
\text{quad}(b_{0,0}, b_{0,1}, b_{0,2})_0 \\
\text{quad}(b_{0,0}, b_{0,1}, b_{0,2})_1 \\
\text{quad}(b_{1,0}, b_{1,1}, b_{1,2})_0 \\
\text{quad}(b_{1,0}, b_{1,1}, b_{1,2})_1 \\
\text{quad}(b_{2,0}, b_{2,1}, b_{2,2})_0 \\
\text{quad}(b_{2,0}, b_{2,1}, b_{2,2})_1 \\
\end{bmatrix}
\]

Solving for the 6 zeroes and the 6 poles yields

\[
\text{zeroes} = \begin{bmatrix}
0.992113306304475 + 0.125344275711353j \\
0.992113306304475 - 0.125344275711353j \\
0.992113306304475 + 0.125344275711353j \\
0.992113306304475 - 0.125344275711353j \\
0.992113306304475 + 0.125344275711353j \\
0.992113306304475 - 0.125344275711353j \\
\end{bmatrix}
\]

\[
\text{poles} = \begin{bmatrix}
0.975562229025173 + 0.122118961950289j \\
0.975562229025173 - 0.122118961950289j \\
0.986359389766498 + 0.11049137107602j \\
0.98075938705171 - 0.139150566360573j \\
0.986359389766498 - 0.11049137107602j \\
0.98075938705171 + 0.139150566360573j \\
\end{bmatrix}
\]

Finally, the resulting transfer function \( \text{H}(z) \) can be graphed to verify that the design was successful.

\[
G(\omega) := H[e^{(j\omega \cdot T)}] \quad \omega := 2\cdot\pi \cdot 40 \ldots 2\cdot\pi \cdot 80
\]

\[
\text{20-log}(|G(\omega)|) \quad 4045 \quad 50 \quad 55 \quad 60 \quad 65 \quad 70 \quad 75 \quad 80
\]

\[
\frac{\omega}{2\cdot\pi}
\]

A-7
Now that the poles and zeroes have been computed, the next step is to create the transfer function $H(z)$ using coefficients which represent real numbers. This will occur by pairing the complex conjugate poles and zeroes together into 3 biquad sections. Each biquad section will use 2 poles and 2 zeroes as shown next. Let us first look at a general case of a single biquad section.

The first biquad section will be used as an example:

\[
\begin{align*}
  z_1 & := \text{zeroes}_0 \quad \text{(zeroes for first biquad section)} \\
  z_2 & := \text{zeroes}_1 \\
  p_1 & := \text{poles}_0 \quad \text{(poles for first biquad section)} \\
  p_2 & := \text{poles}_1
\end{align*}
\]

\[
H_{\text{biquad}}(z) := \frac{(z - z_1)(z - z_2)}{(z - p_1)(z - p_2)} \quad \text{(Transfer function for biquad section)}
\]

(11) \[H_{\text{biquad}}(z) := \frac{z^2 - (z_1 + z_2)z + z_1z_2}{z^2 - (p_1 + p_2)z + p_1p_2}\]

Since the paired zeroes are complex conjugates and likewise for the paired poles, the following simplification can be made:

(12) \[H_{\text{biquad}}(z) := \frac{z^2 - 2\Re(z_1)z + \left|z_1\right|^2}{z^2 - 2\Re(p_1)z + \left|p_1\right|^2}\]

Either equation (11) or (12) can be used to solve for the coefficients for the transfer function of the biquad sections. Note that in equation (12), either pole of the complex conjugate pair may be used to solve for the coefficients and likewise for the zeroes. Equation (12) will be used below to solve for the coefficients for the 3 biquad sections for the bandstop filter which has been designed thus far.
Thus, the transfer function $H(z)$ for the digital filter contains 3 biquad sections with real coefficients. The three biquad sections are defined as:

$$H_1(z) := \frac{\text{coeff}_{a0} \cdot z^2 + \text{coeff}_{a1} \cdot z + \text{coeff}_{a2}}{z^2 + \text{coeff}_{b0} \cdot z + \text{coeff}_{b1}}$$

$$H_2(z) := \frac{\text{coeff}_{a3} \cdot z^2 + \text{coeff}_{a4} \cdot z + \text{coeff}_{a5}}{z^2 + \text{coeff}_{b2} \cdot z + \text{coeff}_{b3}}$$

$$H_3(z) := \frac{\text{coeff}_{a6} \cdot z^2 + \text{coeff}_{a7} \cdot z + \text{coeff}_{a8}}{z^2 + \text{coeff}_{b4} \cdot z + \text{coeff}_{b5}}$$

The transfer function $H(z)$ is simply the product of the three biquad-sections, adjusted by the gain:

$$H(z) := (\text{gain}) \cdot H_1(z) \cdot H_2(z) \cdot H_3(z)$$
Once again, the resulting transfer function may be plotted to verify the correctness of the coefficient design.

\[ G(\omega) := H\left[ e^{j\omega T} \right] \]

\[ \omega := 2\pi 40..2\pi 80 \]

\[ 20\log(|G(\omega)|) \]

\[ \frac{\omega}{2\pi} \]

The passband ripple was specified to be within a certain dB level, specifically 0.75 dB. For a digital filter, the gain should be less than or equal to one if the same number of bits is to be used in the output as in the input. As a result of quantizing the coefficients, the gain may vary slightly. To guarantee that the gain of the filter does not exceed unity, the gain multiplier may need to be decreased slightly.

To simplify the discussion, the entire gain will be applied to the final biquad section.

In a digital filter, the coefficients need to be quantized. The gain of the filter with full precision is set to unity. However, quantizing the coefficients may cause the gain to exceed one. Recall that originally, the filter was designed such that it met the stopband attenuation and exceeded the passband attenuation. This was done intentionally so that the gain could be adjusted by a factor slightly less than one while maintaining the original filter specifications. In an attempt to determine what the factor should be, the coefficients will first be quantized to 20 bits (using truncation to the smallest integer), and then the DC gain will be computed for the quantized filter. The gain will be adjusted by a factor such that when multiplied by the 20-bit coefficient filter, a DC gain of unity will be achieved. The final goal is to use 18-bit quantized coefficients that meet the specifications. The 20-bit quantization will be an intermediate step to help determine the factor to adjust the gain with. Note that coefficients can be represented by \( n \) bits, where 1 bit will be used for the sign bit, 2 bits for the numbers to the left of the decimal point, and \((n-3)\) bits for the actual fixed point representation after the decimal point for the coefficients.

\[ n := 20 \quad b := n - 3 \quad b = 17 \]

\[ \text{num\_bits\_per\_unit} := 2^b \]

\[ \text{num\_units\_per\_bit} := 2^{-b} \]

\[ \text{quantize}(x) := \text{floor}(x \cdot \text{num\_bits\_per\_unit}) \cdot \text{num\_units\_per\_bit} \]
\[
\begin{align*}
\text{coeff}_{\text{aq}} & := \\
& \begin{bmatrix}
\text{quantize}(\text{coeff } a_0) \\
\text{quantize}(\text{coeff } a_1) \\
\text{quantize}(\text{coeff } a_2) \\
\text{quantize}(\text{coeff } a_3) \\
\text{quantize}(\text{coeff } a_4) \\
\text{quantize}(\text{coeff } a_5) \\
\text{quantize}(\text{gain}\cdot\text{coeff } a_6) \\
\text{quantize}(\text{gain}\cdot\text{coeff } a_7) \\
\text{quantize}(\text{gain}\cdot\text{coeff } a_8)
\end{bmatrix} \\
\text{coeff}_{\text{aq}} &= \\
& \begin{bmatrix} 1 \\ -1.984230041503906 \\ 1 \\ 1 \end{bmatrix}
\begin{bmatrix} \text{coeff } a_6 \end{bmatrix} \\
& \begin{bmatrix} 0.966636657714844 \\ -1.918037414550781 \\ 0.966636657714844 \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{coeff}_{\text{bq}} & := \\
& \begin{bmatrix}
\text{quantize}(\text{coeff } b_0) \\
\text{quantize}(\text{coeff } b_1) \\
\text{quantize}(\text{coeff } b_2) \\
\text{quantize}(\text{coeff } b_3) \\
\text{quantize}(\text{coeff } b_4) \\
\text{quantize}(\text{coeff } b_5) \\
\end{bmatrix} \\
\text{coeff}_{\text{bq}} &= \\
& \begin{bmatrix} -1.951126098632813 \\ 0.966629028320313 \\ -1.972724914550781 \\ 0.985107421875 \end{bmatrix}
\begin{bmatrix} \text{coeff } b_6 \end{bmatrix} \\
& \begin{bmatrix} -1.961524963378906 \\ 0.981246948242188 \end{bmatrix}
\end{align*}
\]

