2010

An Automated Approach to a 90-nm CMOS DRFM DSSM Circuit Design

Thomas A. Hopkins
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Electrical and Computer Engineering Commons

Repository Citation
Browse all Theses and Dissertations. 371.
https://corescholar.libraries.wright.edu/etd_all/371

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
AN AUTOMATED APPROACH TO A 90-NM CMOS DRFM DSSM CIRCUIT
DESIGN

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Engineering

By

THOMAS A. HOPKINS
B.S.E., Messiah College, 1994

2010
Wright State University

John M. Emmert, Ph.D.
Thesis Director

Kefu Xue, Ph.D.
Department Chair

Committee on Final Examination

John M. Emmert, Ph.D.

Ray M. Siferd, Ph.D.

Saiyu Ren, Ph.D.

Andrew T. Hsu, Ph.D.
Dean, School of Graduate Studies
Abstract


A digital single sideband modulator (DSSM) for a digital radio frequency memory (DRFM) was designed and implemented in a commercial 90-nm radiation-hardened-by-design (RHBD) structured ASIC by Thomas Pemberton in [1]. This thesis synthesized the same DSSM structure in a non-hardened 90-nm commercial process and compared the synthesis results of the two for power, delay, and area. The number of I/O bits and taps in [1] and this thesis were purposely made high to create a large target for radiation testing. As should be expected, the RHBD DSSM reported greater power and area. However, the RHBD power models were only estimates.

This thesis also showed the costs and benefits for varying bit widths, number of filter taps, and ROM sizes in the DSSM, synthesized at a typical characterization corner. One of the designs was also synthesized at two more characterization corners. Finally, another design variation was tested with extra piping in the Hilbert filter. All of these circuits were measured for power, timing, critical path, area, and spur-free dynamic range (SFDR).

Chip area was found to be solely dependent on the number of I/O pads and thus went up with greater I/O bit widths. Greater I/O bit widths and number of taps also led to more cell area and power consumption. The 16-bit/153-tap and 24-bit/101-tap typical-corner DSSM’s and the 16-bit/101-tap slow-corner DSSM could not meet the synthesis target of 100 MHz. Setup circuitry for the ROM address became the critical path for some of the designs partly due to the fact that the address was set up on the falling edge of the clock but loaded on the rising edge. Increasing I/O bit widths and the number of filter taps improved frequency response and SFDR.
Finally, increasing ROM size increased maximum SFDR for a select range of input frequencies. For SFDR, the predominant spur was the suppressed sideband, which was poorly suppressed.
# Table of Contents

Abstract ................................................................. ii

List of Figures ........................................................... viii

List of Tables ............................................................. xi

Acknowledgements ......................................................... xii

Dedication .................................................................. xiii

I. Introduction ................................................................. 1
   1.1 Motivation ............................................................ 1
   1.2 Project Goal .......................................................... 2
   1.3 Design and Research Approach ................................. 2
   1.4 Literature Overview ............................................... 3
   1.5 Document Organization ........................................... 4

II. Background and Theory ................................................. 5
   2.1 Single Sideband Modulation ...................................... 5
      2.1.1 Single Sideband Modulator ................................. 6
   2.2 Digital Radio Frequency Memory (DRFM) ...................... 7
      2.2.1 DRFM Architecture .......................................... 8
      2.2.2 DRFM Speed Techniques ................................. 9
   2.3 Digital and Discrete-Time Signal Processing ................. 11
      2.3.1 Quantization ................................................. 11
      2.3.2 Sampling Theorem ........................................ 12
      2.3.3 Aliasing ...................................................... 12
      2.3.4 Convolution Sum ........................................... 13
      2.3.5 Practical & Causal Filters ............................... 14
      2.3.6 Finite Impulse Response Filters ......................... 14
      2.3.7 Infinite Impulse Response Filters ...................... 15
      2.3.8 Window Functions .......................................... 15
   2.4 Digital Single Sideband Modulator (DSSM) ................. 15
      2.4.1 Hilbert Filter .............................................. 16
      2.4.2 Digital Mixers and Adders ............................ 18
   2.5 Ionizing Radiation ................................................ 18
   2.6 Radiation Effects .................................................. 20
      2.6.1 Total Dose Effects ........................................ 20
<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.2 Single Event Effects (SEE’s)</td>
</tr>
<tr>
<td>2.6.2.1 Parasitic Bipolar Amplification</td>
</tr>
<tr>
<td>2.6.2.2 Charge Sharing</td>
</tr>
<tr>
<td>2.6.2.3 Single Event Upsets (SEU’s)</td>
</tr>
<tr>
<td>2.6.2.4 Single Event Transients (SET’s)</td>
</tr>
<tr>
<td>2.7 Radiation Hardening CMOS Integrated Circuits</td>
</tr>
<tr>
<td>2.8 Radiation Hardened by Design (RHBD) Methods</td>
</tr>
<tr>
<td>2.8.1 Total Dose RHBD Methods</td>
</tr>
<tr>
<td>2.8.2 Single Event RHBD Methods</td>
</tr>
<tr>
<td>2.9 Guard Rings</td>
</tr>
<tr>
<td>2.9.1 Guard Rings in RHBD</td>
</tr>
<tr>
<td>2.9.2 Guard Ring Costs</td>
</tr>
<tr>
<td>2.10 Measuring Ion Energy</td>
</tr>
<tr>
<td>2.11 Testing Circuits for SET’s</td>
</tr>
</tbody>
</table>

### III. Implementation

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 DSSM Design Modules</td>
</tr>
<tr>
<td>3.1.1 Input Multiplexer</td>
</tr>
<tr>
<td>3.1.2 Hilbert Filter</td>
</tr>
<tr>
<td>3.1.3 Mixer</td>
</tr>
<tr>
<td>3.2 DSSM (DRFM) Inputs and Outputs</td>
</tr>
<tr>
<td>3.2.1 Input and Output (I/O) Pads</td>
</tr>
<tr>
<td>3.3 Design and File Naming Convention</td>
</tr>
<tr>
<td>3.4 Hardware Description Language (HDL)</td>
</tr>
<tr>
<td>3.5 Pre-Testing</td>
</tr>
<tr>
<td>3.6 Software Tools</td>
</tr>
<tr>
<td>3.7 Design Flow</td>
</tr>
<tr>
<td>3.7.1 Design File Generation</td>
</tr>
<tr>
<td>3.7.2 RTL-Code Simulation</td>
</tr>
<tr>
<td>3.7.3 RTL Synthesis</td>
</tr>
<tr>
<td>3.7.4 Gate-Level Netlist Simulation</td>
</tr>
<tr>
<td>3.7.5 Layout and Routing</td>
</tr>
<tr>
<td>3.7.6 Frequency Analysis</td>
</tr>
<tr>
<td>3.8 Technology Files</td>
</tr>
<tr>
<td>3.8.1 Characterization Corners</td>
</tr>
<tr>
<td>3.8.2 Transistor Threshold and RC Values</td>
</tr>
<tr>
<td>3.9 File Directory Structure</td>
</tr>
<tr>
<td>3.10 Test Cases and Metrics</td>
</tr>
<tr>
<td>3.11 HDL Files</td>
</tr>
</tbody>
</table>
3.11.1 Parameterization of the HDL Files ............... 47
3.12 VHDL Test Bench Files ................................. 49
3.13 Automation of Design Flow .............................. 49
   3.13.1 RTL Compiler Script ............................... 50
   3.13.2 First Encounter® Script ............................ 50
   3.13.2.1 First Encounter® Design Files .................. 51
   3.13.3 Simulation Script ................................ 52
   3.13.4 MATLAB® Scripts .................................. 52
   3.13.4.1 Mathematical Model of the DSSM ............... 54
3.14 Rad-Hard Circuit Comparison ............................ 54

IV. Results and Analysis .................................... 56
   4.1 Rad-Hard Comparison .................................. 58
   4.2 I/O Bits Comparison (8, 16, and 24) .................. 60
   4.3 Coefficient Bits Comparison (14 and 16) .............. 63
   4.4 Filter Taps Comparison (33, 101, and 153) .......... 65
   4.5 ROM Size Comparison (128, 256, and 512) .......... 68
   4.6 Summary of Design Parameter Variations ............... 71
      4.6.1 Power ........................................... 72
      4.6.2 Timing .......................................... 72
         4.6.2.1 Critical Paths ............................... 72
      4.6.3 Area ........................................... 73
      4.6.4 SFDR ........................................... 73
      4.6.5 Frequency Response ............................... 74
   4.7 Three Process Corners Comparison (ss, tt, and ff) ... 76
   4.8 Piped Hilbert Filter Comparison (24-bit) ............. 80
   4.9 Time and Frequency Analysis ........................... 83
      4.9.1 SFDR Plots of Models ............................. 89
      4.9.2 Frequency Response Plots of Models .............. 94

V. Conclusions and Future Work .............................. 98
   5.1 Rad-Hard Comparison .................................. 98
   5.2 Parameters ........................................... 98
      5.2.1 Input Bits ...................................... 98
      5.2.2 Coefficient Bits ................................ 99
      5.2.3 Number of Taps ................................ 99
      5.2.4 ROM Size ....................................... 99
   5.3 Area and Power ....................................... 99
   5.4 Timing ............................................... 100
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Double and Single Sideband Modulation</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Single Sideband Modulator</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>I and Q Signals</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>DRFM Architecture with Analog I and Q</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>DRFM Architecture with Digital I and Q</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>1200 Hz Aliased as 800 Hz</td>
<td>12</td>
</tr>
<tr>
<td>2.7</td>
<td>Aliasing from Frequency Overlap</td>
<td>13</td>
</tr>
<tr>
<td>2.8</td>
<td>Direct-Form Implementation of an FIR Filter</td>
<td>14</td>
</tr>
<tr>
<td>2.9</td>
<td>Digital Single Sideband Modulator (DSSM)</td>
<td>16</td>
</tr>
<tr>
<td>2.10</td>
<td>DSSM with Hilbert Transformer</td>
<td>16</td>
</tr>
<tr>
<td>2.11</td>
<td>Hilbert Transform Magnitude Frequency Response</td>
<td>17</td>
</tr>
<tr>
<td>2.12</td>
<td>Hilbert Discrete-Time Impulse Response</td>
<td>17</td>
</tr>
<tr>
<td>2.13</td>
<td>FIR Hilbert Filter</td>
<td>18</td>
</tr>
<tr>
<td>2.14</td>
<td>Cosmic Radiation Cascade Effect</td>
<td>19</td>
</tr>
<tr>
<td>2.15</td>
<td>SET Cross Section</td>
<td>21</td>
</tr>
<tr>
<td>2.16</td>
<td>SET Propagation to a Latch</td>
<td>23</td>
</tr>
<tr>
<td>2.17</td>
<td>Spatial Redundancy</td>
<td>26</td>
</tr>
<tr>
<td>2.18</td>
<td>Example of Temporal Redundancy, A Temporal Latch</td>
<td>26</td>
</tr>
<tr>
<td>2.19</td>
<td>Guard Rings Around Inverter PMOS &amp; NMOS Transistors</td>
<td>28</td>
</tr>
<tr>
<td>2.20</td>
<td>SET Cross Section</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Top Level Module - DSSM</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>A 101-Tap FIR Hilbert Filter</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Mixer Module</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Direct Digital Synthesis (DDS) in ROM Wrapper</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Design and File Naming Convention</td>
<td>38</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>3.6</td>
<td>Design Flow</td>
<td>39</td>
</tr>
<tr>
<td>3.7</td>
<td>Cadence SimVision® 6.11 Output</td>
<td>40</td>
</tr>
<tr>
<td>3.8</td>
<td>SoC Encounter® After Floor Plan</td>
<td>42</td>
</tr>
<tr>
<td>3.9</td>
<td>SoC Encounter® After Placement</td>
<td>42</td>
</tr>
<tr>
<td>3.10</td>
<td>SoC Encounter® After Power Routing</td>
<td>43</td>
</tr>
<tr>
<td>3.11</td>
<td>SoC Encounter® After Clock and Signal Routing</td>
<td>43</td>
</tr>
<tr>
<td>3.12</td>
<td>File Directory Structure</td>
<td>45</td>
</tr>
<tr>
<td>3.13</td>
<td>FFT of Input and Output of DRFM</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Layout for DRFM</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Example Freq. Plot for DRFM</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>SFDR for 8, 16, and 24 I/O Bits</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Freq. Response for 8, 16, and 24 I/O Bits</td>
<td>61</td>
</tr>
<tr>
<td>4.5</td>
<td>Layouts for 8, 16, and 24 I/O Bits</td>
<td>62</td>
</tr>
<tr>
<td>4.6</td>
<td>SFDR for 14 and 16 Bit Filter Coefficients</td>
<td>63</td>
</tr>
<tr>
<td>4.7</td>
<td>Freq. Response for 14 and 16 Bit Filter Coefficients</td>
<td>64</td>
</tr>
<tr>
<td>4.8</td>
<td>Layouts for 14 and 16 Coefficient Bits</td>
<td>64</td>
</tr>
<tr>
<td>4.9</td>
<td>SFDR for 33, 101, and 153 Filter Taps</td>
<td>66</td>
</tr>
<tr>
<td>4.10</td>
<td>Freq. Response for 33, 101, and 153 Filter Taps</td>
<td>66</td>
</tr>
<tr>
<td>4.11</td>
<td>Layouts for 33, 101, and 153 Filter Taps</td>
<td>67</td>
</tr>
<tr>
<td>4.12</td>
<td>SFDR for 128, 256, and 512 ROM Words</td>
<td>69</td>
</tr>
<tr>
<td>4.13</td>
<td>Freq. Response for 128, 256, and 512 ROM Words</td>
<td>69</td>
</tr>
<tr>
<td>4.14</td>
<td>Layouts for 128, 256, and 512 ROM Words</td>
<td>70</td>
</tr>
<tr>
<td>4.15</td>
<td>SFDR for All Designs</td>
<td>74</td>
</tr>
<tr>
<td>4.16</td>
<td>Frequency Response for All Designs</td>
<td>75</td>
</tr>
<tr>
<td>4.17</td>
<td>SFDR for Slow, Typical, and Fast Corners</td>
<td>77</td>
</tr>
<tr>
<td>4.18</td>
<td>Freq. Response for Slow, Typical, and Fast Corners</td>
<td>78</td>
</tr>
<tr>
<td>4.19</td>
<td>Layouts for Three Characterization Corners</td>
<td>79</td>
</tr>
<tr>
<td>4.20</td>
<td>SFDR for Unpiped Vs. Piped Hilbert Filter</td>
<td>81</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>4.21</td>
<td>Freq. Response for Unpiped Vs. Piped Hilbert Filter</td>
<td>81</td>
</tr>
<tr>
<td>4.22</td>
<td>Layouts for DRFM_{24,24,101,256}, Unpiped and Piped</td>
<td>82</td>
</tr>
<tr>
<td>4.23</td>
<td>FFT of Cosine in DRFM_{16,16,101,256}</td>
<td>83</td>
</tr>
<tr>
<td>4.24</td>
<td>DSSM Input Vs. Output for DRFM_{16,14,101,256}</td>
<td>84</td>
</tr>
<tr>
<td>4.25</td>
<td>FFT of DSSM Input and Output from Figure 4.24</td>
<td>85</td>
</tr>
<tr>
<td>4.26</td>
<td>FFT of DRFM_{8,8,101,256}</td>
<td>87</td>
</tr>
<tr>
<td>4.27</td>
<td>Single-Input Freq. Response for DRFM_{16,16,101,256}</td>
<td>88</td>
</tr>
<tr>
<td>4.28</td>
<td>FFT of DRFM_{24,24,101,256}</td>
<td>88</td>
</tr>
<tr>
<td>4.29</td>
<td>SFDR for 101-Tap Design Models</td>
<td>89</td>
</tr>
<tr>
<td>4.30</td>
<td>SFDR for 33-/153-Tap Design Models</td>
<td>90</td>
</tr>
<tr>
<td>4.31</td>
<td>SFDR Closeup of DRFM_{8,8,101,256} Models</td>
<td>90</td>
</tr>
<tr>
<td>4.32</td>
<td>SFDR Closeup of DRFM_{16,16,101,256} Models</td>
<td>91</td>
</tr>
<tr>
<td>4.33</td>
<td>SFDR Closeup of DRFM_{16,16,153,256} Models</td>
<td>91</td>
</tr>
<tr>
<td>4.34</td>
<td>FFT of Math Model for DRFM_{16,16,153,256}</td>
<td>93</td>
</tr>
<tr>
<td>4.35</td>
<td>Freq. Response for 101-Tap Design Models</td>
<td>95</td>
</tr>
<tr>
<td>4.36</td>
<td>Freq. Response for 33-/153-Tap Design Models</td>
<td>96</td>
</tr>
<tr>
<td>4.37</td>
<td>Freq. Response Closeup of DRFM_{8,8,101,256} Models</td>
<td>96</td>
</tr>
<tr>
<td>4.38</td>
<td>Freq. Response Closeup of DRFM_{16,16,101,256} Models</td>
<td>97</td>
</tr>
<tr>
<td>4.39</td>
<td>Freq. Response Closeup of DRFM_{16,16,33,256} Models</td>
<td>97</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Test Cases</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Rad-Hard Comparison</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Results for 8, 16, and 24 I/O Bit Widths</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>Results for 14- and 16-bit Filter Coefficients</td>
<td>63</td>
</tr>
<tr>
<td>4.4</td>
<td>Results for 33, 101, and 153 Filter Taps</td>
<td>65</td>
</tr>
<tr>
<td>4.5</td>
<td>Results for 128, 256, and 512 ROM Words</td>
<td>68</td>
</tr>
<tr>
<td>4.6</td>
<td>Results Summary by Increasing Power (part 1 of 2)</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>Results Summary by Increasing Power (part 2 of 2)</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>Critical Paths for All Designs</td>
<td>71</td>
</tr>
<tr>
<td>4.8</td>
<td>Results for Slow, Typical, and Fast Corners</td>
<td>76</td>
</tr>
<tr>
<td>4.9</td>
<td>Critical Paths for Slow, Typical, and Fast Corners</td>
<td>76</td>
</tr>
<tr>
<td>4.10</td>
<td>Results for DRFM.24.24.101.256, Unpiped and Piped</td>
<td>80</td>
</tr>
<tr>
<td>4.11</td>
<td>Critical Paths for DRFM.24.24.101.256, Unpiped and Piped</td>
<td>80</td>
</tr>
</tbody>
</table>
Acknowledgements