Rewriting the equations yield

\[
\begin{align*}
H_{1q}(z) & := \frac{\text{coeff}_{\text{aq}} \cdot z^2 + \text{coeff}_{\text{aq}} \cdot z + \text{coeff}_{\text{aq}}}{z^2 + \text{coeff}_{\text{bq}} \cdot z + \text{coeff}_{\text{bq}}} \\
H_{2q}(z) & := \frac{\text{coeff}_{\text{aq}} \cdot z^2 + \text{coeff}_{\text{aq}} \cdot z + \text{coeff}_{\text{aq}}}{z^2 + \text{coeff}_{\text{bq}} \cdot z + \text{coeff}_{\text{bq}}} \\
H_{3q}(z) & := \frac{\text{coeff}_{\text{aq}} \cdot z^2 + \text{coeff}_{\text{aq}} \cdot z + \text{coeff}_{\text{aq}}}{z^2 + \text{coeff}_{\text{bq}} \cdot z + \text{coeff}_{\text{bq}}}
\end{align*}
\]
The transfer function $H_q(z)$ for the digital filter with quantized coefficients is simply the product of the three biquad sections:

$$H_q(z) := H_{1q}(z) \cdot H_{2q}(z) \cdot H_{3q}(z)$$

Notice that the DC gain of the filter can be found by substituting $z=1$ into $H_q(z)$:

$$H_q(1) = 1.000820626153403$$

Since the DC gain of the filter is now slightly greater than one, the gain factor must be re-adjusted and the quantized coefficients will now be recomputed using 18-bit coefficients. It should be noted that although the DC gain is real close to one, performing a bit-true simulation often shows that the filter gain is higher at frequencies other than DC. Thus, an alternate set of coefficients may need to be derived.

$$gain2 := \frac{\text{gain}}{H_q(1)} \quad gain2 = 0.965846824799088$$

$$n := 18 \quad b := n - 3 \quad b = 15$$

$$\text{num\_bits\_per\_unit} := 2^b$$

$$\text{num\_units\_per\_bit} := 2^b$$

$$\text{quantize}(x) := \text{floor}(x \cdot \text{num\_bits\_per\_unit}) \cdot \text{num\_units\_per\_bit}$$

$$coeff_{aq} := \begin{bmatrix}
\text{quantize}(coeff_{a_0}) \\
\text{quantize}(coeff_{a_1}) \\
\text{quantize}(coeff_{a_2}) \\
\text{quantize}(coeff_{a_3}) \\
\text{quantize}(coeff_{a_4}) \\
\text{quantize}(coeff_{a_5}) \\
\text{quantize}(gain2 \cdot coeff_{a_6}) \\
\text{quantize}(-gain2 \cdot coeff_{a_7}) \\
\text{quantize}(gain2 \cdot coeff_{a_8})
\end{bmatrix}$$

$$coeff_{aq} = \begin{bmatrix}
1 \\
-1.9842529296875 \\
1 \\
1 \\
-1.9842529296875 \\
1 \\
0.9658203125 \\
-1.916473388671875 \\
0.9658203125
\end{bmatrix}$$
coeff \mathbf{b}_q := \begin{bmatrix}
\text{quantize}(\text{coeff } b_0) \\
\text{quantize}(\text{coeff } b_1) \\
\text{quantize}(\text{coeff } b_2) \\
\text{quantize}(\text{coeff } b_3) \\
\text{quantize}(\text{coeff } b_4) \\
\text{quantize}(\text{coeff } b_5)
\end{bmatrix}

\text{coeff } \mathbf{b}_q = \begin{bmatrix}
-1.951141357421875 \\
0.96661376953125 \\
-1.972747802734375 \\
0.985107421875 \\
-1.9615478515625 \\
0.981231689453125
\end{bmatrix}

Now that the 18-bit coefficients have been computed, they will be used to verify that the DC gain is less than one. This is found to be the case as shown below:

\[
H_{1q}(z) := \frac{\text{coeff } a_{d_0} \cdot z^2 + \text{coeff } a_{q_1} \cdot z + \text{coeff } a_{d_2}}{z^2 + \text{coeff } a_{bq_0} \cdot z + \text{coeff } a_{bq_1}}
\]

\[
H_{2q}(z) := \frac{\text{coeff } a_{d_3} \cdot z^2 + \text{coeff } a_{q_4} \cdot z + \text{coeff } a_{d_5}}{z^2 + \text{coeff } a_{bq_2} \cdot z + \text{coeff } a_{bq_3}}
\]

\[
H_{3q}(z) := \frac{\text{coeff } a_{d_6} \cdot z^2 + \text{coeff } a_{q_7} \cdot z + \text{coeff } a_{d_8}}{z^2 + \text{coeff } a_{bq_4} \cdot z + \text{coeff } a_{bq_5}}
\]

The new quantized transfer function \(H_q(z)\) is now:

\[
H_q(z) := H_{1q}(z) \cdot H_{2q}(z) \cdot H_{3q}(z)
\]

and the DC gain of the new filter is

\[
H_q(1) = 0.999155526335014
\]
Thus, a digital filter whose DC gain is less than one has been successfully designed. The resulting 18-bit quantized coefficients for the filter are

\[
\text{coeff}_{aq} = \begin{bmatrix}
1 \\
-1.9842529296875 \\
1 \\
-1.9842529296875 \\
0.9658203125 \\
-1.916473388671875 \\
0.9658203125
\end{bmatrix} \quad \text{coeff}_{bq} = \begin{bmatrix}
-1.951141357421875 \\
0.96661376953125 \\
-1.972747802734375 \\
0.985107421875 \\
-1.9615478515625 \\
0.981231689453125
\end{bmatrix}
\]

Simulation of the filter verified that the filter met the original specifications. The filter frequencies of the digital filter with quantized coefficients are given below:

\[
f_{\text{lower}\_\text{df1}} := 50.14 \text{ Hz} \quad \text{(where passband ends; } \geq 50 \text{ Hz)}
\]

\[
f_{1\_\text{df1}} := 57.92 \text{ Hz} \quad \text{(where stopband begins; } \leq 58 \text{ Hz)}
\]

\[
f_{2\_\text{df1}} := 62.00 \text{ Hz} \quad \text{(where stopband ends; } \geq 62 \text{ Hz)}
\]

\[
f_{\text{upper}\_\text{df1}} := 71.62 \text{ Hz} \quad \text{(where passband begins; } \leq 72 \text{ Hz)}
\]

At all frequencies between DC and 1500 Hz, the gain of the filter was less than one. In addition, all the specifications were met or exceeded.

A second, more simpler method of obtaining the coefficients will also be presented. The results were quite good, but the filter frequency response was a little less desirable than the filter achieved using the previous method. This second method involves quantizing the original coefficients to 18-bits using a truncation method which results in smaller absolute values for the coefficients for both positive and negative coefficients. Previously, the coefficients were truncated with the "floor" function, which truncates the coefficient to the smaller integer. For negative coefficients, the smaller integer is the more negative of the two choices, and thus, the absolute value of the coefficient will be higher when the "floor" function is used with negative coefficients. This second method will truncate such that the absolute value of the coefficient will always be less than or equal to the absolute value of the unquantized coefficient.
The second method will be set up similar to the original method.