I thank the living God for opening the door of opportunity for this thesis and for granting success in creating and completing it. Thank you to the NEWSTARS (New Electronic Warfare Specialists Through Advanced Research by Students) program for funding this work. Thank you to Dr. J. Marty Emmert for informing me of the research program, helping me in the selection of the topic, training me, and helping guide the work. Thank you to Thomas Pemberton for his technical help and consultation in the work. A thanks especially goes to Wright State University for providing the academic setting and logistical support for the research. Thank you to my parents Allan and Arlene for their support of my pursuits. I also thank numerous believers in Christ, both locally and at a distance, who supported me during this work. Finally, thank you to Dr. J. Marty Emmert, Dr. Ray Siferd, and Dr. Saiyu Ren for chairing the thesis committee.

Thomas A. Hopkins
Dedication

This thesis is dedicated to
my parents
who gave me physical life
and to
the Lord Jesus Christ
who gave me eternal life.
I. Introduction

1.1 Motivation

Digital signal processing (DSP) of former analog functions, such as a single side-band modulator (SSM) in a digital radio frequency memory (DRFM), requires speeds that may now be feasible \([2]\). These DSP circuits also are generally implemented with CMOS technology.

The IC design process has many steps. Presently, these steps are automated with electronic design automation (EDA) software tools and accompanying scripts. In this thesis, they are automated with Cadence design and simulation tools and scripts.

Radiation effects on circuitry at terrestrial, flight, and orbital altitudes are a concern. As IC technologies get smaller and faster, single event transients (SET’s) will become a more prominent cause of single event upsets (SEU’s) for circuits with radiation hardened memories. Radiation hardened fabrication facilities are up to three technology generations behind their commercial counterparts. However, using radiation hardening by design (RHBD) methods alone has become a viable design method. This is also true for space applications where a serendipitous natural total-dose hardening of smaller transistor sizes has occurred. The use of RHBD methods can be applied using commercial processes and still have a performance gain over its
rad-hard processed counterparts. In this thesis, an implementation of a digital single sideband modulator (DSSM) in an RHBD and non-rad-hard circuit are compared.

Because DSP circuits mostly use CMOS technology, much of RHBD focuses on CMOS processes. One RHBD method that is easy to implement is guard rings. The Wright State University Automation and Design Lab may also have a desire in the future to test guard rings.

1.2 Project Goal

The goal was to synthesize a non-radiation hardened version of a DSSM with the same structure as in the RHBD structured ASIC in [1] and compare for common metrics of power, delay, area, SFDR, and chip utilization. The number of I/O bits and filter taps were unusually large in order to fill up the chip area in [1] and provide a large target for radiation testing. The non-hardened DSSM was not sent out for radiation testing. An ancillary goal in this thesis was to research the use of guard rings in RHBD.

The HDL code for the DSSM needed to be parameterizable to allow for automated, or semi-automated, changes in parameters. In addition to the rad-hard comparison case, seven other test cases were to be synthesized with different input bits, number of filter taps, and ROM sizes. The test cases were to be compared to each other for power, delay, area, and SFDR. Synthesis was to be done with Cadence EDA tools via modified versions of existing scripts. The scripts would allow for some automation of the synthesis.

1.3 Design and Research Approach

Our design approach was to consult with Thomas Pemberton and make a DSSM design as close as possible to his [1]. Some VHDL code was taken from generators provided by Pemberton. The rest of the VHDL code was reused and modified from Pemberton’s design model on an FPGA. The design software and scripts necessary to
synthesize the design up to the place-and-route (PAR) level were learned. Next, the designs were implemented, and the command scripts were adjusted as needed. The scripts and top-level test bench were simplified for ease of use. MATLAB® code obtained from Pemberton was also modified and used for plotting single-input frequency response, swept input frequency response, and SFDR. Finally, a mathematical model of the DSSM was coded and plotted in MATLAB®.

Research into RHBD methods, especially guard rings with their costs and weaknesses, started early on and continued during the project. A guard ring design and a testing plan were proposed. Research was also done into articles and material on DRFM, DSSM’s, and other background material.

1.4 Literature Overview

As already mentioned, former analog functions, e.g. a DSSM in a DRFM, may be able to be done digitally. An early paper on utilizing a DSSM for a DRFM can be found in [3]. The operation of a general SSM can be found in a textbook like [4].

The principle used in a DRFM has been around since 1974 [5]. Information on DRFM’s can be found in books on digital receivers and electronic warfare (EW) as well as in technical articles (see [5, 6]).

Research into radiation effects in IC’s began in earnest around 1978 [7]. IBM published a series of papers on its findings in 1996, including [7–9]. The Aerospace Corporation has also done much in this field. Ron Lacoe, of Aerospace Corp., is an expert on RHBD and has written a good introduction in [10]. The electrical engineering department of Vanderbilt University in Nashville, Tennessee (including Daniel M. Fleetwood, Lloyd W. Massengill, Ronald D. Schrimpf, Barat L. Bhuva, W. Timothy Holman, and others) also has done much research into radiation effects and hardening. Much of the material on radiation hardening IC’s can be found in technical papers written by the above people and others.
1.5 *Document Organization*

Chapter 2 gives a brief background of single sideband modulation, DRFM design, digital signal processing, and RHBD. Chapter 3 covers design implementation, namely: design hierarchy, design steps, test cases, and the radiation hardened comparison. Chapter 4 summarizes results for power, delay, critical path, area, utilization, SFDR, and frequency response of the designs. Chapter 5 draws conclusions from the results and gives suggestions for future work. Finally, the VHDL code, test bench, and scripts can be found in the appendices.
II. Background and Theory

In this chapter we look at a digital single sideband modulator (DSSM) that can be used with a digital radio frequency memory (DRFM). A DRFM system is commonly used in military aircraft where it will experience higher levels of natural radiation than on the ground. Toward the end of the chapter, we look at protecting CMOS circuits, e.g. a DSSM and DRFM, against natural radiation and its effects. Particular attention is given to guard rings, especially their benefits, costs, and test procedures.

2.1 Single Sideband Modulation

Many practical signals are generated at baseband frequency, i.e. near zero hertz (see figure 1(a)). Most radio transmissions, however, require modulation of a baseband signal to a higher frequency where the antennae size is more practical (i.e smaller). Modulation from baseband also allows multiple signals to be transmitted at different frequencies [4].

![Double and Single Sideband Modulation](image)

(a) Baseband Signal  
(b) Double Sideband Modulation  
(c) Single Sideband Modulation

Figure 2.1: Double and Single Sideband Modulation

There are a few different modulation techniques. They fall under two categories: linear and angle modulation. Under linear modulation, there are amplitude,
double-sideband (DSB), and single-sideband (SSB) modulation. DSB modulation is accomplished by multiplying a baseband signal by a sine wave. This results in a sideband both above and below the carrier frequency (see figure 1(b)). These sidebands are redundant information, and only one is needed. At a cost of circuit complexity, SSB modulation removes one of these sidebands and reduces the bandwidth back by half (see figure 1(c) [4].

SSB modulation is useful in recombining an in-phase (I) and quadrature (Q) version of a signal as explained in section 2.1.1. Here, I is simply unshifted version of the input signal, and Q is a 90-degree phase-shifted version (see figure 2.3).

SSB modulation is also useful in creating a frequency shift. Sometimes a system, like a DRFM, needs to dynamically shift frequencies to create false Doppler frequency shifts. SSB modulation is useful in accomplishing this.

![Figure 2.2: Single Sideband Modulator](image)

2.1.1 Single Sideband Modulator (SSM). SSB modulation can be accomplished by removing one sideband from a DSB modulated signal. This can be done with just a low or highpass filter. However, for the general case, this would have to be a very selective filter.

As seen in figure 2.2, DSB modulation can also be performed on in-phase (I) and quadrature (Q) versions of a signal where the carrier signals (sine and cosine) are
also quadratures of one another. The resulting signals are then added or subtracted together to give a lower sideband or upper sideband result, respectively [4, 11].

With an analog SSM and varying input frequency, however, there is difficulty in maintaining phase and magnitude balance between the I and Q signals, and there is a tendency to get a DC offset. This causes unwanted spurs in the output [12]. On the other hand, a digital implementation of the SSM can give superior balance and stability to the I and Q and very little DC offset [6].

2.2 Digital Radio Frequency Memory (DRFM)

The DRFM is an electronic circuit that digitally captures and stores radio and microwave signals. These then can be read and retransmitted with a small delay. Digital storage, of course, is the storage method of choice and is more reliable than analog storage. Thus, as its name implies, the digital radio frequency memory (DRFM) uses digital storage methods [3, 5].
A common use of the DRFM is as a key component in a radar repeater jamming circuit, an application that has received much research [6]. In this application, the DRFM stores the signal which is then recreated with a small delay and/or Doppler frequency shift. This gives a deceptive range and speed, respectively, of the aircraft. The entire system of sampling, storing, altering, and recreating the signal is referred to as the DRFM system [5, 6].

2.2.1 DRFM Architecture. The DRFM concept is simple. However, as will be seen in section 2.2.2, the speed requirements placed upon it pose a challenge [6].

As shown in figure 2.4(a), the basic components of a DRFM system with analog quadrature detection are: a receiving antennae, two down-converters (ultimately to baseband), a low pass filter, an ADC, a digital memory, a DAC, another lowpass filter, two up-converters, and a transmitting antennae [6]. The innermost converters also include a quadrature detector and SSM as shown in figure 2.4(b) [5]. Both the I and Q channels are sampled and saved in memory. This requires two ADC’s and extra storage space. However, one benefit of having an analog I and Q signal, is that instead of only being able to distinguish up to half the sampling frequency (F_s) as with one ADC, the pair of ADC’s can distinguish input frequencies up to the full sampling frequency [2].

Digital implementation of the SSM could reduce hardware area. As seen in figure 2.5, only one down/up-converter and ADC/DAC combination are needed. Also, as mentioned in section 2.1.1, digitally-produced I and Q signals are more balanced and stable [6]. However, a digital implementation has been limited in its speed. This is because of technology speed limits and the fact that, with only one analog input signal, bandwidth can now only go up to half of F_s. However, with advances in

---

1Radar jamming is a subset of the military actions of “electronic attack” (EA), part of what is also known as “electronic countermeasures” (ECM). EA, in turn, is a subset of what is known as “electronic warfare” (EW) [6].

2The principle used in the DRFM is credited to an invention in 1974 by Chris Haynes of EMI Electronics Ltd [5].
technology, digital implementation of a high-speed digital SSM (DSSM) in a DRFM may now or soon be feasible [2].

2.2.2 DRFM Speed Techniques. Since modern radars create a wide bandwidth, DRFM’s need to monitor a wide instantaneous bandwidth (e.g. 1 GHz) [2]. This requires the DRFM to have high clock speeds to quickly sample and store the signal. One of the limiting factors on speed is the speed of the A/D (and D/A) converters. GaAs processes have been used to obtain speed gains in the past. Beyond this, increasing the speed of existing A/D converters generally means reducing the number of bits. This in turn adds quantization noise, thus making it more likely
that the jamming signal will be discovered. However, there are other techniques for increasing the bandwidth and speed [6].

One speed-gain technique uses a switched oscillator. Another involves multiple narrowband DRFM’s where each threat can be directed to a different DRFM. (Narrowband DRFM’s, of course, require less speed.) Another method uses only one bit sampling. This gives greater bandwidth at the cost of not distinguishing amplitudes. This method also avoids having to use an A/D converter. Next, there is series-parallel sampling. Finally, a DRFM’s bandwidth can be improved by using multiple receiver channels and parallel processing. One example of this uses 16 channels [13]. Any of
these methods can be used for speed gains [5,6]. As an example of the bit-width capability of a present-day (laboratory-based) DRFM, one has been advertised as having up to 1 GHz instantaneous bandwidth using 10 bits [14].

2.3 Digital and Discrete-Time Signal Processing

Digital and discrete-time signal processing is considered in this section. Digital, here, refers to the quantization of a signal. Discrete-time refers to the sampling of the signal. In digital/discrete-time signal processing, an analog signal is quantized and sampled at some rate, usually with a clock at equal intervals in an analog-to-digital (A/D) converter (ADC). The samples are then processed by the digital/discrete-time circuit. If desired, the results can be converted back into an analog signal via a digital-to-analog (D/A) converter (DAC).

2.3.1 Quantization. Quantization is the process of rounding an analog signal to discrete levels. A comparator-based ADC “floors” each analog value by rounding it to its next lowest quantization value (towards negative infinity). There are \(2^b\) levels for a signal representation of \(b\) bits. These levels in turn are usually evenly spaced between a minimum and maximum value of interest (e.g. -0.5 and 0.5, or 0 and 1).

Rounding for quantization means the quantized signal will have slight differences (errors) from the original signal. These errors translate into noise in the frequency spectrum known as quantization noise. The difference between a signal’s peak and the highest noise spur is known as spur free dynamic range (SFDR). Less quantization bits translate into more noise and potentially lower SFDR. Assuming no other noise sources, the SFDR (in decibels) can be calculated from equation 2.1 based on the number of bits, \(b\) [15].

\[
SFDR_{dB} = 6b
\]  
(2.1)
2.3.2 **Sampling Theorem.** In order to be able to faithfully reconstruct a sampled signal back into the analog domain, it is sufficient to sample the analog signal at a rate greater than twice the bandwidth of the signal. This rate is called the Nyquist rate \[4\]. If \(F_s\) is the sampling frequency, the frequency range: \([-F_s/2 \text{ to } F_s/2]\) is called the first Nyquist band. This band will keep repeating itself infinitely in both directions. So, for example, the next band in the positive direction is from \([F_s/2 \text{ to } 3F_s/2]\).

![Figure 2.6: 1200 Hz Aliased as 800 Hz](image)

2.3.3 **Aliasing.** Aliasing is the addition of unwanted signals in a Nyquist frequency band. This occurs when there are signals above half the sampling rate in the sample. The resulting frequency samples will “fold over” and appear as aliased signals in the Nyquist band. An example of this is shown in figure 2.6. Here just the positive (right) half of the Nyquist band is shown where a 1200 Hz signal has “folded over” to show up as 800 Hz. To prevent this, a lowpass (i.e. anti-aliasing) filter is placed before the ADC to remove frequencies that are above half the sampling rate \([11]\). However, as mentioned in section 2.2.1, with two quadrature versions of a analog signal, an I and a Q, this “fold over” can still be distinguished for up to twice the Nyquist bandwidth (i.e. for \(F_s\)). Therefore, the anti-aliasing filter can be widened to twice the bandwidth at \(F_s\).
Another form of aliasing comes from overlap in the frequency spectrum. In practical signal processing, input signals are not bandlimited (because they are truncated). This means the frequency spectrum “spreads” and “leaks” into the next Nyquist band, because the sampling theorem’s sufficiency “requirement” is not met. This leakage causes the repeated frequency bands to overlap, add to each other, and make a distortion (aliasing). This kind of aliasing can be reduced by increasing the sampling frequency. However, it can never be totally removed [11]. Figure 2.7 shows an 800-Hz sine wave sampled with just 256 samples at 2000 Hz. Aliasing can be seen where the “-800 MHz” from the next Nyquist frequency band (shown at 1200 MHz on the scale) overlaps with the +800 MHz in the first Nyquist frequency band (shown at 800 MHz on the scale).

2.3.4 Convolution Sum. If we know the impulse response of a discrete-time system and its input, we can calculate the output using the convolution sum given by equation 2.2. Here, k is the inputs sample number, and n is the impulse response’s sample number. The convolution sum gives the sum of delayed versions of the input multiplied by the impulse response. FIR and IIR filter equations 2.3 and 2.4 use a form of the convolution sum.
\[
y(n) = h(n) \ast x(n) \equiv \sum_{k=-\infty}^{\infty} h(k)x(n-k)
\] (2.2)

2.3.5 Practical & Causal Filters. A filter is causal if its output is solely dependent on present and past inputs. It cannot be dependent on future inputs. All practical/real-time filters are causal. A filter’s impulse response is made causal by shifting its impulse response forward in time so that its convolution sum starts at \(k = 0\). The result is that the response is delayed by half the length of the filter. (See equations 2.3 and 2.4.)

Practical filters also have a limited filter length. Therefore, in the FIR and IIR filter equations, 2.3 and 2.4, \(k\) has fixed limits. \(k\)'s infinite limits in equation 2.2 were solely an ideal case.

2.3.6 Finite Impulse Response Filters. A finite impulse response (FIR) filter utilizes only feed-forward components. It can be expressed by equation 2.3, which is a form of the convolution sum. Here, \(b_k\) is a coefficient, and \(M\) is the number of coefficients. Since there is no feedback, FIR filters are always stable. A direct-form structure implementing this equation is shown in figure 2.8 [16].

\[
y(n) = \sum_{k=0}^{M-1} b_kx(n-k)
\] (2.3)

![Figure 2.8: Direct-Form Implementation of an FIR Filter](image-url)
2.3.7 Infinite Impulse Response Filters. An infinite impulse response (IIR) filter uses feedback. An IIR filter usually can be much smaller than an FIR filter. Care must be taken, however, in designing an IIR filter to make sure it is stable. Equation 2.4 is its general equation where \( x \) is the input, \( y \) is the output, \( a_k \) is a feedback coefficient, and \( b_{-k} \) is a feedback coefficient. Also, \( N \) is the number of feedback coefficients, and \( M \) is the number of feed-forward coefficients [16].

\[
y(n) = \sum_{k=1}^{N} a_k y(n - k) + \sum_{k=0}^{M-1} b_k x(n - k) \tag{2.4}
\]

2.3.8 Window Functions. In general, whenever a sample train is truncated, it is said to be windowed. If the train is simply truncated with no attenuation, the window function is a rectangular window. However, truncation creates sharp discontinuities at the beginning and end of a sample train. This in turn creates undesirable ripples in the measured frequency response. These ripples can be minimized or removed by gradually attenuating the sample train from its center to its ends. The functions that determine this attenuation are also called window functions. The downside of attenuating a signal is that it lessens the frequency bandwidth. However, often the regions of reduced bandwidth are not used anyway, or they can just simply be accepted.

One common window function is called the Hanning window and is shown in equation 2.5, where \( n \) is the sample number, and \( M \) is the number of samples.

\[
h(n) = \begin{cases} 
\frac{1}{2} \left( 1 - \cos \frac{2\pi n}{M-1} \right), & 0 \leq n \leq M-1 \\
0, & \text{otherwise}
\end{cases} \tag{2.5}
\]

2.4 Digital Single Sideband Modulator (DSSM)

In the DRFM system with the digital quadrature detector and SSM shown in figure 2.5(b), if the DRFM is removed, the combination of the quadrature detector and SSM is also called an SSM, or digital SSM (DSSM), figure 2.9.
As shown in figure 2.10, the Q signal in the DSSM can be created using a Hilbert filter. This filter, however, creates an extra latency of half of one less the number of FIR filter taps. Therefore, the I signal is given the same latency to keep it in balance with the Q.

2.4.1 Hilbert Filter. The ideal Hilbert transform is an all-pass filter with a 90-degree phase shift. It’s frequency response can be mathematically represented by equation 2.6, where $\omega$ is the frequency in radians [16]. It’s magnitude response is shown in red in figure 2.11 where it is compared to the fast Fourier transform (FFT) of a practical Hilbert filter with 33 coefficients (in blue). Since the Hilbert is an all-pass filter, it’s magnitude is ideally 0 dBc for all frequencies. However, a practical filter shows ripples in its response (because it is unwindowed) and a reduced bandwidth.
Figure 2.11: Hilbert Transform Magnitude Frequency Response

Figure 2.12: Hilbert Discrete-Time Impulse Response

\[
H(\omega) = \begin{cases} 
-j & 0 < \omega \leq \pi \\
-j & -\pi < \omega < 0 
\end{cases} \quad (2.6)
\]

For a practical digital Hilbert filter, the discrete-time impulse response for an odd number of coefficients can be represented by equation 2.7, where \(b_k\) is a coefficient and \(k\) is the coefficient number [3]. The coefficients can be seen to calculate to zero for \(k\) even and \(2/\pi k\) for \(k\) odd. For 33 coefficients, a graph of this impulse response
would look like figure 2.12. As already mentioned, a graph of the FFT of this filter is in figure 2.11.

\[
h(k) = b_k = \begin{cases} 
\frac{2\sin^2(\pi k/2)}{\pi k} & |k| > 0 \\
0 & k=0
\end{cases}
\] (2.7)

Equation 2.7 can be implemented in direct FIR form as in figure 2.13 where \(M\) again represents the number of FIR filter coefficients. Since even coefficients are zero, only \((M-1)/2\) coefficients and taps are actually needed. The filter response coefficients should be attenuated with (i.e. multiplied by) a window function to remove ripples from the output frequency response [3]. Since the Hilbert is a linear filter with an antisymmetric response, if desired, half of the multipliers could be removed by subtracting the corresponding tapped inputs first and then multiplying [16].

2.4.2 Digital Mixers and Adders. As shown in figure 2.10, the second half of the DSSM is where the single sideband modulation actually takes place. The digital mixers multiply the I and Q signals by digitally synthesized cosine and sine waves, respectively. Both synthesized waves have the same frequency. One of the sidebands of the resulting signals is then cancelled by either adding or subtracting them. Adding them cancels the upper sideband (leaving the lower) and subtracting cancels the lower sideband (leaving the upper).

2.5 Ionizing Radiation

Both terrestrial radioactive elements and cosmic radiation are of concern to IC’s. Outside of nuclear environments, the main terrestrial elements of concern are
alpha particles emitted by traces of radioactive elements in chip packaging and cosmic radiation. Cosmic radiation can include atomic particles, subatomic particles, and electromagnetic radiation. In earth orbit, cosmic high-energy heavy ions along with electrons and protons in the Van Allen belts are the main concern [10, 17].

Within the earth’s atmosphere (i.e. at flight altitudes and on the earth) the main cosmic radiation of concern are neutrons. These neutrons are created when high-energy cosmic particles collide with the atmosphere. This causes an assortment of secondary radioactive particles and electromagnetic radiation to rain down (figure 2.14). The neutrons are the highest energy particles/radiation in this shower [8,18–20]. This secondary shower of cosmic radiation can cause upsets in modern VLSI circuits even at sea level [7]. At flight altitudes, the cosmic radiation effects are as much as one hundred times worse, and the neutron flux is three hundred times higher than on earth [8, 21].

Since heavy ions and other highly energetic atomic particles are largely shielded by the earth’s atmosphere, these are mainly of concern in space. (At sea level, typically few of the original particles that started a cosmic shower are left (< 1%) [8].) In space, the radiation environment is the most hazardous where heavy ions carrying high energies cause not only bit upsets but can destroy the circuit [22].
Electronic warfare (EW) circuits, like radar jammers function at flight altitudes. A few bit changes may or may not be of critical concern in the memory or logic of a DRFM. However, a DRFM and DSSM serve as a good representative of any circuit that might require radiation hardening.

2.6 Radiation Effects

Ionizing radiation generally has negative effects in IC’s. The radiation effects in CMOS IC’s can be divided into two types: total dose and single event effects (SEE’s). Total dose effects and destructive SEE’s are of concern in earth orbit and outer space where there are heavy ions. Earth orbit also has trapped electrons and protons in the Van Allen belts [10]. On earth and in aircraft, the main concern is soft errors caused by SEE’s [19]. A nuclear explosion also causes what is called dose rate effects, i.e. photocurrents throughout the circuit over a short time [23]. Hardening IC’s against nuclear explosions is not covered in this thesis.

2.6.1 Total Dose Effects. Total ionizing dose (TID) effects are caused by positive charge that becomes trapped in the silicon dioxide and accumulates over time from multiple ion hits. Thus, TID is a long-term, and not immediate, effect. Electron-hole pairs are created in the oxide after an ion hit. Positive charge gets trapped because the electrons are more mobile. The electrons are swept out by the gate’s electric field, and holes get left behind where they are converted into interface states at the oxide/silicon interface [24, 25].

There are three locations where this trapped positive charge causes problems. In the gate oxide, the trapped positive charge causes threshold shifts in the transistors. This is especially of concern for the gates of NMOS transistors where the shift is in the negative direction and the transistor could be stuck in the “on” state. The second area of concern is the field oxide along NMOS transistor edges. The oxide boundary acts as an NMOS gate which leaks current across the edge when the transistor is
off. Finally, positive charge trapped in the field/isolation oxide also causes leakage between neighboring N-type diffusions [10].

Formerly, with older technologies, a gate only could be total-dose hardened with a rad hard process [10, 25]. However, with deep submicron technologies, the gate oxide has become naturally hardened. In fact, for advanced technologies where the gate oxide thickness is on the order of the characteristic tunneling length (e.g. \( \sim 5 \) nm for SiO\(_2\)), total dose effects in the gate are not an issue at all. In these technologies, the gate oxide is thin enough that the positive charge can tunnel out before getting trapped [10]. This fact has opened the door to the possibility of using only radiation hardening by design (RHBD) methods for hardening IC’s for space. Section 2.8.1 looks briefly at how to use RHBD methods to mitigate the two other total dose effects outside the gate.

![SET Cross Section](image)

**Figure 2.15: SET Cross Section**

2.6.2 Single Event Effects (SEE’s). A “single event effect” (SEE) is an immediate circuit upset caused by an ion strike at a sensitive node. A sensitive node is usually the drain of an off transistor (see figure 2.15) [18]. If the deposited charge is large enough, it can alter the logic value. If the strike is in a latch, this wrong value can be captured immediately. If the strike is in the combinational logic, the pulse could still propagate through the circuit to a latch and be captured as a wrong value (see figure 2.16). However, this is less likely. Under harsh conditions of a space
environment, these strikes can also cause latchup (single event latchup) and, in high voltage devices, burnout (single event burnout) [10, 19, 21].

Smaller CMOS technologies have led to lower voltages, smaller transistors, higher operating speeds, tighter transistor spacing, smaller wells, and lower nodal capacitances. Unfortunately, when the size of an IC technology shrinks, the amount of charge deposited by an ion does not shrink with it (unless the effective collection depth changes [26]). And, less drive strength in these smaller technologies means less power to overcome a deposited charge. Also, smaller channel lengths mean greater bipolar amplification. Smaller well size means less space to dissipate the charge. Lower nodal capacitance means less capacitance to absorb the charge. Tighter spacing and smaller wells mean charge sharing is more likely between nodes and angular strikes can reach more nodes. And, faster clock speeds mean a latch has a greater chance to catch a single event pulse. Therefore, in smaller technologies, circuits have a greater potential for soft errors [26].

2.6.2.1 Parasitic Bipolar Amplification. Higher energy ion hits (e.g. in space) to a sensitive node can upset voltages and cause latchup. This happens, because parasitic BJT’s that are normally off are turned on. As shown in figure 2.15, ion strikes also turn on a lateral parasitic BJT between the source, drain, and channel. When this BJT is turned on, it injects more charge into the node [10, 27]. The gain of this BJT is dependent on the length of the transistor channel. As transistor technologies scale down in size, the channel length decreases and the bipolar gain increases [10, 26].

2.6.2.2 Charge Sharing. The charge from BJT amplification described in section 2.6.2.1 is only collected at the node. It is not shared between nodes [26]. However, ion-deposited charge does get shared between nearby nodes. This sharing is especially pronounced in a shared well, because the charge is constrained [27]. The voltage disruption from the ion’s charge can turn on a neighbor’s parasitic bipolar transistor, injecting even more charge into its node, especially in a shared well [26].
This potentially causes another upset. Neighboring transistors without wells also experience charge sharing. However, it does not create as much bipolar amplification because the charge is not constrained [27].

2.6.2.3 Single Event Upsets (SEU’s). An SEU occurs when an ion directly hits the sensitive node of a latch and a flipped bit is immediately captured and stored. This is especially of concern in SRAM cells (versus a latch in another part of the circuit), because a RAM latch is relatively small and does not have much drive to overcome the disruption [19]. An ion hit can also cause multiple latch bits to flip. This is called a multiple bit upset (MBU). An MBU occurs because of charge sharing between cells and/or an angular strike. MBU’s are more likely in smaller technologies, because SRAM cells are in closer proximity to each other.