\[ n := 18 \quad b := n - 3 \quad b = 15 \]

\[ \text{num_bits_per_unit} := 2^b \]
\[ \text{num_units_per_bit} := 2^b \]

\[ \text{quantize}_1(x) := \text{floor}(x \cdot \text{num_bits_per_unit}) \cdot \text{num_units_per_bit} \]
\[ \text{quantize}_2(x) := \text{ceil}(x \cdot \text{num_bits_per_unit}) \cdot \text{num_units_per_bit} \]

\[
\begin{bmatrix}
\text{quantize}_1(\text{coeff } a_0) \\
\text{quantize}_2(\text{coeff } a_1) \\
\text{quantize}_1(\text{coeff } a_2) \\
\text{quantize}_1(\text{coeff } a_3) \\
\text{quantize}_2(\text{coeff } a_4) \\
\text{quantize}_1(\text{coeff } a_5) \\
\text{quantize}_1(|\text{gain} \cdot \text{coeff } a_6|) \\
\text{quantize}_2(-|\text{gain} \cdot \text{coeff } a_7|) \\
\text{quantize}_1(|\text{gain} \cdot \text{coeff } a_8|)
\end{bmatrix}
\begin{bmatrix}
1 \\
-1.984222412109375 \\
1 \\
1 \\
0.96661376953125 \\
-1.91802978515625 \\
0.96661376953125
\end{bmatrix}
\]

\[
\begin{bmatrix}
\text{quantize}_2(\text{coeff } b_0) \\
\text{quantize}_1(\text{coeff } b_1) \\
\text{quantize}_2(\text{coeff } b_2) \\
\text{quantize}_1(\text{coeff } b_3) \\
\text{quantize}_2(\text{coeff } b_4) \\
\text{quantize}_1(\text{coeff } b_5)
\end{bmatrix}
\begin{bmatrix}
-1.95111083984375 \\
0.96661376953125 \\
-1.97271728515625 \\
0.985107421875 \\
-1.961517333984375 \\
0.981231689453125
\end{bmatrix}
\]
Rewriting the equations yield

\[ H_{1q}(z) := \frac{\text{coeff}_{aq_0} \cdot z^2 + \text{coeff}_{aq_1} \cdot z + \text{coeff}_{aq_2}}{z^2 + \text{coeff}_{bq_0} \cdot z + \text{coeff}_{bq_1}} \]

\[ H_{2q}(z) := \frac{\text{coeff}_{aq_3} \cdot z^2 + \text{coeff}_{aq_4} \cdot z + \text{coeff}_{aq_5}}{z^2 + \text{coeff}_{bq_2} \cdot z + \text{coeff}_{bq_3}} \]

\[ H_{3q}(z) := \frac{\text{coeff}_{aq_6} \cdot z^2 + \text{coeff}_{aq_7} \cdot z + \text{coeff}_{aq_8}}{z^2 + \text{coeff}_{bq_4} \cdot z + \text{coeff}_{bq_5}} \]

The transfer function \( H_q(z) \) for the digital filter with quantized coefficients is again, simply the product of the three biquad sections:

\[ H_q(z) := H_{1q}(z) \cdot H_{2q}(z) \cdot H_{3q}(z) \]

Once again, the DC gain of the filter can be found by substituting \( z=1 \) into \( H_q(z) \):

\[ H_q(1) = 0.999052164558751 \]

This quantized filter exceeded the filter frequency specifications as shown below:

\[ f_{\text{lower}_{df2}} := 50.15 \text{ Hz} \quad \text{(where passband ends; } \geq 50 \text{ Hz)} \]

\[ f_{1_{df2}} := 57.98 \text{ Hz} \quad \text{(where stopband begins; } \leq 58 \text{ Hz)} \]

\[ f_{2_{df2}} := 62.05 \text{ Hz} \quad \text{(where stopband ends; } \geq 62 \text{ Hz)} \]

\[ f_{\text{upper}_{df2}} := 71.62 \text{ Hz} \quad \text{(where passband begins; } \leq 72 \text{ Hz)} \]

The gain of the filter was less than one for all frequencies less than 93.17 Hz. However, the gain for frequencies above 93.17 Hz rose to a peak of 0.00171 dB (magnitude gain of 1.00019689) at 108.50 Hz and then decreased exponentially, asymptotic to the positive side of the 0 dB line. This gain is so small that when the maximum amplitude input signal of 8-bits (represent -128 to 127, for a maximum amplitude of 128) is used, the gain is essentially no greater than one since when the maximum amplitude is multiplied by a factor such as 1.00019689, the same 8-bit quantized value would result.

Thus, this filter seems to give acceptable results also. However, since the results of the first method achieved a digital filter meeting all the specifications with a gain less than one for all frequencies, the first method will be used for the VLSI digital filter.
Appendix B: Mistral1 Generated VHDL Code

LIBRARY msl VHDL lib;
USE msl VHDL lib.all;

-- primary unit : entity declaration

ENTITY DFL IS
  PORT (ms1_IN, blcjn, scaninj, scanin_2, scanin_3, phi, tc : IN bit;
 /ms1_OUT, scanout_1, scanout_2, scanout_3 : OUT bit);
END DFL;

-- secondary unit : structural architecture body

ARCHITECTURE struct OF DFL IS

-- component declarations

COMPONENT ms1_brep
        PORT (inl, r, p, scan_in, phi, tc : IN bit;
        outl : OUT bit);
END COMPONENT;

COMPONENT ms1_enla
        PORT (in1, e, scan_in, phi, tc : IN bit;
        outl : OUT bit);
END COMPONENT;

COMPONENT ms1_mux
        PORT (m, in1, in2, scan_in, phi, tc : IN bit;
        outl : OUT bit);
END COMPONENT;

COMPONENT ms1_addere
        PORT (in1, in2, cin, s, scan_in, phi, tc : IN bit;
        outl, cout : OUT bit);
END COMPONENT;

COMPONENT ms1_adder
        PORT (in1, in2, s, scan_in, phi, tc : IN bit;
        outl, cout : OUT bit);
END COMPONENT;

COMPONENT ms1_and
        PORT (in1, in2, scan_in, phi, tc : IN bit;
        outl : OUT bit);
END COMPONENT;

COMPONENT ms1_exor
        PORT (in1, in2, scan_in, phi, tc : IN bit;
        outl : OUT bit);
END COMPONENT;

COMPONENT ms1_inv
        PORT (in1, scan_in, phi, tc : IN bit;
        outl : OUT bit);
END COMPONENT;

COMPONENT ms1_or
        PORT (in1, in2, scan_in, phi, tc : IN bit;
        outl : OUT bit);
END COMPONENT;

COMPONENT ms1_subtc
        PORT (in1, in2, cin, s, scan_in, phi, tc : IN bit;
        outl, cout : OUT bit);
END COMPONENT;

B-1
COMPONENT msl_subt
PORT (inl, in2, s, scanjn, phi, tc: IN bit; outl, cout: OUT bit);
END COMPONENT;

COMPONENT msl_dff
PORT (inl, phi: IN bit; outl: OUT bit);
END COMPONENT;

COMPONENT msl_srga
PORT (s, r, scanjn, phi, tc: IN bit; outl: OUT bit);
END COMPONENT;

COMPONENT msl_zinj
PORT (inl, zero, p, scanjn, phi, tc: IN bit; outl: OUT bit);
END COMPONENT;

COMPONENT msl_juffer
PORT (inl: IN bit; outl: OUT bit);
END COMPONENT;

COMPONENT msl_scdf
PORT (inl, scanjn, phi, tc: IN bit; outl: OUT bit);
END COMPONENT;

COMPONENT msl_np_brep
PORT (ml, r, p, scanjn, phi, tc: IN bit; outl, scan_out: OUT bit);
END COMPONENT;

COMPONENT msl_np_enla
PORT (inl, e, scanjn, phi, tc: IN bit; outl, scan_out: OUT bit);
END COMPONENT;

COMPONENT msl_np_mux
PORT (m, inl, in2: IN bit; outl: OUT bit);
END COMPONENT;

COMPONENT msl_np_adderc
PORT (ml, in2, cin, s, scanjn, phi, tc: IN bit; outl, cout, scan_out: OUT bit);
END COMPONENT;

COMPONENT msl_np_adder
PORT (inl, in2, s, scanjn, phi, tc: IN bit; outl, cout, scan_out: OUT bit);
END COMPONENT;

COMPONENT msl_np_and
PORT (inl, in2: IN bit; outl: OUT bit);
END COMPONENT;

COMPONENT msl_np_exor
PORT (inl, in2: IN bit; outl: OUT bit);
END COMPONENT;

COMPONENT msl_jifp
PORT (inl: IN bit; outl: OUT bit);
END COMPONENT;
COMPONENT msl_np_or
  PORT (in1, in2 : IN bit;
    out1 : OUT bit);
END COMPONENT;

COMPONENT msl_scdff
  PORT (s, phi, r, in2, tmp0, scan_out : IN bit;
    out1, scan_out : OUT bit);
END COMPONENT;

COMPONENT msl_np_srga
  PORT (s, phi, r, in2, scan_in, phi, tc : IN bit;
    out1, out2, scan_out : OUT bit);
END COMPONENT;

COMPONENT msl_np_subtc
  PORT (s, phi, r, in2, scan_in, phi, tc : IN bit;
    out1, out2, scan_out : OUT bit);
END COMPONENT;

COMPONENT msl_zinj
  PORT (in1, zero, p, scan_in, phi : IN bit;
    out1, scan_out : OUT bit);
END COMPONENT;

FOR ALL: msl_zinj USE ENTITY msl_vhdl_lib.msl_zinj(behav);
FOR ALL: msl_brep USE ENTITY msl_vhdl_lib.msl_brep(behav);
FOR ALL: msl_subt USE ENTITY msl_vhdl_lib.msl_subt(behav);
FOR ALL: msl_adder USE ENTITY msl_vhdl_lib.msl_adder(behav);
FOR ALL: msl_np_brep USE ENTITY msl_vhdl_lib.msl_np_brep(behav);
FOR ALL: msl_np_subt USE ENTITY msl_vhdl_lib.msl_np_subt(behav);
FOR ALL: msl_np_zinj USE ENTITY msl_vhdl_lib.msl_np_zinj(behav);
FOR ALL: msl_dff USE ENTITY msl_vhdl_lib.msl_dff(behav);
FOR ALL: msl_buffer USE ENTITY msl_vhdl_lib.msl_buffer(behav);
FOR ALL: msl_scdff USE ENTITY msl_vhdl_lib.msl_scdff(behav);

-- internal signals

SIGNAL tmp0_cas2, tmp0, OUT_bu41, Mn1_nmp5, Mn1_nmp6, Mn1_nmp6_cy10, Mn1_nmp7,
  Mn1_nmp8, Mn1_nmp8_cy14, Mn1_nmp9, Mn1_cy18, Mn2_nmp12, Mn2_nmp13, Mn2_nmp13_cy22,
  Mn2_nmp14, Mn2_nmp15, Mn2_nmp15_cy26, Mn2_nmp16, Mn2_nmp17, Mn2_nmp17_cy30, Mn2_nmp18, Mn2_nmp19,
  Mn2_nmp19_cy34, Mn2_nmp20, Mn2_nmp21, Mn2_nmp21_cy38, Mn2_nmp22, Mn2_nmp23, Mn2_nmp23_cy42, Mn2,
  Mn3_nmp25, Mn3_nmp26, Mn3_nmp26_cy48, Mn3_nmp27, Mn3_nmp28, Mn3_nmp28_cy52, Mn3_nmp29, Mn3,
  Mn3_cy56, Md1_nmp32, Md1_nmp32_cy56, Md1_nmp33, Md1_nmp33_cy60, Md1_nmp33 Scot, Md1_nmp34, Md1_nmp34_cy64,
  Md1_nmp35, Md1_nmp35_cy64, Md1_nmp35_cy64, Md1_nmp35_cy64, Md1_nmp35_cy66, Md1_nmp36, Md1_nmp36 Scot, Md1_nmp37, Md1_nmp37_cy68,
Md1_nmp37 Scot,
  Md1_nmp38, Md1_nmp38 Scot, Md1_nmp39, Md1_nmp39_cy72, Md1_nmp39 Scot, Md1, Md1 Scot30, Md2_nmp41,
  Md2_nmp42, Md2_nmp42_cy78, Md2_nmp43, Md2_nmp44, Md2_nmp44_cy82, Md2_nmp45, Md2_nmp46, Md2_nmp46_cy86,
  Md2_nmp47, Md2_nmp48, Md2_nmp48_cy90, Md2_nmp49, Md2, Md2_cy94, tmp10, tmp10_cy96,
OUT_cas3, OUT_cas3_cy98, tmp11, tmp11_cy100, tmp11_cy100, tmp11_cy100, A1, A1_cy102, A1_Scout4,
  Mn2_uni5, Mn2_uni5 Scot, Mn2_uni5 Scot, Mn2_uni5 Scot, Mn2_uni5 Scot, Mn2_uni5 Scot,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
Md1_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55, Mn2_uni5_nmp55,
comp_74: msl_adder
PORT MAP(i1 => Md1_uni4, in2 => Md2_uni4, s => blc_6, scan_in => Md2_uni4, out1 => tmp21, Cout => tmp21_cyl148, scan_out => tmp21 scouts7, phi => phi, tc => tc);

comp_75: msl_ni_adder
PORT MAP(i1 => tmp21_he44, in2 => Y1, s => blc_1, scan_in => Y1, out1 => Mn1_uni0, Cout => Mn1_uni0_cyl50, scan_out
=> Mn1_uni6 scouts5, phi => phi, tc => tc);

comp_76: msl_br
PORT MAP(i1 => Mn1_uni7, r => blc_18, p => blc_5, scan_in => Md2_uni1 npm68, out1 => Mn2_uni8 npm76, phi => phi, tc => tc);

comp_77: msl_adder
PORT MAP(i1 => Mn1_uni10 npm75_h, in2 => Mn2_uni8 npm76, s => blc_7, scan_in => Mn2_uni8 npm75_h, out1 =>
Mn2_uni8 npm77, Cout => Mn2_uni8 npm77_c, phi => phi, tc => tc);

comp_78: msl_br
PORT MAP(i1 => Mn1_uni10 npm77, r => blc_7, p => blc_14, scan_in => Mn2_uni8 npm77, out1 => Mn2_uni8 npm78, phi
=> phi, tc => tc);

comp_79: msl_sub
PORT MAP(i1 => Mn2_uni8 npm78, in2 => net_4, s => blc_16, scan_in => net_4, out1 => Mn2_uni8 npm79, Cout =>
Mn2_uni8 npm79_c, phi => phi, tc => tc);

comp_80: msl_zinj
PORT MAP(i1 => Mn2_uni8 npm79_h, zero => blc_17, p => blc_18, scan_in => Mn2_uni8 npm79_h, out1 => Mn2 unii8, phi
=> phi, tc => tc);

comp_81: msl_np_br
PORT MAP(i1 => Md1_uni10 npm81, r => blc_18, p => blc_4, scan_in => Mn2_uni8 npm76, out1 => Md1_uni10 npm82, scan_out
=> net_15, phi => phi, tc => tc);

comp_82: msl_np_sub
PORT MAP(i1 => Md1_uni10 npm82, in2 => delay27_ne11, s => blc_5, scan_in => Md1_uni3 scouts3, out1 =>
Md1_uni10 npm83, Cout => net_5, scan_out => net_19, phi => phi, tc => tc);

comp_83: msl_np_br
PORT MAP(i1 => Md1_uni10 npm83, r => blc_4, p => blc_7, scan_in => net_15, out1 => Md1_uni10 npm84, scan_out
=> net_16, phi => phi, tc => tc);

comp_84: msl_np_adder
PORT MAP(i1 => delay27_he20, in2 => Md1_uni10 npm84, s => blc_8, scan_in => net_19, out1 => Md1_uni10 npm85, Cout
=> net_6, scan_out => net_20, phi => phi, tc => tc);

comp_85: msl_np_br
PORT MAP(i1 => Md1_uni10 npm85, r => blc_7, p => blc_9, scan_in => net_16, out1 => Md1_uni10 npm86, scan_out
=> net_17, phi => phi, tc => tc);

comp_86: msl_np_sub
PORT MAP(i1 => Md1_uni10 npm86, in2 => delay27_he26, s => blc_10, scan_in => net_20, out1 => Md1_uni10 npm87, Cout
=> net_7, scan_out => net_21, phi => phi, tc => tc);

comp_87: msl_np_br
PORT MAP(i1 => Md1_uni10 npm87, r => blc_9, p => blc_14, scan_in => net_17, out1 => Md1_uni10 npm88, scan_out
=> net_18, phi => phi, tc => tc);

comp_88: msl_np_adder
PORT MAP(i1 => delay27_he48, in2 => Md1_uni10 npm88, s => blc_15, scan_in => delay27_he48, out1 =>
Md1_uni10 npm89, Cout => net_8, scan_out => net_14, phi => phi, tc => tc);

comp_89: msl_np_zinj
PORT MAP(i1 => Md1_uni10 npm89, zero => blc_13, p => blc_14, scan_in => net_21, out1 => Md1_uni10, scan_out
=> Md1_uni10 scouts, phi => phi, tc => tc);

comp_90: msl_br
PORT MAP(i1 => Md2_uni11 npm90, r => blc_19, p => blc_1, scan_in => net_18, out1 => Md2_uni11 npm91, phi => phi, tc
=> tc);