Figure 2.16: SET Propagation to a Latch

2.6.2.4 Single Event Transients (SET’s). A SET occurs when an ion hits a sensitive node in a combinational logic cell. This causes a temporary disruption in the logic state of the cell. As shown in figure 2.16, if the pulse is large and close to a latch, it has a chance of propagating and being captured by the latch [28].

Because SEU’s are much more likely than SET’s, they have received the most attention in the past. However, with smaller technologies, SET’s will become more of an issue. With hardened memories, SET’s become the dominant source of soft errors for technologies below 0.25 \( \mu \text{m} \) [10]. In 2005, SET’s were predicted to become a significant issue at 65-nm technologies and smaller [19].
2.7 Radiation Hardening CMOS Integrated Circuits

Digital signal processing is central to today’s circuits, and these circuits are largely made with CMOS technology [10]. In this chapter, we focus on radiation hardening methods for commercial CMOS processes. We especially look at radiation hardening CMOS circuits using guard rings. Only radiation hardening by design (RHBD) methods for a typical CMOS silicon process are covered in this thesis and not, for example, the use of silicon-on-insulator (SOI) or BJT processes.

Radiation hardening reduces the negative effects of radiation in an IC either through a manufacturing process, design methods, or both. Radiation-hardened (rad-hard) foundries generally use both [10]. During the cold war, rad-hard foundries were more numerous. However, with a changing political situation and less demand, rad-hard foundries have diminished greatly [10,23]. Partly as a result of the relatively low demand, rad-hard foundries can be up to three CMOS-technology scaling generations behind their commercial counterparts [10]. Therefore, it becomes advantageous if another method (e.g. just using radiation hardening by design) can be found with less performance penalties.

2.8 Radiation Hardened by Design (RHBD) Methods

As mentioned in section 2.6.1, the MOS gate oxide in newer technologies has become thin enough to make the gates radiation hardened to total dose effects. (This does not mean it is rad hard in other areas however.) With the recent natural radiation hardness gained in the commercial gate oxide, even more attention has shifted to using RHBD methods in commercial processes. This makes sense, because commercial processes are readily available. Also, because of the lag in technology scaling at the rad-hard foundries, a RHBD commercial process can yield an area and power advantage even with the area penalties caused by RHBD methods [29].
RHBD focuses on all levels of a circuit. It can be on the level of a transistor, a sub-circuit, or a whole system [10]. A guard ring, for example, focuses on a transistor, whereas circuit redundancy focuses on a sub-circuit.

There are different radiation hardening by design (RHBD) methods to choose from. Different methods target different effects. A RHBD circuit usually uses multiple methods at once.

RHBD works by doing one of two things. Either it removes the radiation-induced charge or removes its effects [30]. Guard rings, for example, remove the charge. Whereas, circuit redundancy removes its effects. Usually, both are needed. For example, a guard ring reduces the charge so that temporal redundancy only needs to filter out shorter pulse widths.

2.8.1 Total Dose RHBD Methods. Two total dose hardening methods are annular gates and guard rings. The annular, or edgeless, gate eliminates current leakage at the edges of transistors (described in section 2.6.1) by totally removing all edges. This, however, comes at a cost in area and capacitance [10].

Another total-dose method is guard rings. Guard rings are added as channel stops between transistors of the same type to reduce leakage currents between similar diffusions [10]. As also described in section 2.9, in an n-well process, guard rings are only needed for the NMOS transistors. Guard rings come at a cost to area.

There are other total-dose methods such as choosing NAND gates over NOR gates and reverse-body biasing [23,31]. However, these won’t be described in detail here.

2.8.2 Single Event RHBD Methods. Most SEE RHBD techniques can be used for mitigating both SEU’s and SET’s. One SEE method mitigates the charge itself by increasing the size of the gate. This gives it greater drive strength to overcome deposited charge [32]. [33] mentions that this can also be done by only increasing the
size of the NMOS transistor. Another SEE method mitigates the charge itself through guard rings. Guard rings are discussed more in section 2.9.

An SEE method that corrects the effects of the deposited charge is circuit/signal redundancy (a form of error correction). Redundancy is effective, because it is less likely that more than one copy of a subcircuit/signal will be upset at the same time. A voter circuit selects the “correct” value of the output by selecting the output of the majority of the circuit/signal copies (see figures 2.17 and 2.18).

![Figure 2.17: Spatial Redundancy][10]

Redundancy can be either spatial or temporal. Spatial redundancy creates three or more copies of the same circuit (see figure 2.17). Temporal redundancy, on the other hand, creates copies of the signal, each with a varying delay. One example of temporal redundancy, a temporal latch, is shown in figure 2.18. Another example would be a guard gate. Spatial redundancy, of course, comes with a significant cost to area, and temporal redundancy can reduce maximum circuit speed.
As mentioned in section 2.6.2.3, multiple node upsets (MBU’s) are much more likely in the smaller technologies. Note that, especially in SRAM’s where transistors are tightly packed, MBU’s can reduce the effectiveness of redundancy techniques if the redundant latches are not separated [26].

All of the SEE hardening techniques come at a cost of speed or area. These costs can be minimized by only hardening critical nodes in the circuit, or at least by hardening the others less. An interesting application of this selective hardening technique is found in [32] where the author uses the gate sizing method. Here, he suggests only increasing the size of gates at sensitive nodes.

2.9 Guard Rings

Guard rings are traditionally used for preventing latchup and noise isolation. Latchup is caused by voltage disturbances in a well or substrate. These disturbances turn on parasitic bipolar junction transistors (BJT’s) in the material. Latchup can be prevented with a well tap, bar, or ring. A well/substrate tap is a spot of diffusion that connects the well/substrate to a power source. This helps to stabilize the potential of the well/substrate. It acts as a sink to bleed off unwanted voltage and charge. This in turn prevents latchup and reduces noise at the sensitive node in the well/substrate. Since these taps, bars, and rings protect a node from outside noise sources, they are also called guard taps, bars, and rings.

Guard taps with greater area and closer distance to the transistor provide greater protection [27]. Therefore, guard taps are often extended into a strip, called bars. For the best protection, they are completely extended into rings (or bands) around a transistor (see figure 2.19). A guard ring maximizes the protection of a transistor by increasing the total area of the guard tap while maintaining the minimum distance to the transistor. Increasing the width of the ring also increases protection, because this
Figure 2.19: Guard Rings Around Inverter PMOS & NMOS Transistors [34]

increases the resistance between the noise source and the transistor [35]. However, the benefit of greater width does diminish after a point [36].

For low radiation environments, guard taps (along with standard processing techniques) are usually all that is needed to prevent latchup in the standard cells [37]. For increased latchup protection, guard bars are placed between the NMOS and PMOS transitors. I/O pads, on the other hand, are completely surrounded with guard rings, because they are vulnerable to latchup caused by voltage swings from outside sources [37]. Guard rings are also useful in mixed signal circuits where analog parts need to be shielded from substrate noise caused by digital circuits [35, 36].

\footnote{Most substrate noise flows near the surface of the substrate where the guard ring can mitigate it, though some can flow under the guard ring [35, 36].}
2.9.1 Guard Rings in RHBD. In RHBD, guard rings are mainly used to reduce bipolar amplification, which is only of concern for devices in a well. In an n-well process, this would be with PMOS devices. A guard ring mitigates bipolar amplification at the hit node. A guard ring around a neighboring node also prevents bipolar amplification caused by charge sharing.

A guard ring mitigates bipolar amplification in a well by lowering resistance from the well to the source. The desire is to reduce both vertical and horizontal resistance (see figure 2.20). Vertical resistance to the guard ring can be minimized by increasing its area. However, after a certain point the horizontal resistance begins to dominate [27]. Horizontal resistance can also be minimized by moving the guard ring as close as possible to the transistor.

Unfortunately, with smaller technologies, guard rings become less effective at reducing bipolar amplification. This is because of decreasing channel width (which

---

4Bipolar amplification is only a significant problem in a well, because the well constrains the deposited charge [27]. For transistors without a well, the main charge collection mechanisms at a hit node are drift and diffusion, not bipolar amplification. Also, for charge sharing, the main charge collection is diffusion [33]. [26] found that in a 130-nm process, the guard ring around an NMOS device (with no well) gave only some mitigation on the charge sharing. And, [38] found that in a 90-nm process, the guard ring around an NMOS device gave essentially no benefit for mitigating charge sharing.

5This at least is true in a shallow trench isolation (STI) process. If there is a desire to test guard ring widths, [36] demonstrates a way that a guard ring’s effective width can be switched during operation.

6Note that it is also suggested that the guard rings use high-density well contacts [34].

![Figure 2.20: SET Cross Section](image)
increases bipolar gain) and the smaller-sized guard ring [27]. The smaller-sized guard ring could be counteracted by re-sizing it. However, of course, this would lead to even larger spacing between the transistors. Also, the increased bipolar amplification would still be a problem. Even though minimum-sized guard rings were probably used, [26] demonstrates that in a 130-nm process n-well, the guard ring is very effective in mitigating bipolar amplification caused by charge sharing between two PMOS devices. However, [38] demonstrates that in a 90-nm process n-well, the guard ring is somewhat effective, but does not remove all of the collected charge.

Guard rings need to be used in conjunction with other RHBD methods. Guard rings reduce, but do not totally remove, the amount of collected charge from bipolar amplification. By extension then, guard rings also reduce the SET pulse width. Other techniques are still needed to overcome the charge or remove the pulse. Increasing the size of the NMOS transistor could be used to overcome the collected charge in the PMOS device [33]. Spatial or temporal redundancy techniques could be used to remove the pulse. For temporal redundancy, a guard ring would reduce the pulse width going into the temporal redundancy circuit, which would allow it to run faster [33]. Additionally, charge sharing could be reduced by nodal separation and interdigitation [26,38,39]. Charge sharing could also be mitigated by putting PMOS transistors into separate wells (at a cost in area) [40].

In summary, guard rings are placed around PMOS devices in an n-well process to mitigate SEE’s. In total dose hardening, however, they are placed around the NMOS transistors (see section 2.8.1).\(^7\) In total-dose hardening, a complete guard ring with no breaks is required to keep potential leakage from getting in through an opening. This means polysilicon cannot be allowed to cross a guard ring as this would cause a break (see figure 2.19) [10].\(^8\)

---

\(^7\)This is because NMOS transistors are susceptible to the threshold shifts caused by total dose. In this case, the guard ring stops the creation of a parasitic NMOS transistor channel between similar diffusions.

\(^8\)Note that for total dose hardening, sometimes more than one transistor can be enclosed by a guard ring [10,41].
2.9.2 Guard Ring Costs. A guard ring around a transistor means there can be no shared diffusion. This costs area and at least a slight addition to diffusion capacitance [37]. The width of the guard bars/rings above and below NMOS and PMOS transistors can also add to the cell height. Thus, the main cost of guard rings is added area.

2.10 Measuring Ion Energy

Ion energy is measured in terms of millions of electron volts (MeV). Ion hits in IC’s, on the other hand, generally are measured in terms of linear energy transfer (LET). LET is a measure of an ion’s energy deposited per travelled distance in a given material. Its units are MeV cm$^2$/mg. The LET of space ions range from a few hundredths to almost 100 MeV cm$^2$/mg. The higher LET particles are less common [24,34,42]. Lower LET energies (e.g. 9.74 to 58.6 MeV cm$^2$/mg) can also be used to represent typical neutron-generated particles [20,38].

2.11 Testing Circuits for SET’s

Circuits destined for the space environment are typically tested with heavy ions generated by a particle accelerator [42]. This is a form of accelerated testing. Laser testing can also be used to test a circuit. It focuses on the sensitivity of a particular node, and is not random like heavy ion testing. It is also much cheaper [43]. Total-dose testing, on the other hand, is often done with a Cobalt-60 source [44].

In heavy-ion testing, the ions strike the circuit in a random pattern, similar to a shotgun pattern. To characterize a cell for SET pulse widths, a test circuit should have large numbers of the same cell arranged in chains. This increases the likelihood of a cell being hit [45]. The SET pulse is then propagated through the chain.

A few different methods can be used to measure the width of the pulse. One has latches distributed along the target chain [46]. Another has a pulse capture circuit
with latches at the output of the target circuit [45]. Test results in [34] and [47] could be used as test benchmarks if one were doing guard ring SET testing.
III. Implementation

Several versions of the DSSM were synthesized up to the place-and-route (PAR) level. First, eight versions were synthesized with a typical characterization corner. Next, one of the 16-input-bit designs was synthesized for two more characterization corners. (One of these was compared to the radiation-hardened version.) A piped version of the 24-bit design was also synthesized to test for speed gain. Power, timing, and area were estimated, where possible. Key aspects of the implementation were that the design was parameterizable and the synthesis, simulation, and analysis were automated. In this chapter, we look at the implementation details and steps.

3.1 DSSM Design Modules

![Diagram of DSSM design modules]

Figure 3.1: Top Level Module - DSSM

The DSSM was composed of a digital Hilbert filter and “mixer” module (which is actually an SSM). The Hilbert filter created the in-phase (I) and quadrature (Q) signals. The mixer module used these signals to create a digital sideband. The entire
circuit was preceded by a multiplexer to allow switching between an ADC input or the output of the Hilbert register.

### 3.1.1 Input Multiplexer
The input multiplexer was designed for testing. One port was for direct input from an ADC. The other could be connected to an external DRFM memory. However, in this thesis the second input was connected to the output of the Hilbert register as in [1]. This was because the chip in [1] did not have space for more I/O pads.

![Figure 3.2: A 101-Tap FIR Hilbert Filter](image)

### 3.1.2 Hilbert Filter
The HDL for the Hilbert filter was structured as a direct-form FIR filter. The number of taps for the test cases were odd. The number of multipliers were not reduced in the HDL code as mentioned in 2.4.1. As in [1], the Hilbert coefficients were only windowed with a rectangular window function. Therefore, we should expect a ripple in the output. Optimization of the multipliers and adders were left up to the design synthesizer.

The number of taps were varied in some of the designs. The coefficient bit width was kept the same as the input bit width for all of the designs except one (DRFM_16_14_101_256).

[48] developed and tested Hilbert and DSSM HDL code to compensate for amplitude and phase error between the I and Q signals. That work was not completed or available at the time of the work of this thesis. Therefore, some of those errors may still stand uncorrected here. [1] roughly matched the magnitudes of I and Q by artificially amplifying the Q signal with a bit shift. That “trick” was maintained for these designs since the same code was used.
3.1.3 Mixer. The “mixer” module is shown in figure 3.3. A ROM wrapper was used as a subcomponent for controlling the ROM’s. The address for the ROM’s were clocked on the falling edge of the clock. The address setup circuitry was the only part that functioned on the falling edge of the clock.

A high speed ROM for each wave, sine and cosine, was created using the vendor’s software ROM generator (see figure 3.4). The vendor ROM’s had a 512 minimum word size. Therefore, part of the vendor ROM’s went unused for ROM sizes that
were smaller than 512. Each ROM stored a quarter of the wave. Using two extra address bits from the counter, a control circuit in the ROM wrapper constructed the full waves.

\[
\text{DDS \,} \text{Freq.} = \frac{(F_{c_{in}})(F_s)}{(4)(\text{ROM\_size})}
\]  

(3.1)

Equation 3.1 can be used to calculate the frequency from the DDS, where \(F_s\) stands for the clock frequency and \(\text{ROM\_size}\) stands for the number of words in the ROM. The input factor, \(F_{c_{in}}\), determined the increment between the samples in the ROM (between 1 and \(\text{ROM\_size}/2 - 1\)).