comp_91: msl_adder
PORT MAP(i1 => delay29_he12, in2 => Md2_uni11 npm91, s => blc_3, scan_in => delay29_he12, out1 => Md2_uni11 npm92, Cout
=> net_9, phi => phi, tc => tc);

comp_92: msl_br
PORT MAP(i1 => Md2_uni11 npm92, r => blc_3, p => blc_6, scan_in => Md2_uni11 npm92, out1 => Md2_uni11 npm93, phi
=> phi, tc => tc);

comp_93: msl_adder
PORT MAP(i1 => delay29_he21, in2 => Md2_uni11 npm93, s => blc_8, scan_in => delay29_he21, out1 => Md2_uni11 npm94, Cout
=> net_10, phi => phi, tc => tc);

comp_94: msl_br
PORT MAP(i1 => Md2_uni11 npm94, r => blc_8, p => blc_12, scan_in => Md2_uni11 npm94, out1 => Md2_uni11 npm95, phi
=> phi, tc => tc);

comp_95: msl_adder
PORT MAP(i1 => delay29_he27, in2 => Md2_uni11 npm95, s => blc_14, scan_in => delay29_he27, out1 =>
Md2_uni11 npm96, Cout => net_11, phi => phi, tc => tc);

comp_96: msl_br
PORT MAP(i1 => Md2_uni11 npm96, r => blc_14, p => blc_19, scan_in => Md2_uni11 npm96, out1 => Md2_uni11 npm97, phi
=> phi, tc => tc);
comp_97: msl_subt
PORT MAP(in1 => Md2_uni11_nmp97, in2 => delay29_he49, s => blc_1, scan_in => delay29_he49, outl => Md2_uni11, cout => Md2_uni11_cyl94, phi => phi, tc => tc);

comp_98: msl_adder
PORT MAP(in1 => Mn1_uni7, in2 => Mn2_uni8, s => blc_19, scan_in => Mn2_uni8, outl => tmp30, cout => tmp30_cyl196, phi => phi, tc => tc);

comp_99: msl_adder
PORT MAP(in1 => Mn1_uni7, in2 => Mn3_uni9, s => blc_in_bu40, scan_in => Mn1_uni3_nmp59_s, outl => Y1, cout => Y1_cyl198, phi => phi, tc => tc);

comp_100: msl_in_subt
PORT MAP(in1 => Mn1_uni10, in2 => Mn2_uni11_he50, s => blc_14, scan_in => Mn2_uni11_he50, outl => tmp31, cout => tmp31_cyl200, scan_out => tmp31_scout9, phi => phi, tc => tc);

comp_101: msl_in_subt
PORT MAP(in1 => Mn1_uni13, in2 => tmp0, s => blc_19, scan_in => tmp0, outl => Mn1_uni7, cout => Mn1_uni7_cyl202, scan_out => Mn1_uni7_scout2, phi => phi, tc => tc);

comp_102: msl_in_subt
PORT MAP(in1 => Ground, in2 => A1, s => blc_3, scan_in => Ground, outl => Md2_nmp40, cout => Md2_nmp40_cyl204, phi => phi, tc => tc);

comp_103: msl_in_subt
PORT MAP(in1 => Ground, in2 => Mn1_uni0, s => blc_1, scan_in => Mn1_uni10_scout, outl => Mn1_uni3_nmp58, cout => Mn1_uni3_nmp58_c, scan_out => Mn1_uni3_nmp58_s, phi => phi, tc => tc);

comp_104: msl_in_subt
PORT MAP(in1 => Ground, in2 => Mn1_uni7, s => blc_19, scan_in => Mn1_uni3_nmp58_s, outl => Mn1_uni10_nmp81, cout => net_12, scan_out => net_22, phi => phi, tc => tc);

comp_105: msl_in_subt
PORT MAP(in1 => Ground, in2 => Mn1_uni7, s => blc_19, scan_in => Mn2_uni11_nmp91, outl => Mn2_uni11_nmp90, cout => net_13, phi => phi, tc => tc);

comp_106: msl_in_diff
PORT MAP(in1 => Mn2_nmp11_he1, out1 => net_24, phi => phi);

comp_107: msl_in_diff
PORT MAP(in1 => delay9_he4, out1 => net_26, phi => phi);

comp_108: msl_in_diff
PORT MAP(in1 => delay27_he11, out1 => net_27, phi => phi);

comp_109: msl_in_diff
PORT MAP(in1 => Md1_nmp31_he15, out1 => Md1_nmp31_he23_e, phi => phi);

comp_110: msl_in_diff
PORT MAP(in1 => delay9_he16, out1 => net_28, phi => phi);

comp_111: msl_in_diff
PORT MAP(in1 => delay17_he18, out1 => net_30, phi => phi);

comp_112: msl_in_diff
PORT MAP(in1 => delay27_he20, out1 => delay27_he26_ex, phi => phi);

comp_113: msl_in_diff
PORT MAP(in1 => Mn2_nmp11_he22, out1 => net_32, phi => phi);

comp_114: msl_in_diff
PORT MAP(in1 => delay9_he24, out1 => net_34, phi => phi);

comp_115: msl_in_diff
PORT MAP(in1 => Mn2_nmp11_he28, out1 => net_36, phi => phi);

comp_116: msl_in_diff
PORT MAP(in1 => Mn2_nmp23, out1 => net_42, phi => phi);

comp_117: msl_in_diff
PORT MAP(in1 => Md2, out1 => Md2_he37_exp6, phi => phi);

comp_118: msl_in_diff
PORT MAP(in1 => Mn2_uni1_nmp56, out1 => net_43, phi => phi);

comp_119: msl_in_diff
PORT MAP(in1 => Mn2_uni8_nmp79, out1 => net_44, phi => phi);

comp_120: msl_in_diff
PORT MAP(in1 => delay27_he26, out1 => net_47, phi => phi);

comp_121: msl_in_diff
PORT MAP(in1 => blc_2, out1 => blc_3, phi => phi);

comp_122: msl_in_diff
PORT MAP(in1 => blc_3, out1 => blc_4, phi => phi);

comp_123: msl_in_diff
PORT MAP(in1 => blc_4, out1 => blc_5, phi => phi);

comp_124: msl_in_diff
PORT MAP(in1 => blc_5, out1 => blc_6, phi => phi);

comp_125: msl_in_diff
PORT MAP(in1 => blc_6, out1 => blc_7, phi => phi);

comp_126: msl_in_diff
PORT MAP(in1 => blc_7, out1 => blc_8, phi => phi);
PORT MAP(in1 => Mn3_uni9_exp30, out1 => Mn3_uni9_exp29, phi => phi);
comp_300: msl_dff
PORT MAP(in1 => Mn3_uni9_exp31, out1 => Mn3_uni9_exp30, phi => phi);
comp_301: msl_dff
PORT MAP(in1 => Mn3_uni9_exp32, out1 => Mn3_uni9_exp31, phi => phi);
comp_302: msl_dff
PORT MAP(in1 => Mn3_uni9_exp33, out1 => Mn3_uni9_exp32, phi => phi);
comp_303: msl_dff
PORT MAP(in1 => Mn3_uni9_exp34, out1 => Mn3_uni9_exp33, phi => phi);
comp_304: msl_dff
PORT MAP(in1 => Mn3_uni9_exp35, out1 => Mn3_uni9_exp34, phi => phi);
comp_305: msl_dff
PORT MAP(in1 => Mn3_uni9_exp36, out1 => Mn3_uni9_exp35, phi => phi);
comp_306: msl_dff
PORT MAP(in1 => Mn3_uni9_exp37, out1 => Mn3_uni9_exp36, phi => phi);
comp_307: msl_dff
PORT MAP(in1 => Mn3_uni9_exp38, out1 => Mn3_uni9_exp37, phi => phi);
comp_308: msl_dff
PORT MAP(in1 => Mn3_uni9_exp39, out1 => Mn3_uni9_exp38, phi => phi);
comp_310: msl_dff
PORT MAP(in1 => Md2_uni4_nmp67, out1 => Md2_uni4_nmp67, phi => phi);
comp_311: msl_dff
PORT MAP(in1 => net_48, out1 => Md2_uni4_nmp67, phi => phi);
comp_312: msl_dff
PORT MAP(in1 => net_50, out1 => net_48, phi => phi);
comp_313: msl_dff
PORT MAP(in1 => net_51, out1 => net_49, phi => phi);
comp_314: msl_dff
PORT MAP(in1 => net_52, out1 => net_50, phi => phi);
comp_315: msl_dff
PORT MAP(in1 => net_53, out1 => net_51, phi => phi);
comp_316: msl_dff
PORT MAP(in1 => net_54, out1 => net_52, phi => phi);
comp_317: msl_dff
PORT MAP(in1 => net_55, out1 => net_53, phi => phi);
comp_318: msl_dff
PORT MAP(in1 => net_56, out1 => net_54, phi => phi);
comp_319: msl_dff
PORT MAP(in1 => net_57, out1 => net_55, phi => phi);
comp_320: msl_dff
PORT MAP(in1 => net_58, out1 => net_56, phi => phi);
comp_321: msl_dff
PORT MAP(in1 => net_59, out1 => net_57, phi => phi);
comp_322: msl_dff
PORT MAP(in1 => net_60, out1 => net_58, phi => phi);
comp_323: msl_dff
PORT MAP(in1 => net_61, out1 => net_59, phi => phi);
comp_324: msl_dff
PORT MAP(in1 => net_62, out1 => net_60, phi => phi);
comp_325: msl_dff
PORT MAP(in1 => Mn3_uni2_exp1, out1 => Mn3_uni2, phi => phi);
comp_326: msl_dff
PORT MAP(in1 => Mn3_uni2_exp2, out1 => Mn3_uni2_exp1, phi => phi);
comp_327: msl_dff
PORT MAP(in1 => Mn3_uni2_exp3, out1 => Mn3_uni2_exp2, phi => phi);
comp_328: msl_dff
PORT MAP(in1 => Mn3_uni2_exp4, out1 => Mn3_uni2_exp3, phi => phi);
comp_329: msl_dff
PORT MAP(in1 => Mn3_uni2_exp5, out1 => Mn3_uni2_exp4, phi => phi);
comp_330: msl_dff
PORT MAP(in1 => Mn3_uni2_exp6, out1 => Mn3_uni2_exp5, phi => phi);
comp_332: msl_dff
PORT MAP(in1 => Mn3_uni2_exp7, out1 => Mn3_uni2_exp6, phi => phi);
comp_333: msl_dff
PORT MAP(in1 => Mn3_uni2_exp8, out1 => Mn3_uni2_exp7, phi => phi);
PORT MAP(in1 => net_81, out1 => net_4, phi => phi); comp_369: ms1_ff
PORT MAP(in1 => net_82, out1 => net_81, phi => phi); comp_370: ms1_ff
PORT MAP(in1 => net_83, out1 => net_82, phi => phi); comp_371: ms1_ff
PORT MAP(in1 => net_84, out1 => net_83, phi => phi); comp_372: ms1_ff
PORT MAP(in1 => net_85, out1 => net_84, phi => phi); comp_373: ms1_ff
PORT MAP(in1 => net_86, out1 => net_85, phi => phi); comp_374: ms1_ff
PORT MAP(in1 => net_87, out1 => net_86, phi => phi); comp_375: ms1_ff
PORT MAP(in1 => net_88, out1 => net_87, phi => phi); comp_376: ms1_ff
PORT MAP(in1 => net_89, out1 => Mn2_uni1_nmp52_h, phi => phi); comp_377: ms1_ff
PORT MAP(in1 => net_90, out1 => net_89, phi => phi); comp_378: ms1_ff
PORT MAP(in1 => net_91, out1 => net_90, phi => phi); comp_379: ms1_ff
PORT MAP(in1 => net_92, out1 => net_91, phi => phi); comp_380: ms1_ff
PORT MAP(in1 => net_93, out1 => net_92, phi => phi); comp_381: ms1_ff
PORT MAP(in1 => net_94, out1 => net_93, phi => phi); comp_382: ms1_ff
PORT MAP(in1 => net_95, out1 => net_94, phi => phi); comp_383: ms1_ff
PORT MAP(in1 => net_96, out1 => Mn2_uni8_nmp75_h, phi => phi); comp_384: ms1_ff
PORT MAP(in1 => net_97, out1 => net_96, phi => phi); comp_385: ms1_ff
PORT MAP(in1 => net_98, out1 => net_97, phi => phi); comp_386: ms1_ff
PORT MAP(in1 => net_99, out1 => net_98, phi => phi); comp_387: ms1_ff
PORT MAP(in1 => net_100, out1 => net_99, phi => phi); comp_388: ms1_ff
PORT MAP(in1 => net_101, out1 => net_100, phi => phi); comp_389: ms1_ff
PORT MAP(in1 => net_102, out1 => net_101, phi => phi); comp_390: ms1_ff
PORT MAP(in1 => delay9_he36_exp, out1 => delay9_he36, phi => phi); comp_391: ms1_ff
PORT MAP(in1 => net_103, out1 => delay9_he36_exp, phi => phi); comp_392: ms1_ff
PORT MAP(in1 => net_104, out1 => net_103, phi => phi); comp_393: ms1_ff
PORT MAP(in1 => net_105, out1 => net_104, phi => phi); comp_394: ms1_ff
PORT MAP(in1 => net_106, out1 => net_105, phi => phi); comp_395: ms1_ff
PORT MAP(in1 => net_107, out1 => net_106, phi => phi); comp_396: ms1_ff
PORT MAP(in1 => net_108, out1 => net_107, phi => phi); comp_397: ms1_ff
PORT MAP(in1 => net_109, out1 => net_3, phi => phi); comp_398: ms1_ff
PORT MAP(in1 => net_110, out1 => net_109, phi => phi); comp_399: ms1_ff
PORT MAP(in1 => net_111, out1 => net_110, phi => phi); comp_400: ms1_ff
PORT MAP(in1 => net_112, out1 => net_111, phi => phi); comp_401: ms1_ff
PORT MAP(in1 => net_113, out1 => net_112, phi => phi); comp_402: ms1_ff
PORT MAP(in1 => net_114, out1 => net_113, phi => phi);
comp_472: msl_dff
PORT MAP(inl => A1_he17_exp3, outl => A1_he17_exp2, phi => phi);

comp_473: msl_dff
PORT MAP(inl => A1_he17_exp4, outl => A1_he17_exp3, phi => phi);

comp_474: msl_dff
PORT MAP(inl => delay17_he18_ex, outl => delay17_he18, phi => phi);

comp_475: msl_dff
PORT MAP(inl => net_157, outl => delay17_he18_ex, phi => phi);

comp_476: msl_dff
PORT MAP(inl => net_158, outl => net_157, phi => phi);

comp_477: msl_dff
PORT MAP(inl => net_159, outl => net_158, phi => phi);

comp_478: msl_dff
PORT MAP(inl => delay29_he21_ex, outl => delay29_he21, phi => phi);

comp_479: msl_dff
PORT MAP(inl => net_160, outl => delay29_he21_ex, phi => phi);

comp_480: msl_dff
PORT MAP(inl => net_161, outl => net_160, phi => phi);

comp_481: msl_dff
PORT MAP(inl => net_162, outl => net_161, phi => phi);

comp_482: msl_dff
PORT MAP(inl => tmp21_he44, outl => tmp21_he44, phi => phi);

comp_483: msl_dff
PORT MAP(inl => tmp21_he44_exp2, outl => tmp21_he44_exp1, phi => phi);

comp_484: msl_dff
PORT MAP(inl => tmp21_he44_exp3, outl => tmp21_he44_exp2, phi => phi);

comp_485: msl_dff
PORT MAP(inl => tmp21_he44_exp4, outl => tmp21_he44_exp3, phi => phi);

comp_486: msl_dff
PORT MAP(inl => tmp31_he51, outl => tmp31_he51, phi => phi);

comp_487: msl_dff
PORT MAP(inl => tmp31_he51_exp2, outl => tmp31_he51_exp1, phi => phi);

comp_488: msl_dff
PORT MAP(inl => tmp31_he51_exp3, outl => tmp31_he51_exp2, phi => phi);

comp_489: msl_dff
PORT MAP(inl => tmp31_he51_exp4, outl => tmp31_he51_exp3, phi => phi);

comp_490: msl_dff
PORT MAP(inl => delay29_he12_ex, outl => delay29_he12, phi => phi);

comp_491: msl_dff
PORT MAP(inl => net_163, outl => delay29_he12_ex, phi => phi);

comp_492: msl_dff
PORT MAP(inl => net_164, outl => net_163, phi => phi);

comp_493: msl_dff
PORT MAP(inl => Md1_nmp31_he15_e, outl => Md1_nmp31_he15, phi => phi);