In the rad-hard design in [1], there was some concern whether the ROM’s in the structured ASIC would work. Therefore, [1] created an extra selectable ROM for each wave as a backup. These backup ROM’s were not included in this thesis.

The multipliers in the mixer used Wallace tree multipliers (WTM’s) designed by Thomas Pemberton [1]. These were created using Pemberton’s generator. The adder and subtractor were coded separately and left to the compiler to implement. The output multiplexer was used to select between the addition and subtraction for sideband selection.

### 3.2 DSSM (DRFM) Inputs and Outputs

The following are the inputs and outputs of the DSSM and the nomenclature used. In the HDL code, the I/O names were preceded by a “P” if it was a module port. Since this design was meant for a DRFM, the DSSM is called a DRFM in the design files throughout.

**Inputs:**

ADC\_IN: Test input

---

\(^1\)The range of \(F_{c_{in}}\) should have been from 1 to \(\text{ROM\_size}\). It was limited to just under half that because of an error in the mixer VHDL code. The code used a signed library, and the \(F_{c_{in}}\) term added to the address should have had two zeros concatenated to the front of it, which it did not. This has been corrected in the mixer code in appendix A.
IE\_IN: Input enable for the I/O pads
CLK\_IN: Clock input
MUX\_S: Select for ADC\_IN or Hilbert register output
RST\_IN: Reset for Fc counter
Fc\_IN: Determines center/shift frequency
ADD\_SUB\_SEL: Sideband select

Outputs:
MUX\_OUT: Output of first MUX (test port)
HIL\_Q\_OUT: Hilbert quadrature output (test port)
ROM\_COS\_OUT: Cosine output (test port)
DRFM\_OUT: Final output

3.2.1 Input and Output (I/O) Pads. The designs had I/O pads for every bit of the inputs and outputs. In the HDL code, pads were named starting with the letter “G”. The rad-hard design in [1] had an extra input pad for a ROM select. One pad space was kept open (blank) in this thesis to keep the designs comparable for number of pads.

There were also I/O pads for pad power (GDVDD & GDVSS) and core circuit power (GVDD & GVSS). For the 16-bit designs, the ratio of estimated power to total number of core power pads was kept approximately the same as in the RHBD design for its estimated power (∼19mW/pair). This led to three pairs of core power pads for the 16-bit designs. The RHBD design had an equal number of pad power pads and core power pads. This was not clear at the time of choosing, so two pairs of pad power pads were chosen. (The 512 ROM design did have a third pair of power pads since there was extra space created by the need for another address pin). These power pad numbers were approximately halved for the 8-bit design (two pair of core
and one pair of power pads) and doubled for the 24-bit design (seven pair of core and four pair of power pads).

### 3.3 Design and File Naming Convention

<table>
<thead>
<tr>
<th>DRFM</th>
<th>16</th>
<th>14</th>
<th>101</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Input</td>
<td>Coeff.</td>
<td>No.</td>
<td>ROM</td>
</tr>
<tr>
<td>Bits</td>
<td>Bits</td>
<td>Taps</td>
<td>Words</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.5: Design and File Naming Convention

As shown in figure 3.5, the designs and associated files and directories were named based on the parameters used. This made it easy to organize the designs and files and to reference them in the scripts.

### 3.4 Hardware Description Language (HDL)

VHDL is the language of choice for the Automation Design and Test Laboratory at Wright State University. It was also the language of existing code that we had from others’ previous work with the Hilbert filter and test bench and the language used by Pemberton in [1]. Therefore, it was also the HDL of choice for the modules in this thesis.

### 3.5 Pre-Testing

The DSSM design was originally tested on an FPGA development board (Xilinx® Virtex II Pro) in the lab by Pemberton [1]. The author of this paper gained familiarity with the FPGA implementation and its results by working alongside Pemberton for some of the process. The VHDL code used for the FPGA was modified by each author for implementation of the respective ASIC designs.

### 3.6 Software Tools

Simulations were done with Cadence NCSIM 6.11 and its GUI, SimVision 6.11. The HDL code was synthesized with Cadence Encounter® RTL Compiler 6.2. The
RTL Compiler generated netlist was laid out and routed using the Cadence Encounter family of tools (First Encounter® 6.2, Power Planner/ViaGen 6.2, NanoRoute™ 6.2, and the GUI, SoC Encounter™ RTL to GDS Implementation Solution 6.2). Finally, the frequency analysis and DSSM models were done with MATLAB® 7.8 R2009a.

3.7 Design Flow

![Diagram of Design Flow]

**Figure 3.6: Design Flow**

As shown in figure 3.6, a generally standard design flow was followed. The order of some steps could be varied. The design steps are described next.

3.7.1 Design File Generation. Some of the VHDL files were automatically generated with software created by Thomas Pemberton [1]. The lib, vclef, and behavioral Verilog files for the ROM’s were generated with the vendor’s software. Remaining VHDL files were created from existing design files. A template and script was created for generating the various top-level VHDL files.
Figure 3.7: Cadence SimVision® 6.11 Output

3.7.2 RTL-Code Simulation. Simulations were automated with a script described in section 3.13.3 and a test bench. NCSIM® usually was run without the SimVision® GUI. However, the GUI was used during early simulations when the VHDL code was being debugged and for probing particular signal values. The test bench created sampled sine waves for the input and saved the input and output data in files for use later in MATLAB®.

3.7.3 RTL Synthesis. The RTL code was synthesized using a script that is described in section 3.13.1. The optimization effort was set to “high” and the carry-save adder optimization effort was set to “high.” The script did not need the Encounter® graphical user interface (GUI) to run, and the GUI generally was not used. (However, it could be used to view the schematic.) The timing target given was 100 MHz. Reports for estimated power and timing from the synthesis were saved in a directory for later use.

3.7.4 Gate-Level Netlist Simulation. The gate-level netlist produced by RTL Compiler was simulated. This simulation used the same script as the RTL-code simulation. Clock speed limits of each design were found by decreasing the period in 5
ns increments from some upper value until timing violations occurred (see test bench in appendix B). The netlist was then retested for a full cycle at the last “clean” speed. If it also produced violations, it was run again at the next higher 5-ns increment.

As mentioned later in section 4.6, the 24-bit case was the slowest design case with a maximum operating frequency of at least 16 MHz. Therefore, to keep all of the designs on the same scale, the simulation clock had to be around 16 MHz for all designs. An \( F_c \) factor of 50 was chosen for the 256-word ROM designs. Therefore, the clock frequency was chosen to be 15.36 MHz to give a convenient shift frequency of 0.75 MHz. This means the usable frequency band was half that at 7.68 MHz.

Next, the netlist was simulated with a sweep of input frequencies from 0.75 to 6.75 MHz at 0.15 MHz intervals for a total of 41 frequencies. Input frequencies below 0.75 and above 6.75 MHz were left out to give a better average SFDR. All designs were simulated in the upper sideband (USB) mode for the frequency sweeps. The pre-synthesized RTL code was also simulated for all the same input frequencies for comparison.

First Encounter\textsuperscript{®} also produced a netlist after the place and route (PAR) step. For one or more designs, the post-PAR netlist single-frequency simulation gave a similar timing limitation and the same SFDR as the post-RTL-Compiler netlist single-frequency simulation. Therefore, to save time, simulations were only done using the netlist produced by RTL Compiler.

3.7.5 Layout and Routing. The layout and routing step used a script that is described in section 3.13.2. Again, the SoC Encounter GUI was not needed to run the script but was useful for viewing the results. See figures 3.8 through 3.11 for example results of the layout steps. The script accomplished: a floor plan, placement, power routing, clock tree insertion, network routing, cell fill, save design, and design checks and verification, in that order. The save step produced a DEF file which could be used at the final physical layout step in a program like Cadence Virtuoso\textsuperscript{®}. This final physical layout step was not taken, and GDSII files were not generated.
Figure 3.8: SoC Encounter® After Floor Plan

Figure 3.9: SoC Encounter® After Placement
Figure 3.10: SoC Encounter® After Power Routing

Figure 3.11: SoC Encounter® After Clock and Signal Routing
Notice in figure 3.9 that the total circuit cell areas for the designs were much smaller than the minimum chip and core areas allowed by I/O pad constraints. Therefore, the design chip areas were solely determined by the number of I/O pads. The result is that the chip and core areas were set and did not need to be changed after cell placement and routing.

First Encounter® reported chip, core, standard-cell, and ROM-macro areas in it’s summary report. Of course, the chip and core areas were pre-determined by the I/O and command files. Total cell area was calculated by subtracting the filler cell area from the total standard cell area and adding the ROM macro areas. This area did not include the ROM power rings and halos. Percent of core area utilized was calculated by dividing this total cell area by the core area. Estimated timing data for worst slack was obtained from the post-PAR timing analysis report.

3.7.6 Frequency Analysis. MATLAB® m-files were used to analyze output data generated by both the simulation and a mathematical model of the circuit. The designs were then compared for output and SFDR for various parameters. A comparison was also made between the HDL-code simulation, gate-level netlist simulation, and the MATLAB® model for each design. (See appendix D for the MATLAB® scripts.)

3.8 Technology Files

The standard cells and ROM macros were taken from a non-hardened commercial CMOS 90-nm process with eight metal layers. The vendor ROM’s would ultimately be implemented with programmable vias. We used the via technology implementation as it was the only choice available. Though via programmable ROM’s are not the most compact choice, they have the benefit of being able to make last minute changes to ROM data without changing the design layout.

The standard cell vendor provided behavioral Verilog (.v) files for simulation as well as timing (.lib) and layout (.lef) files for synthesis and layout. The vendor
ROM software generator also created behavioral Verilog (.v), timing (.lib), and layout (.vclef) files. The lib, lef, and vclef files were referenced by both the RTL Compiler and First Encounter® software.

3.8.1 Characterization Corners. Characterization corners were chosen from the technology lib files for the standard cells, I/O cells, and ROM macros. For the eight-design comparisons listed in section 3.10, typical characterization corners were used, i.e. typical NMOS, typical PMOS (tt), with 1.20V voltage source and 25°C chip temperature. One design, DRFM_16_14_101_256, was also created with two more characterization corners: fast PMOS fast NMOS (ff) at 1.32V and 125°C, and slow PMOS slow NMOS (ss) at 1.08V and 125°C. The three design corners were then compared with each other. The worst case of these three (ss corner) was also compared with the rad-hard version of the circuit described in [1] (see section 3.14).

3.8.2 Transistor Threshold and RC Values. All designs used a regular threshold voltage (rvt) in their library files. The I/O lef file, however, was labeled as being for high threshold (hvt).

The RC lef file noted that it used worst-case RC interconnect models from the foundry. Therefore, as also noted in the lef file, the post-PAR timing results should be assumed to be very conservative.

3.9 File Directory Structure

![File Directory Structure](image)

Figure 3.12: File Directory Structure
A directory structure, shown in figure 3.12, was devised to compartmentalize the design steps. This top-level directory had separate directories for source files, generated files using Pemberton’s generators, vendor-generated ROM files, technology (.lib, .lef, and Verilog) files, RTL compiler files, place and route files, and simulation files (src, gen, rom, [technology files], rc, enc, and sim, respectively). The source directory (./src) had some of the design modules, namely: the top-level DRFM files, the mixer file, and ROM wrapper files. The generator directory (./gen) had a components file, the VHDL generators provided by Thomas Pemberton, and the resulting generated files.

The vendor ROM directory (./rom) had separate directories created for each bit and ROM size with the associated generated files placed in each. The technology directory (./[techn. name]) had the layout and timing files for the standard cells and I/O pads. The RTL Compiler directory (./rc) had the script file and directories created for each design's reports. The First Encounter® directory (./enc) had the command script file and directories for the resulting layout and report files. The simulation directory (./sim) had the simulation script file and separate directories for the results from each design.

### 3.10 Test Cases and Metrics

There were eight different parameter variations implemented. Table 3.1 lists each combination of input bit widths, filter coefficient bit widths, number of filter taps, and number of ROM words.

All of the cases, except one, had Hilbert coefficient bit widths the same as the input. All of the ROM bit widths (not shown) were the same as the input bit widths.

As mentioned in section 3.8.1, the DRFM\(_{16\_14\_101\_256}\) was also implemented with two extra characterization corners. To test speed improvement via piping, a second 24-bit case was also synthesized with the Hilbert filter multipliers piped.
<table>
<thead>
<tr>
<th>I/O Bit Width</th>
<th>Coeff Bit Width</th>
<th>No. of Taps</th>
<th>ROM Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>101</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>101</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>101</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>33</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>153</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>101</td>
<td>128</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>101</td>
<td>512</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>101</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 3.1: Test Cases

The timing target was 100 MHz. After implementation of the designs, we summarized power, delay, critical path, and area using the generated reports. As described in section 3.13.4, the simulated outputs were also measured for spur-free dynamic range (SFDR) for a sweep of 41 input frequencies. The results are compared in chapter 4.

### 3.11 HDL Files

Each of the major modules in the hierarchy were in separate HDL files (see appendix A). The DSSM (called DRFM in the files) was the top-level file. Then there were: the mixer, the ROM wrapper, the ROM’s, and the Wallace tree multiplier files.

The DRFM top-level code, mixer code, and ROM-wrapper code were developed from the code used for FPGA testing in [1] and from discussions with Thomas Pemberton. Code for the Hilbert filter, Wallace tree multiplier, and ROM bit file were obtained from generators developed by Pemberton. The code for the Hilbert filter generator and the test bench, in turn, were derived from existing code used in other research in the laboratory.

#### 3.11.1 Parameterization of the HDL Files

The DSSM (called DRFM in the files) was fully parameterizable in terms of: input bit width, Hilbert filter coefficient bit width, number of filter taps, and number of ROM words. The first step in a parameter change(s) was to create new source files.
Files for some to all modules needed to be recreated for each new parameter combination. First, a new file was always made for the top-level DRFM. This was generated with a template file and UNIX-shell script (with nested Perl commands). The Hilbert filter, ROM files, and WTM files were also made with file generators. New files for the ROM wrapper were created from copies of old files in a text editor. The mixer file never needed to be changed for new parameters. Next, what needed to be changed in the files for each parameter change is described.

For changing the number of input bits, the generic values were automatically changed in the new DRFM module using the DRFM generator. The Hilbert files received no changes. However, all new ROM files and Wallace tree multiplier files were generated. Of course, the accompanying files for the vendor ROM’s (.lib and .vclef) were generated as well.

To change the Hilbert filter’s coefficient bit width or number of taps, a new Hilbert file was generated, and a new DRFM file was made with the DRFM generator.

To change the ROM size, new ROM files were generated. A new ROM wrapper file was also made, because there were changes in the number of address bits assigned to zero (‘0’) in the vendor ROM.\(^2\) A new DRFM file was also made with the DRFM generator.

To summarize, there were multiple versions of the module files for the test cases. There were three of the Wallace tree files (for each bit size); five of each ROM and associated files (for each bit size and ROM length combination); six of the Hilbert filter (for each number of taps and coefficient bit size combination); three of the ROM wrapper (for each ROM length); one of the mixer; and ten top-level DRFM’s (for all the designs).

\(^2\)Some address bits got assigned to zero, because usually the whole vendor ROM was not used. This was because the vendor ROM generator had a minimum number of words of 512. If the ROM length was less than 512, dummy data was appended to the data file to make it 512 words long.
Note that ROM bit widths were always chosen to be the same as the input bit widths. Likewise, the bit width of the cosine output port was the same as the ROM bit width.