comp_494: msl_dff
PORT MAP(inl => net_165, outl => Md1_nmp31_he15_e, phi => phi);

comp_495: msl_dff
PORT MAP(inl => net_166, outl => net_165, phi => phi);

comp_496: msl_dff
PORT MAP(inl => Mn2_nmp11_he22_e, outl => Mn2_nmp11_he22, phi => phi);

comp_497: msl_dff
PORT MAP(inl => net_167, outl => Mn2_nmp11_he22_e, phi => phi);

comp_498: msl_dff
PORT MAP(inl => net_168, outl => net_167, phi => phi);

comp_499: msl_dff
PORT MAP(inl => Mn3_nmp24_he2_e, outl => Mn3_nmp24_he2, phi => phi);

comp_500: msl_dff
PORT MAP(inl => net_169, outl => Mn3_nmp24_he2_e, phi => phi);

comp_501: msl_dff
PORT MAP(inl => A1_he5_exp1, outl => A1_he5, phi => phi);

comp_502: msl_dff
PORT MAP(inl => A1_he5_exp2, outl => A1_he5_exp1, phi => phi);

comp_503: msl_dff
PORT MAP(inl => net_170, outl => Md2_uni4_nmp67_h, phi => phi);

comp_504: msl_dff
PORT MAP(inl => net_171, outl => net_170, phi => phi);

comp_505: msl_dff
PORT MAP(inl => Md1_nmp31_he3_e, outl => Md1_nmp31_he3, phi => phi);

comp_506: msl_dff
PORT MAP(in1 => delay17_he7, out1 => delay17_he7, phi => phi);
END struct;
Appendix C: I/O Schematic For VLSI Chip
Appendix D: Tutorial

This is a basic tutorial to lead the user through the tools using simplistic models and commands. This is meant to be an introduction to the Mentor Graphics DSP Station tools and is not meant to create a flawless design. Before beginning, make a TUTORIAL directory in your home account:

```
> mkdir ~/TUTORIAL
```

**FILlab**

**Versions:**
- FILlab v8.4_2.3
- Falcon Framework v8.4_1.1

To invoke the FILlab program, type

```
> fillab
```

To open the basic design, select Design->Select from the top menu bar. Highlight TUTORIAL from your home directory, and select OK. A simple IIR low-pass filter will be designed as an example of how to use the tools. From the top menu bar, select Synthesis->Filter Type->IIR->Lowpass. Set the following parameters in the dialog box:

- **Sample Frequency:** 24000
- **Passband cutoff:** 2000
- **Stopband cutoff:** 10000
- **Passband ripple:** -1 dB
- **Stopband ripple:** -36 dB
- **Filter type:** Butterworth
- **Quantization:** 10

Click Estimate, and then select the Butterworth design filter of order 2, followed by OK. Select OK on the main filter specifications dialog box. Some graphs of the design will appear. Look through the graphs, and then exit when finished by closing the main window.
Design Architect

Versions:
analog_da (ANALOG/DA Personality Module) v8.4_1.1
DSP Architect v8.4_2.3
quickvhdl_da (QuickVHDL/DA Personality Module) v8.4_1.1
syn_blocks_da (AutoLogic BLOCKS/DA Personality Module) v.8.4_1.4
vhdlwrite_da (VHDLwrite/DA Personality Module) v.8.4_1.8
Design Architect v8.4_1.3
EDDM v8.4_1.2
Falcon Framework v8.4_1.1

To invoke Design Architect, type

> da

Click on the “OPEN DFL” icon from the session_palette. Enter “~/TUTORIAL/DFL” as the component, and select OK. A new window will be opened for edit. To import the DFL used in the FILlab program, use the top menu bar to select File->Import. Move to the directory ~/TUTORIAL/default_fillab, and select fillab_dfl. Select OK. The DFL description should be loaded into the editor. As is, this will causes some errors for Mistral1 because of some current limitations in the features supported. Specifically, Mistral1 cannot add two signals of different sizes. Thus, the DFL must be modified. Also, the accumulator width is to be reduced to simplify the complexity for the tutorial.

First, modify the AW constant from fix<32,31> to fix<20,19>. This accumulator width will be acceptable for the purpose of introducing the user to the tools. Next, the OUT definition must be changed to the following:

\[ \text{OUT} = \text{OW(second_order_section(AW(IN),C1[]))}; \]

This change casts the output to the output wordsize (OW). This also causes the input to get cast to the accumulator width so that Mistral1 will always add numbers of the same size wordlength and type. The final change involves changing the second_order_section definition to the following:

\[ \text{func second_order_section(IN : AW; C : CW[5]) OUT : AW} = \]

This causes the inputs and outputs to the biquad-section to be of the same size as the accumulators.
The final DFL description should read as follows:

```c
#define IW unsigned<8,6>
#define OW unsigned<8,6>
#define CW fix<10,8>
#define AW fix<20,19>

func IIR_CASC_DF_II( IN : IW ) OUT : OW =
begin
  /*---------------------------------------------*/
  /* Definition of the coefficient arrays */
  /* of the second-order sections */
  C1[] = CW({8.98437500E-02,
             1.75781250E-01,
             8.98437500E-02,
             1.03515625E+00,
             -3.67187500E-01});

  /*---------------------------------------------*/
  /* Description of the series of */
  /* second-order sections */
  OUT = OW(second_order_section(AW(IN),C1[]));
end;

func second_order_section(IN : AW ; C : CW[5]) OUT : AW =
begin
  Mnl = AW(C[0] * A1);
  Mn2 = AW(C[1] * A101);
  Mn3 = AW(C[2] * A102);
  Md1 = AW(C[3] * A101);
  Md2 = AW(C[4] * A102);
  OUT = OW(Mnl + Mn2 + Mn3);
  A1 = IW(Md1 + Md2 + IN);
end;
```

To compile the DFL, select Compile->Compile from the menu at the top of the screen. There should be one warning about missing initializations. Mistral1 will not use the initializations, so the DFL will not be modified to add initializations that will only be ignored later. Select File->Save, and then exit Design Architect.
To invoke DSPlab, type

> dsplab

After DSPlab is loaded, select “TSIM” from the palette to begin a time-domain simulation, and when the dialog box for the component name appears, enter the DFL name “TUTORIAL/DFL.” Select OK. Enter “24000” (Hz) as the session sampling frequency; then select OK. The time-domain setup dialog box will appear. Select “Cancel” for now so that some waveforms can first be created. Note that the Mentor Graphics palette options at the right of the screen change as different windows are “active” on the screen. When the time-domain simulation window is active, the “SETUP” icon may be selected at any time to bring up the current setup dialog box that was just closed.

To generate waveforms in DSPlab, select in the background of DSPlab to make the background active. This will bring up the main palette. Select the “WFGEN” icon to open the WFGEN palette. Select the “SINE” icon to create a sine wave. Enter the name that will be used in the waveform database, along with the amplitude, frequency, DC offset, and sampling frequency. For this tutorial, first create a sine wave named “tut_sine_1000,” with frequency 1000, sampling frequency 24000, and select OK. Create another sine wave named “tut_sine_11000,” with frequency 11000, sampling frequency 24000, and select OK.

From the menu at the top of the screen, select “Waveform->Expression Editor” to bring up an editor in which waveforms can be combined. A C-like expression can be compiled to create a waveform that is a function of other waveforms. In the editor, type

\[
tut_{\text{in}} = 0.1 \times \text{tut}_\text{sine}_\text{1000} + 0.1 \times \text{tut}_\text{sine}_\text{11000} + 0.2;
\]

The 0.1 multiplication factors are chosen to be small so that the accumulators do not overflow. If designing a practical filter, then the accumulators must be chosen to prevent overflow. The 0.2 factor is added in the waveform so that the signal is positive because the DFL used unsigned inputs.
After entering the expression, press and hold the right mouse button from within the expression editor window, and select “Parse.” If there are errors, the editor will complain; otherwise, it will inform the user that the waveform “tut_in” has been stored to the database.

Select the time-domain window to make it the active window. Select the “SETUP” icon in the palette to bring up the configurations. The new waveform can be entered as the input to the filter by selecting the “Select Stim” button in the Setup dialog box. Select the appropriate waveform database “dsplab,” select “Explore,” and then select the waveform “tut_in,” followed by OK. The “Model” in the setup dialog box can be set to “High-level,” “Bit-true,” or “Both.” This should be set to “Both.” Select “RUN” from the setup dialog when ready to perform the simulation.