3.12 VHDL Test Bench Files

The VHDL top-level test bench was adapted from an existing (Hilbert filter) test bench and made into a template (see appendix B). The template was copied by the simulation UNIX-shell script, which inserted the appropriate names and variable values (via Perl commands) for each simulation. The test bench created a single-tone sine wave, which it quantized like a comparator-based ADC where each value was “floored” to the next lowest quantization value. The test bench could save up to seven output data files for: the input data, the output of the input-MUX, the Q-signal from the Hilbert filter, the cosine signal from the ROM, the DRFM output, a general parameter listing, and a parameter file to be read into MATLAB®. T

One vendor ROM behavioral code was also tested with its own test bench to confirm operation. The Wallace tree multiplier generator also generated its own test bench for testing it.

3.13 Automation of Design Flow

The design synthesis and simulations in this thesis were automated with script files (see appendix C). Script files were used for RTL Compiler, First Encounter®, NCSIM, and the MATLAB® analysis. The first three scripts were modified from files received from Dr. J. Marty Emmert. The MATLAB® m-file for frequency analysis was modified from a file created by Thomas Pemberton.

Parameter and library variables were placed at the top of the scripts where they could easily be changed. In turn, entity names and file suffixes in the scripts were built using the variables. As mentioned in section 3.9, a directory structure to store
the various designs was devised, based on the top-level entity names. References to the files and directories were coded with the variables in the scripts.

3.13.1 RTL Compiler Script. There was a starter UNIX-shell script which launched RTL Compiler and called the command script. The command script synthesized all RTL code into a Verilog netlist (see appendix C). The netlist and configuration files were saved to the design file folder (./*files) in the First Encounter® directory (./*enc).

The reports for estimated power, delay, and area were saved to a folder with the design name in the RTL Compiler directory (./*rc). Multiple designs could be synthesized sequentially and/or simultaneously by making copies of the script, updating the variables, and running them with another shell script. The log files were moved to the report directories manually.

3.13.2 First Encounter® Script. As with RTL Compiler, there was a starter UNIX-shell script to start First Encounter® and call the command script. The command script was revised from an example script provided by Dr. J. Marty Emmert (see appendix C). The script was made up of seven sub-scripts which were retained. In the order used, the sub-scripts were for: floorplanning, placement, power place and routing, network routing, clock tree insertion, adding filler cells, and checking and verifying. (There was another sub-script for metal fill that was not used, because the designs were not being sent out for fabrication.) These scripts were then concatenated into one script, “build_all.cmd.”

For each design, the design name needed to be changed at the top of the script. One of three floor plans also had to be selected. This was because the number of I/O would vary and change the size of the design. The floorplans not used were commented out. As mentioned below in 3.13.2.1, there were also two supporting design files that were created.
The design was saved at the end of each sub-script. This allowed for the process to be interrupted, if needed, without losing the results of previous steps. It also meant, subsequent steps could be changed, if desired, without having to recreate the previous steps. To continue the process, completed steps in the script could be deleted, a restore-design command uncommented, and the script rerun. This division of scripts also allowed the order to easily be changed, where appropriate.

As with RTL Compiler, multiple designs could be synthesized sequentially and/or simultaneously. As the working directory for synthesis was the main First Encounter® directory, some files that were not directed to a design folder and the log file needed to be manually moved to the appropriate design folder after synthesis.

3.13.2.1 First Encounter® Design Files. Besides the technology files, First Encounter® had four input design files. They were all text files: the Verilog (.v) netlist from RTL Compiler, a timing file (.sdc) from RTL Compiler, an I/O file (.io) created by hand, and a configuration file (.conf). All four of these files were kept in a separate directory called ”des_files.”

The I/O file listed each I/O port in order, each pad location, and each pad name. New I/O files were created from old ones and named based on input bit width and ROM size. If the I/O changed, the contents of the I/O file were changed. Of course, the port names had to match the port names in the technology and top-level DRFM file. The number of power pads were also kept comparable to the number proportionally used in the comparison rad-hard circuit in [1] (see appendix C).

The configuration file was also a script file of sorts and was read in by one of the first commands in the First Encounter® command script. Again, new files were created from old ones and were given the same name as the design. As in the other scripts, the configuration file had parameter variables at the top of the file that could simply be changed for the new design (see appendix C).
3.13.3 *Simulation Script.* There was a UNIX-shell script for simulation (see appendix C). The simulation script copied a template test bench and inserted the appropriate names and variable values for simulation. It also created a design directory where up to seven output data files were saved (see section 3.12). The appropriate design files were then compiled and elaborated, and the design was simulated.

3.13.4 *MATLAB® Scripts.* MATLAB® was used to analyze the data output from the simulation files in the frequency domain using a fast Fourier transform (see appendix C). An m-file script was built from a core script written by Thomas Pemberton (see appendix D). The script read in the simulation parameters and output data, plotted the frequency spectrums for all the input frequencies (see figure 3.13), plotted the output versus input frequency sweep, plotted the spur-free dynamic range (SFDR) versus input frequency sweep, saved the frequency sweep data, and plotted a sample input frequency in the time domain.

Figure 3.13 shows an example frequency response plot of an 8-bit DSSM netlist for an input frequency of 6.30 MHz and shift frequency of 0.75 MHz. The red peaks in the frequency plots were the input frequency components. The blue peaks were the output frequency components. Input and output peaks were not relative to each other but were only calculated relative to themselves. Both were calculated in dBc (decibels in reference to the carrier, i.e. in reference to the maximum peak). As shown in figure 13(b), at each input frequency, SFDR was measured (in dBc) between the maximum output peak (at 0 dBc) and the highest spur. The mean average SFDR was calculated for each design model by averaging the SFDR for all 41 input frequencies.

When selected, the script also generated and plotted its own mathematical model of the design. This could be compared to the simulated data. As will be seen in sections 4.9.1 and 4.9.2, a second script plotted all three models for each design for DRFM frequency response and SFDR (again, see appendix D).

As in the other scripts, parameters for the MATLAB® analyses were placed at the top of the scripts where they could easily be changed. However, the simulation
Figure 3.13: FFT of Input and Output of DRFM_8_8_101_256 (dBc)  
Fin = 6.30 MHz; Fout = 7.05 MHz
parameters were automatically set by the input file “drfm\_param\_MATLAB.txt”. For the mathematical model, to plot signals other than DRFM\_OUT, a line could be uncommented further down in the script.

3.13.4.1 Mathematical Model of the DSSM. The HDL code for the DSSM was modeled mathematically in MATLAB\textsuperscript{®} for functional verification (see appendix D). The math model used the same equation for the input signal as the test bench. It used equation 2.7 for calculating the coefficients. It convolved the coefficients and the input to calculate Q. I and Q were calculated with more samples than needed. I and Q were then truncated in such a way that I had the appropriate shift relative to Q. The address for the ROM was generated using the Fc factor and the modulus operator. The DDS cosine and sine values were then calculated directly with the address as an argument. Mixing was done with multiplication and summing with addition. Finally, all calculated values were quantized via a quantizer object with word length equal to the number of input bits and fraction length equal to the number of input bits minus one.

3.14 Rad-Hard Circuit Comparison

The DSSM circuit was also implemented by Thomas Pemberton, in a 90-nm RHBD structured ASIC [1]. The structured ASIC used programmable via. It was radiation hardened for: total dose, SET’s, single event latchup (SEL) in memory, and perhaps other effects. One of the RHBD methods used in the technology was temporal latches [49].

One version of the DSSM in [1] was sent for manufacturing. This thesis synthesized a DSSM with the same parameters in the non-hardened technology for comparison. The synthesis and simulations for both DSSM’s (up to place and route) were then compared for power, timing, area, and chip utilization. Note, however, that the RHBD structured ASIC in [1] had not been characterized for delay or power. Therefore, the delay and power models in its library were only estimates.
The exact code in [1] was not used in this thesis. However, the HDL was made as close as possible to that used in [1] through the reuse of code used in FPGA testing and through personal conversations.

The parameters for the two comparison circuits were: 16 input bits, 14 coefficient bits, and a 256-word ROM. Pemberton’s process corner was designed at what was considered a worst case at 1.08V supply voltage and 125°C chip temperature. Since it was a worst case, this paper assumed that his process corner was: slow NMOS, slow PMOS (ss). Thus, our test case for comparison used: ss, 1.08V, and 125°C.
IV. Results and Analysis

In this chapter, results are summarized for power, timing, area, layout, SFDR, and frequency response. To show detail, the frequency response plots are zoomed in to a peak-to-peak range of about 0.7 dBc or less. This means the ripple is not as dramatic as it may appear. Since the behavioral files provided by the vendors for the standard cells and ROM’s used generic values (e.g. 1 ns) for its delays, the simulation timing results are not considered to be absolutely accurate. However, they provided a relative comparison between the designs. As mentioned in section 3.7.5, cell area was calculated as the total of the standard cell areas and ROM macro areas, not including the power rings or halos around the ROM’s. All values in the summary tables were derived from the EDA tools, simulations, and frequency analyses.

Section 4.1 first summarizes the results for the non-hardened versus radiation hardened DSSM comparison. These two designs are compared for power, timing, area, and chip utilization. Next, sections 4.2 through 4.5 summarize the results of varying four design parameters (I/O bit width, coefficient bit width, number of filter taps, and ROM size) with one section given for each parameter. These designs were all synthesized at a typical characterization corner. Each parameter’s section summarizes results for power, timing, area, and SFDR. Section 4.6 then summarizes all the parameter variations and gives the critical paths.

Section 4.8 shows the results of not piping versus piping the Hilbert filter multipliers in the DRFM.24.24.101.256. Then, section 4.7 shows the results of using three different characterization corners (slow, typical, and fast with varying operating conditions) in the DRFM.16.14.101.256. Both of these sections summarize results for power, timing, area, SFDR, and critical path.

Again, as mentioned in section 3.7.5 and as demonstrated in figure 4.1, the total circuit cell areas for the designs were much smaller than the minimum core and chip
Figure 4.1: Layout for DRFM_16_14_101_256
areas allowed by I/O pad constraints. Therefore, the design chip areas were solely determined by the number of I/O pads.

Figure 4.2: Example Freq. Plot for DRFM_8_8_101_256

Figure 4.2 shows a sample FFT plot for the 8-bit DSSM design. Ideally, the DSSM’s lower sideband would always be suppressed below the noise floor. Unfortunately, for many input frequencies, like in this figure, it was not. For the eight-bit I/O case, the lower sideband was the highest spur for about half of the input frequencies. For the 16 and 24-bit I/O designs, the lower sideband was the highest spur the majority of the time. Section 4.9 discusses this further.

4.1 Rad-Hard Comparison

The 90-nm RHBD ASIC technology libraries had not been characterized for power or delay. Therefore, the power and timing values are only estimates. Just going by these estimates, the rad-hard circuit utilized 577% as much power as the non-hardened version as seen in table 4.1. The RHBD design had its timing analyzed
by two different programs giving two results for timing. The post-PAR timing results for the RHBD design were about 30% higher than for the non-hardened design.

Chip area of the RHBD design was 123% higher than the non-hardened version. The RHBD chip size was predetermined. On the other hand, the non-hardened chip size was solely determined by the number of I/O pads. One reason the non-hardened chip was smaller was because it had less power pads due to its lower power needs. Another reason was because it did not have any rad-hard specific I/O pins, of which the RHBD had a few. If the RHBD power estimate is off significantly, the non-hardened area would change, because the number of power pads would change. However, this would only be expected to change slightly as all the other I/O pads would still remain the same.

The RHBD design’s goal was to use as much of the chip area as possible. Therefore, if DFF utilization were equivalent to area utilization, the RHBD circuit utilized 64.7% of its available area. On the other hand, the non-hardened version used only 5.4% of its available area (not counting the spacing between the cells, or ROM-macro power rings and halos). This was because the non-hardened design was I/O pad constrained, and its goal was to match the RHBD design. The two circuits were not compared for SFDR. However, since both designs used the same architecture, the SFDR is expected to be very similar.
### Table 4.2: Results for 8, 16, and 24 I/O Bit Widths

<table>
<thead>
<tr>
<th></th>
<th>8_8_101_256</th>
<th>16_16_101_256</th>
<th>24_24_101_256</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Power (mW)</td>
<td>22.0</td>
<td>59.0</td>
<td>111.8</td>
</tr>
<tr>
<td>Max Sim (MHz)</td>
<td>33</td>
<td>25</td>
<td>16</td>
</tr>
<tr>
<td>RTL Cmplr (MHz)</td>
<td>110</td>
<td>116</td>
<td>105</td>
</tr>
<tr>
<td>Post-PAR (MHz)</td>
<td>117</td>
<td>110</td>
<td>68</td>
</tr>
<tr>
<td>Cell Area ($\mu m^2$)</td>
<td>68,156</td>
<td>161,155</td>
<td>310,421</td>
</tr>
<tr>
<td>Core Area ($\mu m^2$)</td>
<td>818,880</td>
<td>2,916,991</td>
<td>7,059,224</td>
</tr>
<tr>
<td>Core Utilization</td>
<td>8.3%</td>
<td>5.5%</td>
<td>4.4%</td>
</tr>
<tr>
<td>Chip Area ($\mu m^2$)</td>
<td>2,082,249</td>
<td>5,044,516</td>
<td>10,208,025</td>
</tr>
<tr>
<td>Mean SFDR (dBc)</td>
<td>37.7</td>
<td>45.5</td>
<td>45.5</td>
</tr>
</tbody>
</table>

#### 4.2 I/O Bits Comparison (8, 16, and 24)

As seen in table 4.2, increasing I/O bits led to increasing area and power. Increasing the I/O bits by 100% from eight to sixteen bits increased the switching power by 168%. Increasing the bits again by 50%, increased the power by 89%.

From eight to sixteen bits, PAR timing decreased by 6%. From sixteen to twenty-four bits, PAR timing decreased more dramatically by 38%.

From eight to sixteen bits, cell area increased by 136%. (Chip area increased by 142%.) From sixteen to twenty-four bits, cell area increased by 93%. (Chip area increased by 102%.)

From eight to sixteen bits, average SFDR increased by about 8 dBc. However, SFDR stayed the same for the sixteen and twenty-four bit designs (see also figure 4.3).

As seen in figure 4.4, the ripple was the greatest in the frequency response for the eight bit design. Whereas, for the sixteen and twenty-four bit designs, the ripple stayed the same. Though not to scale, the layouts can be seen in figure 4.5.
Figure 4.3: SFDR for 8, 16, and 24 I/O Bits (dBc)

Figure 4.4: Freq. Response for 8, 16, and 24 I/O Bits (dBc)
Figure 4.5: Layouts for 8, 16, and 24 Input Bits (not to scale)
4.3 Coefficient Bits Comparison (14 and 16)

<table>
<thead>
<tr>
<th></th>
<th>16,14,101,256</th>
<th>16,16,101,256</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Power (mW)</td>
<td>55.0</td>
<td>59.0</td>
</tr>
<tr>
<td>Max Sim (MHz)</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>RTL Cmplr (MHz)</td>
<td>116</td>
<td>116</td>
</tr>
<tr>
<td>Post-PAR (MHz)</td>
<td>114</td>
<td>110</td>
</tr>
<tr>
<td>Cell Area (µm²)</td>
<td>152,856</td>
<td>161,155</td>
</tr>
<tr>
<td>Core Area (µm²)</td>
<td>2,916,991</td>
<td>2,916,991</td>
</tr>
<tr>
<td>Core Utilization</td>
<td>5.2%</td>
<td>5.5%</td>
</tr>
<tr>
<td>Chip Area (µm²)</td>
<td>5,044,516</td>
<td>5,044,516</td>
</tr>
<tr>
<td>Mean SFDR (dBc)</td>
<td>45.5</td>
<td>45.5</td>
</tr>
</tbody>
</table>

Table 4.3: Results for 14- and 16-bit Filter Coefficients

As seen in table 4.3, a two-bit increase in coefficient bit width (from 14 to 16 bits) gave about a 5% increase in cell area and 7% increase in power. Post-PAR timing decreased by 3.5%. Average SFDR and frequency response remained almost exactly the same (see figures 4.6 and 4.7). The layouts are shown in figure 4.8.

![Figure 4.6: SFDR for 14 and 16 Bit Filter Coefficients (dBc)](image-url)
Figure 4.7: Freq. Response for 14 and 16 Bit Filter Coefficients (dBc)

Figure 4.8: Layouts for 14 and 16 Coefficient Bits
4.4 Filter Taps Comparison (33, 101, and 153)

<table>
<thead>
<tr>
<th></th>
<th>16_16_33_256</th>
<th>16_16_101_256</th>
<th>16_16_153_256</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Power (mW)</td>
<td>44.5</td>
<td>59.0</td>
<td>70.2</td>
</tr>
<tr>
<td>Max Sim (MHz)</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>RTL Cmplr (MHz)</td>
<td>116</td>
<td>116</td>
<td>116</td>
</tr>
<tr>
<td>Post-PAR (MHz)</td>
<td>109</td>
<td>110</td>
<td>97</td>
</tr>
<tr>
<td>Cell Area (µm²)</td>
<td>108,841</td>
<td>161,155</td>
<td>197,385</td>
</tr>
<tr>
<td>Core Area (µm²)</td>
<td>2,916,991</td>
<td>2,916,991</td>
<td>2,916,991</td>
</tr>
<tr>
<td>Core Utilization</td>
<td>3.7%</td>
<td>5.5%</td>
<td>6.8%</td>
</tr>
<tr>
<td>Chip Area (µm²)</td>
<td>5,044,516</td>
<td>5,044,516</td>
<td>5,044,516</td>
</tr>
<tr>
<td>Mean SFDR (dBc)</td>
<td>37.3</td>
<td>45.5</td>
<td>48.6</td>
</tr>
</tbody>
</table>

Table 4.4: Results for 33, 101, and 153 Filter Taps

As seen in table 4.4, increasing the number of filter taps by 206% (by 68 taps from 33 to 101) added about 33% power consumption. Increasing the taps again by 51% (by 52 taps from 101 to 153) added 19% power consumption.

From 33 to 101 taps, PAR timing stayed approximately the same. From 101 to 153 taps, PAR timing decreased by 12%.

From 33 to 101 taps, circuit area increased by 61%. From 101 to 153 taps, it increased by 26%.

From 33 to 101 taps, average SFDR increased by about 8 dB. From 101 to 153 taps, it increased by 3 dB (see also figure 4.9). As seen in figure 4.10, more taps gave smaller peaks in the frequency response. Layouts are shown in figure 4.11 (not to scale as the 153-tap design is actually larger by one pin on each side).
Figure 4.9: SFDR for 33, 101, and 153 Filter Taps (dBc)

Figure 4.10: Freq. Response for 33, 101, and 153 Filter Taps (dBc)
Figure 4.11: Layouts for 33, 101, and 153 Filter Taps (not to scale)
### 4.5 ROM Size Comparison (128, 256, and 512)

<table>
<thead>
<tr>
<th></th>
<th>16_16_101_128</th>
<th>16_16_101_256</th>
<th>16_16_101_512</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Power (mW)</td>
<td>56.6</td>
<td>59.0</td>
<td>60.0</td>
</tr>
<tr>
<td>Max Sim (MHz)</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>RTL Cmplr (MHz)</td>
<td>120</td>
<td>116</td>
<td>107</td>
</tr>
<tr>
<td>Post-PAR (MHz)</td>
<td>106</td>
<td>110</td>
<td>105</td>
</tr>
<tr>
<td>Cell Area (µm²)</td>
<td>161,101</td>
<td>161,155</td>
<td>161,250</td>
</tr>
<tr>
<td>Core Area (µm²)</td>
<td>2,916,991</td>
<td>2,916,991</td>
<td>3,171,676</td>
</tr>
<tr>
<td>Core Utilization</td>
<td>5.5%</td>
<td>5.5%</td>
<td>5.5%</td>
</tr>
<tr>
<td>Chip Area (µm²)</td>
<td>5,044,516</td>
<td>5,044,516</td>
<td>5,377,761</td>
</tr>
<tr>
<td>Mean SFDR (dBc)</td>
<td>43.8</td>
<td>45.5</td>
<td>46.5</td>
</tr>
</tbody>
</table>

Table 4.5: Results for 128, 256, and 512 ROM Words

As mentioned in section 3.1.3, for designs with the same bit width, the ROM size did not change, because 512 was the minimum number of words for the ROM generator. However, for each ROM size chosen, the address line increased by one bit. As seen in table 4.5, between the 128 ROM and 256 ROM, estimated power consumption did increase by 2.4%. Between the 256 and 512 ROM’s, it increased by 1.7%.

From 128 to 256 ROM, PAR timing improved by about 4%. However, from the 256 to 512 ROM, PAR timing was back down at 105 MHz.

Circuit area did not change between designs. However, chip area changed for the 512 ROM, because pins were added.

Between the 128 and 256-word ROM’s and 256 and 512-word ROM’s, average SFDR increased by about 1.7 and 1.0 dBC, respectively (see also figure 4.12). As seen in figure 4.13, the frequency response was almost exactly the same for all three designs. The layouts can be seen in figure 4.14.
Figure 4.12: SFDR for 128, 256, and 512 ROM Words (dBc)

Figure 4.13: Freq. Response for 128, 256, and 512 ROM Words (dBc)
Figure 4.14: Layouts for 128, 256, and 512 ROM Words (not to scale)
### 4.6 Summary of Design Parameter Variations (at a typical corner)

<table>
<thead>
<tr>
<th></th>
<th>8_8_101_256</th>
<th>16_16_33_256</th>
<th>16_14_101_256</th>
<th>16_16_101_128</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Power (mW)</td>
<td>22.0</td>
<td>44.5</td>
<td>55.0</td>
<td>56.6</td>
</tr>
<tr>
<td>Max Sim (MHz)</td>
<td>33</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>RTL Cmplr (MHz)</td>
<td>110</td>
<td>116</td>
<td>116</td>
<td>120</td>
</tr>
<tr>
<td>Post-PAR (MHz)</td>
<td>117</td>
<td>109</td>
<td>114</td>
<td>106</td>
</tr>
<tr>
<td>Cell Area (µm²)</td>
<td>68,156</td>
<td>108,841</td>
<td>152,856</td>
<td>161,101</td>
</tr>
<tr>
<td>Core Area (µm²)</td>
<td>818,880</td>
<td>2,916,991</td>
<td>2,916,991</td>
<td>2,916,991</td>
</tr>
<tr>
<td>Core Utilization</td>
<td>8.3%</td>
<td>3.7%</td>
<td>5.2%</td>
<td>5.5%</td>
</tr>
<tr>
<td>Chip Area (µm²)</td>
<td>2,082,249</td>
<td>5,044,516</td>
<td>5,044,516</td>
<td>5,044,516</td>
</tr>
<tr>
<td>Mean SFDR (dBc)</td>
<td>38</td>
<td>37</td>
<td>45</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 4.6: Results Summary by Increasing Power (part 1 of 2)

<table>
<thead>
<tr>
<th></th>
<th>16_16_101_256</th>
<th>16_16_101_512</th>
<th>16_16_153_256</th>
<th>24_24_101_256</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Power (mW)</td>
<td>59.0</td>
<td>60.0</td>
<td>70.2</td>
<td>111.8</td>
</tr>
<tr>
<td>Max Sim (MHz)</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>16</td>
</tr>
<tr>
<td>RTL Cmplr (MHz)</td>
<td>116</td>
<td>107</td>
<td>116</td>
<td>105</td>
</tr>
<tr>
<td>Post-PAR (MHz)</td>
<td>110</td>
<td>105</td>
<td>97</td>
<td>68</td>
</tr>
<tr>
<td>Cell Area (µm²)</td>
<td>161,155</td>
<td>161,250</td>
<td>197,385</td>
<td>310,421</td>
</tr>
<tr>
<td>Core Area (µm²)</td>
<td>2,916,991</td>
<td>3,171,676</td>
<td>2,916,991</td>
<td>7,059,224</td>
</tr>
<tr>
<td>Core Utilization</td>
<td>5.5%</td>
<td>5.5%</td>
<td>6.8%</td>
<td>5.5%</td>
</tr>
<tr>
<td>Chip Area (µm²)</td>
<td>5,044,516</td>
<td>5,377,761</td>
<td>5,044,516</td>
<td>10,208,025</td>
</tr>
<tr>
<td>Ckt Area (µm²)</td>
<td>276,000</td>
<td>276,000</td>
<td>348,000</td>
<td>563,000</td>
</tr>
<tr>
<td>Mean SFDR (dBc)</td>
<td>46</td>
<td>46</td>
<td>49</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 4.6: Results Summary by Increasing Power (part 2 of 2)

<table>
<thead>
<tr>
<th></th>
<th>RTL Compiler</th>
<th>Post-PAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crit. Path</td>
<td>Crit. Path</td>
<td></td>
</tr>
<tr>
<td>8_8_101_256</td>
<td>ROM address</td>
<td>ROM address</td>
</tr>
<tr>
<td>16_16_33_256</td>
<td>ROM Address</td>
<td>Qtr Wave Address</td>
</tr>
<tr>
<td>16_14_101_256</td>
<td>ROM address</td>
<td>Hilbert Filter</td>
</tr>
<tr>
<td>16_16_101_128</td>
<td>ROM Address</td>
<td>Qtr Wave Address</td>
</tr>
<tr>
<td>16_16_101_256</td>
<td>ROM Address</td>
<td>Hilbert Filter</td>
</tr>
<tr>
<td>16_16_101_512</td>
<td>ROM Address</td>
<td>Hilbert Filter</td>
</tr>
<tr>
<td>16_16_153_256</td>
<td>ROM Address</td>
<td>Hilbert Filter</td>
</tr>
<tr>
<td>24_24_101_256</td>
<td>Hilbert Filter</td>
<td>Hilbert Filter</td>
</tr>
</tbody>
</table>

Table 4.7: Critical Paths for All Designs
4.6.1 Power. All eight designs in table 4.6 are listed in order of increasing estimated switching power. The designs with higher bit widths, greater number of taps, and larger ROM sizes had higher estimated switching powers.

4.6.2 Timing. RTL Compiler timing reports at this typical characterization corner for all eight designs in table 4.6 showed positive slack and met the specification of 100 MHz. However, the post-place-and-route (post-PAR) timing analysis was below the specification (100 MHz) for the 153-tap and 24-input-bit designs. The 8-bit design gave the best post-PAR timing results (117 MHz) and the 24-bit design gave the worst (68 MHz). Later, in section 4.7, it will be seen that DRFM_16_14_101_256 also did not meet the timing specification for a worst-case corner.

As mentioned earlier, the timing values in the behavioral files for the standard cells and ROM’s were considered to be very rough estimates. Therefore, the simulation timing results are also considered to be very rough estimates. The 8-bit design was the best at 33 MHz, and the 24-bit design was the worst at 16 MHz. All of the 16-bit designs ran up to 25 MHz.

4.6.2.1 Critical Paths. Locations of the critical paths are given in table 4.7. The first column gives the RTL Compiler’s estimate of the critical path and the second column gives the estimate for post-PAR. The RTL compiler synthesized the Hilbert multipliers and adders into a carry-save adder tree and a final adder. Since the Hilbert filter had a long chain of gates (about 33 gates in one 16-bit design), it would be expected to be the critical path for all design cases. However, as seen in the table, the critical paths in the reports varied.

For most of the designs, RTL Compiler reported the critical path in the setup of the ROM address from the Fc-input to an address register (this path included a few adders). This was a critical path partly because the address line only had half a period to propagate. This was because the address was setup on the falling edge of the clock but clocked into the ROM on the rising edge. The inputs to the DSSM, also,
were mistakenly not registered. (This has been fixed in the mixer code in appendix A.) Therefore, the delays of the input enable (IE) and \( F_c \) in pads were added to the path.

Most of the post-PAR critical paths reported were in the Hilbert filter (from an input register within the filter to an output register). However, this was not the case for the 33-tap design where the Hilbert filter was smaller. It also was not the case for the 128-word ROM. The post-PAR critical paths for the 33-tap and 128-word ROM designs were in the quarter wave address setup. Again, this path only had half a period to propagate. However, there was only one main gate in the path, a MUX just before the input address to the ROM. RTL Compiler chose a MUX with a drive strength of only \( x1 \), and it had a large gate delay (e.g. \( 2.404 \) ns). The output of this MUX also was not registered before the ROM address port. (This has been fixed in the ROM wrapper code in appendix A.) Clock tree buffers also added some delay.

4.6.3 Area. All of the design chip areas were constrained by the number of I/O pads required. Therefore, for the designs in table 4.6, core utilization was only about 4-8% depending on the design. The 8-bit design had the highest utilization as it had the least amount of pins required. The 33-tap design had the lowest utilization, because it had the least area in the Hilbert filter.

As seen in table 4.6, the designs with the highest number of input bits and taps (24 bits and 153 taps) had the greatest cell area. The designs with lowest number of input bits and taps (8 bits and 33 taps) had the least cell area. The different ROM sizes, however, did not significantly contribute to a change in cell area, because the area of the ROM’s stayed the same size for all designs with the same input bit width. This was because the generator had a minimum size of 512 words.

4.6.4 SFDR. The greater variation in the designs occurred in the SFDR. As can be seen in figures 4.3 and 4.15, the 16- and 24-bit 101-tap DSSM’s followed a similar SFDR path. The larger ROM’s gave a higher peak. The 8-bit DSSM had a
lower and more level SFDR. The 33 and 153-tap DSSM show a more variable SFDR than the others. Although, the 8-bit shows a little variation as well (see figure 4.15).

4.6.5 Frequency Response. As can be seen in figure 4.16, the 33-tap response has the largest peaks of all the frequency responses. Its peak also diminishes over the sweep. It should be remembered, however, that the graph is only showing a small peak-to-peak range of about 0.7 dBC.

![SFDR for All Designs (highest to lowest avg) [Fs=15359999Hz; Fc=750000Hz]](image)

Figure 4.15: SFDR for All Designs (dBC)
Figure 4.16: Frequency Response for All Designs (dBc)
4.7 Three Process Corners Comparison (ss, tt, and ff)

<table>
<thead>
<tr>
<th></th>
<th>ss_1p08v_125c</th>
<th>tt_1p20v_25c</th>
<th>ff_1p32v_125c</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Power (mW)</td>
<td>48.5</td>
<td>55.0</td>
<td>66.0</td>
</tr>
<tr>
<td>Max Sim (MHz)</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>RTL Cmplr (MHz)</td>
<td>100</td>
<td>116</td>
<td>133</td>
</tr>
<tr>
<td>Post-PAR (MHz)</td>
<td>59</td>
<td>114</td>
<td>135</td>
</tr>
<tr>
<td>Cell Area (µm²)</td>
<td>158,463</td>
<td>152,856</td>
<td>152,587</td>
</tr>
<tr>
<td>Core Area (µm²)</td>
<td>2,916,991</td>
<td>2,916,991</td>
<td>2,916,991</td>
</tr>
<tr>
<td>Core Utilization</td>
<td>5.4%</td>
<td>5.2%</td>
<td>5.2%</td>
</tr>
<tr>
<td>Chip Area (µm²)</td>
<td>5,044,516</td>
<td>5,044,516</td>
<td>5,044,516</td>
</tr>
<tr>
<td>Mean SFDR (dBc)</td>
<td>45.5</td>
<td>45.5</td>
<td>45.5</td>
</tr>
</tbody>
</table>

Table 4.8: Results for Slow, Typical, and Fast Corners

<table>
<thead>
<tr>
<th></th>
<th>ss_1p32v_125c</th>
<th>tt_1p20v_25c</th>
<th>ff_1p08v_125c</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Compiler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crit. Path</td>
<td>HiIbert Filter</td>
<td>Qtr Wave Address</td>
<td></td>
</tr>
<tr>
<td>PAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crit. Path</td>
<td>HiIbert Filter</td>
<td>ROM Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HiIbert Filter</td>
<td>Qtr Wave Address</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.9: Critical Paths for Slow, Typical, and Fast Corners

Results for DRFM_16_14_101_256 are summarized in table 4.8 for slow, typical, and fast corners. Note that the operating temperatures for the slow and fast corners were higher than the typical corner. Therefore, the slow and fast cases would be expected to be a little slower than they would be at a lower temperature. Also, note that the operating voltages varied.