When the simulation completes, two results libraries will be loaded. To display the results graphically, select the “XY” icon from the palette. This will open a window that waveforms can be displayed in. Select “ADDWF” icon, and then select the “hl_tsim_default_dsp_DFL” database, followed by OK. Select the waveform choice, followed by OK. This waveform, which is the high-level simulation waveform, will be displayed graphically in the active XY Tool window. Next, we will overlay the bit-true simulation waveform. Select the “ADDWF” icon, followed by “Up,” then “bt_tsim_default_dsp_DFL” and OK, and finally, the waveform and OK. Note that truncation causes a variance between the high-level simulation and the bit-true simulation. When finished with the time-domain simulation, close the time-domain simulation window. This will free up memory for the system so that the frequency-domain simulation can run.

To perform the frequency-domain simulation, select FSIM from the main palette. After entering the component name, which should remain the same, select OK, and a setup dialog will appear. This will sweep through frequencies up to half the sampling frequency, or the Nyquist rate, unless specified otherwise. The number of points can be modified to be 12000 with “linear” selected to test the filter in steps of 1 Hz. Select “RUN” from the setup dialog to run the simulation. Upon completion, the results database will be loaded, and the result can be viewed once again using the XY Tool. Note that it is possible to zoom in and out of the XY
Tool using the menu activated by holding down the right mouse button from within the XY Tool window. To exit the frequency-domain simulation, close the active simulation window.

**Mistral1**

*Versions:*
Mistral 1 v8.4_2.2  
Falcon Framework v8.4_1.1

To invoke Mistral1, type

```
> mistral1
```

From the menu bar at the top of the screen, select Design->Select, and choose TUTORIAL/DFL, followed by OK. Select Synthesis->Default Run. Mistral1 will run through all the design steps needed to generate the VHDL and will store the VHDL code. When running, Mistral1 will create two scan chains, with scan chain #1 having 165 flip-flops and scan chain #2 having 166 flip-flops. When finished, select Evaluate->Expanded CFG. Then, select “Full circuit overview,” followed by OK. This lists the potential, which is the number of delay stages for the given signal. Notice that the input IN is at potential 0, and the output OUT is at potential 26. The wordlength is of size 28. This informs the user how the timing works for the VHDL. A new input is started every 28 clock cycles, and the output for a given input is valid after 26 clock cycles from when the LSB of the input was loaded. Exit Mistral1, saving the database.

To prepare for compilation, the VHDL code that was generated will be moved to another directory for convenience. From the home directory, type:

```
> mkdir TUTORIAL/MISTRAL1_VHDL
> cp TUTORIAL/DFL/default_dsp/default_mistrall/DFL.hdl TUTORIAL/MISTRAL1_VHDL
> cd TUTORIAL/MISTRAL1_VHDL
> hdl ~/TUTORIAL/MISTRAL1_VHDL/DFL.hdl -MAP ms1_vhdl_lib
    /home/idea_lib/ms1_vhdl_lib -synthesis
```
Note that the msl_vhdljib may be located elsewhere if not using the computer engineering system network at RIT. Next, the ~/.cshrc file should be updated to prepare for the 1.2 micron CMOSN design. Add the following lines to the end of the .cshrc file:

```
setenv CMOSN_TECHNOLOGY onedot2_micron_worst
setenv PATH /home/idea_lib/cmosn_lib/bin:$PATH
```

Note once again that the path update may be different if not using the RIT computer engineering network. To rerun the .cshrc file so that these variables become updated, type:

```
> source ~/.cshrc
```

### AutoLogic

**Versions:**

- AutoLogic v8.4_1.2
- EDDM v8.4_1.2
- Falcon Framework v8.4_1.1

To invoke AutoLogic, type

```
> cmosn_autologic
```

Enter the component TUTORIAL/MISTRAL1_VHDL/dfl. Select Setup->Set Destination Technology, and verify that 1.2 micron worst technology is highlighted; select OK. Select Setup->Set Schematic Output, and click on “# of sheets” so that the schematic will appear on one sheet. Next, select Optimize->Set Opt Constraints->Set Hierarchy Controls. Change the Flatten parameter to “Yes.” Select Optimize->Set Opt Recipes. When the dialog box appears, select Add, followed by Area Report, then Add, then Area, then Add, and once again Area Report, then Add. Select Done, followed by OK.

To synthesize the design, select Synthesize->Synthesize design. Synthesize the design to “Technology,” and select the output format to be a schematic. Select OK. This will run for up to an hour or so. When the run is complete, hold down the right mouse button, and select “Export Transcript.” Type “AUTOLOGIC_TRANSCRIPT” as the file to save it to, followed by OK. Select File->Save Design Ypt to save the design. If one would like to view the schematic, select File->Open->Sheet. After looking at the schematic, exit AutoLogic.
Design Architect

Versions:
- analog_da (ANALOG/DA Personality Module) v8.4_1.1
- DSP Architect v8.4_2.3
- quickvhdl_da (QuickVHDL/DA Personality Module) v8.4_1.1
- syn_blocks_da (AutoLogic BLOCKS/DA Personality Module) v8.4_1.4
- vhdlwrite_da (VHDLwrite/DA Personality Module) v8.4_1.8
- Design Architect v8.4_1.3
- EDDM v8.4_1.2
- Falcon Framework v8.4_1.1

Invoke Design Architect by typing:

> cmosn_da

Select the “OPEN SHEET” icon in the session_palette. Enter the name “TUTORIAL/SYNTH_SHEET,” followed by OK. Select the “CHOOSE SYMBOL” icon, then select TUTORIAL/MISTRAL1_VHDL/dfl, followed by OK. Place the part using the left mouse button.

Refer to the final page in this section, which contains the schematic to be created. Select Libraries->CMOSN_LIB from the menu bar at the top of the screen. Select “BY TYPE” from the palette menu. Select “1.2 u pads.” Place the gnd12 and pwr12 pads above the symbol. These do not need to be connected to anything. Select and place four opad12 pads for the four output signals, and select and place six x12ipd pads for the six input signals. Holding the right mouse button down in the palette menu, select Back. Then select Misc. Here, the portin and portout nets can be placed as shown in the circuit on the final page of this section. Connect the wiring as shown. To change the name of the nets, move the cursor over the name to be changed, and select SHIFT-F7; then, enter the new name, followed by <ENTER>. When done, select Check->Sheet. There will be 1 warning resulting from 14 unconnected pins. Select File->Save Sheet to save the sheet, and then exit Design Architect.
Design Viewpoint Editor

Versions:
DVE v8.4_1.2
EDDM v8.4_1.2
Falcon Framework v8.4_1.1

To invoke DVE, type:

> cmosn_dve

Select the “OPEN VPT” icon from the palette menu. Enter TUTORIAL/SYNTH_SHEET, followed by OK. Select Setup->CMOSN Simulation, followed by OK to load the default simulation settings. Select Miscellaneous->Check Design->Check Option. Add the ERC checks option. Select OK. When completed, select File->Save Design Viewpoint. Exit DVE when done.

Note that the QuickSim II simulation and QuickPath analysis will not be performed in this tutorial. It will be assumed that the high-level simulation is the simulation that is important since the results of the thesis project seemed to indicate that the VHDL code generated by Mistral1 functions properly when simulated at a low-level.

IC Station

Versions:
ICgraph v8.4_2.1
EDDM v8.4_1.2
Falcon Framework v8.4_1.1

To invoke the IC Station package, type:

> cmosn_ic

Select the “Create” icon from the palette. Enter the following information:

Cell Name: TUTORIAL/IC_CELL
Attach Library: $MOSISCMOS
Process: $MOSISCMOS
Logic Source: TUTORIAL/SYNTH_SHEET/default

Select CE, then Eddm, then Flat. Enter TUTORIAL/SYNTH_SHEET as the viewpoint directory. Select OK when done. Select File->Load rules; enter $MOSISLIB/cmosn.rules.
Select place & route, then autofloorplan, followed by OK. Press shift-F8 when completed to view all. Select Ports, followed by OK to place the pads. Select Std cells, followed by OK to place the standard cells from the CMOSN library. Select All under the Autoroute heading followed by OK to perform the routing between the standard cells. When completed, select File->Cell->Save Cell.

To check the layout for DRC errors, select Back from the palette menu. Select IC Rules, followed by Load Rules with the file $MOSISLIB/cmosn.rules. Select Check, followed by OK. The total results will list the number of design rule violations present in the design.