The operating voltage for the slow case (1.08V) was 10% lower than the typical case, and it’s power consumption was about 12% lower than the typical design. The operating voltage for the fast case (1.32V) was 10% higher than the typical case (1.20V), and it’s power consumption was 20% higher.

Post-PAR timing for the slow design was 48% slower than the typical design. Post-PAR timing for the fast design was 18% faster than the typical design. The unequal temperature operating temperatures may help to explain the unequal timing losses and gains that the slow and fast corners made from the typical corner, respec-
tively. As seen in table 4.9 the critical paths again varied between the same three locations as in section 4.6.2.1.

The areas for all three corner designs were about the same with the slow design taking the most area. With the slow corner, the synthesis tool may have created more area in its attempt to meet the timing specification. Average SFDR’s for all three corners were the same. As can be seen in figures 4.18 and 4.17, the frequency response and SFDR at each of the input frequencies for all of the corners were exactly the same.

![Graph showing SFDR for Slow, Typical, and Fast Corners (dBc)](image)

Figure 4.17: SFDR for Slow, Typical, and Fast Corners (dBc)
Figure 4.18: Freq. Response for Slow, Typical, and Fast Corners (dBc)
Figure 4.19: Layouts for Three Characterization Corners


## 4.8 Piped Hilbert Filter Comparison (24-bit)

Table 4.10: Results for DRFM\_24\_24\_101\_256, Unpiped and Piped

<table>
<thead>
<tr>
<th></th>
<th>Unpiped</th>
<th>Piped</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Power (mW)</td>
<td>111.8</td>
<td>89.5</td>
</tr>
<tr>
<td>Post-PAR (MHz)</td>
<td>68</td>
<td>85</td>
</tr>
<tr>
<td>Cell Area (µm²)</td>
<td>310,421</td>
<td>346,759</td>
</tr>
</tbody>
</table>

Table 4.11: Critical Paths for DRFM\_24\_24\_101\_256, Unpiped and Piped

<table>
<thead>
<tr>
<th></th>
<th>RTL Compiler</th>
<th>PAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crit. Path</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unpiped</td>
<td>Hilbert Filter</td>
<td>Hilbert Filter</td>
</tr>
<tr>
<td>Piped</td>
<td>ROM Address</td>
<td>Qtr Wave Address</td>
</tr>
</tbody>
</table>

Table 4.10 compares the results for a DRFM\_24\_24\_101\_256 without and with piped Hilbert filter multipliers. The 24-bit design was chosen to test the effect of piping the Hilbert multipliers since it was the only design which had the Hilbert filter as the critical path in both the RTL Compiler and post-PAR reports. The switching power estimate for the piped design was 20% less. It’s PAR timing was about 25% faster. As seen in table 4.11, the critical path for the piped design was no longer the Hilbert filter. However, it required 12% more cell area. Average SFDR’s were essentially the same. As shown in figures 4.21 and 4.20, frequency response and SFDR for the unpiped and piped Hilbert filter were almost exactly the same.
Figure 4.20: SFDR for Unpiped Vs. Piped Hilbert Filter (dBc)

Figure 4.21: Freq. Response for Unpiped Vs. Piped Hilbert Filter(dBc)
Figure 4.22: Layouts for DRFM\textsubscript{24,24,101,256}, Unpiped and Piped
4.9 Time and Frequency Analysis

Figure 4.23: FFT of Cosine in DRFM_16_16_101_256; Fc = 0.75 MHz
Figure 5.23(a) shows an FFT of the cosine signal from the DRFM\_16\_16\_101\_256 gate-level netlist. Here, the cosine had harmonic spurs. This is expected since the address increment of 50 for the ROM in the simulations was not a multiple of the ROM size of 256. Figure 5.23(b) shows the the cosine wave from the mathematical model of the design. It did not have the same harmonic spurs, and its highest spur was about 42 dBc lower than that for the gate-level netlist.

![Graph showing FFT of cosine signal](image1)

As can be seen in figures 4.24 and 4.25, the DSSM gate-level netlist simulated as designed. First, figure 4.24 shows the time-domain plots for the DSSM input and output with an input frequency of 0.75 Mhz and shift frequency of 0.75 MHz. The output shows about a 75% reduction in peak-to-peak amplitude from the input. Next, figure 4.25 shows the frequency spectrums of the signals. The output frequency was at 1.5 MHz as expected.

Usually, for the single-frequency plots, the input and output signals were plotted on one plot. For example, see figure 4.26. In the plots, the leftmost peak (at 0 dBc and shown in red) was the input signal and the rightmost peak (also at 0 dBc and in blue)
Figure 4.25: FFT of DSSM Input and Output from Figure 4.24
Input Freq. = 0.75 MHz, Fc = 0.75 MHz, Fout = 1.5 MHz
was the output signal. (Note that the signal peak dBc measurements were in reference to themselves and not to each other.) Each design had every input/output signal pair plotted for all 41 input frequencies. Figures 4.26 through 4.28 show samples of the plots for 8, 16, and 24 I/O bit DRFM gate-level netlists. It can be seen here that the lower sideband varies. The highest spurs in these plots are discussed in section 4.9.1.
Figure 4.26: FFT of DRFM_8_8_101_256; Fc = 0.75 MHz

(a) Fin=4.2 MHz, Fout=4.95 MHz
(b) Fin=4.35 MHz, Fout=5.10 MHz
(c) Fin=4.5 MHz, Fout=5.25 MHz
(d) Fin=4.65 MHz, Fout=5.40 MHz
DRFM_16_16_101_256_syn [Fs=15359999 Hz; Fc=750000 Hz; USB]

(a) Fin=3.75 MHz, Fout=4.50 MHz

(b) Fin=3.9 MHz, Fout=4.65 MHz

Figure 4.27: Single-Input Freq. Response for DRFM_16_16_101_256; Fc =0.75 MHz

DRFM_24_24_101_256_syn [Fs=15359999 Hz; Fc=750000 Hz; USB]

(a) Fin=3.15 MHz, Fout=3.90 MHz

(b) Fin=3.6 MHz, Fout=4.35 MHz

(c) Fin=4.05 MHz, Fout=4.80 MHz

Figure 4.28: FFT of DRFM_24_24_101_256; Fc =0.75 MHz

88
4.9.1 SFDR Plots of Models. Figures 4.29 and 4.30 show plots of SFDR for three models of each design: an RTL-code model, a gate-level netlist model, and a
Figure 4.30: SFDR for 33-/153-Tap Design Models (dBc)

Figure 4.31: SFDR Closeup of DRFM_8_8_101_256 Models (dBc)
Figure 4.32: SFDR Closeup of DRFM_{16,16,101,256} Models (dBc)

Figure 4.33: SFDR Closeup of DRFM_{16,16,153,256} Models (dBc)
mathematical model. Larger plots of select designs are shown in figures 4.31 through 4.33. The RTL-code and netlist data for these plots came from the test bench, which saved it when it ran the simulations. The math-model data, on the other hand, was generated and saved by MATLAB®. As seen in the plots, the RTL-code and gate-level netlist simulations followed each other closely. The math model’s SFDR, however, tended to be a little higher than the others and departed greatly near one frequency for the 16 and 24-bit I/O designs.

For the 8-bit design’s gate-level netlist plots, the highest spur was the DSSM’s lower sideband 20 times out of 41. For the other 21 input frequencies, the highest spur was at 0.75 MHz, which was the same frequency as the shift frequency. Figure 4.26, mentioned earlier, shows examples of these spurs at four different input frequencies. The varying lower sideband can be seen in these plots. The spur at 0.75 MHz did not vary more than 3 dBc over the sweep. However, the lower sideband spur varied by as much as 9.8 dBc. The 8-bit design’s math model similarly had the lower sideband as the highest spur for 24 of the 41 input frequencies. However, for 15 of the other 17 input frequencies, the highest spur occurred at the input frequency, and only 2 occurred at the shift frequency.

For the 101-tap designs with 16 and 24 I/O bits, the gate-level netlist plots had the largest spur at the lower sideband until approximately where the plots level off (see figure 4.29). The sloping portions of these plots are indicative of the lower sideband slowly getting smaller or larger, respectively from left to right. That means a smaller sideband gave larger SFDR. The level portions of the plots are where the highest spur became 2.25 MHz below the input frequency. Both of these spurs can be seen in the single-input-frequency plots for a 16-bit design in figure 4.27. Here, the lower sideband has begun to dominate again around 3.9 MHz. Figure 4.28 shows similar examples for the 24-bit design. Here, the lower sideband has begun to dominate again around 4.05 MHz.
For the 33-tap design, the gate-level netlist plot had the lower sideband as the largest spur except at 3.6 MHz. Here, the largest spur was again at 2.25 MHz below the input frequency. For the 153-tap design, the gate-level netlist plot had the lower sideband as the largest spur, except between 3 and 5.7 MHz. Here, the highest spur was at 2.25 MHz below the input frequency for 9 out of the 19 input frequencies.

![Figure 4.34: FFT of Math Model for DRFM_16_16_153_256 at Input Frequency 4.8 MHz](image)

For the 16 and 24-bit designs in figures 4.29 and 4.30, all the math-model SFDR plots also gradually came to a peak because of the gradual suppression of the lower sideband. However, there was no large spur in the frequency spectrum at 2.25 MHz below the input frequency to take over the SFDR, and there was no leveling off in the SFDR plots. The highest spur in the frequency spectrum was always the lower sideband except at 4.8 MHz input frequency in the 153-tap math model. Here, the lower sideband got suppressed far enough (to -103.2 dBC) that a spur located at the input frequency (4.8 MHz) and -93.68 dBC was now the highest (see figure 4.34).
Spurs were not checked for the RTL-code simulations. However, they were assumed to be similar to those for the RTL code since the SFDR plots were very close for the both of them.

In summary, for all the design gate-level netlists, the second dominant spur was at 2.25 MHz below the input frequency. 2.25 MHz is a multiple of 0.75 Hz, and the cosine wave had harmonics. Also, the math-model simulation did not have harmonics in its cosine wave, and this spur did not appear in the math model. Therefore, this spur is most likely due to the DDS’s harmonics.

4.9.2 Frequency Response Plots of Models. Frequency responses were also plotted for all three models of the designs. They are shown in figures 4.35 and 4.36. Larger plots of select designs are also shown in figures 4.37 through 4.39. The RTL and gate-level netlist simulation responses were very close. The math model also followed roughly the same frequency response path, although, again, it was a little higher.
Figure 4.35: Freq. Response for 101-Tap Design Models (dBc)
Figure 4.36: Freq. Response for 33-/153-Tap Design Models (dBc)

Figure 4.37: Freq. Response Closeup of DRFM_8_8_101_256 Models (dBc)
Figure 4.38: Freq. Response Closeup of DRFM_16_16_101_256 Models (dBc)

Figure 4.39: Freq. Response Closeup of DRFM_16_16_33_256 Models (dBc)
V. Conclusions and Future Work

5.1 Rad-Hard Comparison

RHBD normally has area and power penalties. Although, the RHBD technology had not been characterized for power (or delay), a 580% power penalty still was very high. If the estimate was too conservative by 20%, the power penalty would still be about 440%. Nevertheless, it is recommended that another comparison be made after the RHBD technology is characterized for power and delay to give a more reliable comparison.

The non-hardened design was not constrained to a fixed area or to a limited number of flip flops as in the RHBD design. However, since the non-hardened design area was constrained by the number of I/O pads, the two designs were somewhat comparable for area. There was no data for total circuit area in the RHBD design. Therefore, total circuit area could not be directly compared.

Aside from RHBD, a structured ASIC may have its own area and power penalties. For a future non-hardened vs. RHBD comparison, it would be ideal to have both circuits use standard cells or structured ASIC’s.

5.2 Parameters

There were four different parameters in the test cases for the DSSM: input bits, coefficient bits, number of filter taps, and ROM size. The findings and conclusions on the parameter variations are summarized below.

5.2.1 Input Bits. First, less input bits mean faster clock speeds. However, more bits lead to greater SFDR. Of course, more bits also means greater area and more power consumption. If the goal was not to increase area for radiation testing, the
fewest number of input bits could be chosen that still gives the minimum acceptable SFDR [6]. This would give the greatest speed for SFDR.

5.2.2 Coefficient Bits. For a 16-input-bit, 101-tap, 256-ROM DSSM, going from 14 to 16 coefficient bits gives less than an 8% increase in area, power, and delay with essentially no change to SFDR. Again, if maximum area were not a concern, for the conditions given, the 14-bit coefficients could be used. However, for other parameter variations, coefficient bit variations and costs should be tested or calculated. Also, if the lower sideband were better suppressed, the SFDR might change more.

5.2.3 Number of Taps. A greater number of filter taps gives a higher average SFDR. As expected, it also has less ripple in the frequency response. Though not tested here, the frequency response should also have a wider bandwidth with sharper transitions. Of course, the cost of more taps is greater area and power. Again, if maximum area were not the goal, the fewest number of taps needed to give the desired SFDR could be chosen.

5.2.4 ROM Size. For 16 input bits and 101 taps, increasing ROM size increases the maximum SFDR for a select range of input frequencies. This is because a second spur is most likely due to harmonics in the DDS sine and cosine. A larger ROM gives greater resolution and should produce less harmonics. The same ROM size with varying numbers of address lines does slightly change power consumption. Since there is a minimum ROM size of 512, all of it should be used to get the maximum resolution and SFDR out of it. The cost is only a slight increase in power.

5.3 Area and Power

More bits, taps, and larger ROM give a higher SFDR. More bits and taps also lead to greater area and power. Power and area increase somewhat proportionally.

The area was limited by the large number of I/O pins. These could be reduced by multiplexing the inputs and outputs. However, this is not recommended as it
would reduce speed. Some of the test outputs may not be needed either, though perhaps they could be accessed through a scan chain.

5.4 Timing

A dense circuit area was a higher goal than speed in this thesis and [1]. Normally, DRFM’s need to be wideband and thus fast. A couple ways to increase the speed of the DSSM would be, as already mentioned, to reduce the input bits and/or to channelize the inputs. Of course, fixing and improving the critical paths outside the Hilbert filter will improve the timing results slightly for the designs that had critical paths outside the Hilbert filter.

5.5 Critical Paths

If the critical path is the Hilbert filter, it can be improved by piping between its multipliers and adders. The ROM address setup path will be improved with the corrected top-level VHDL code in appendix A that registers the input signals. To reduce the delay in the quarter-wave select path, it’s MUX could manually be resized in the netlist to one with greater drive strength. The output of the MUX should also be registered. The ROM wrapper code in appendix A has been changed to reflect this also. Registering may help improve this path.

5.6 SFDR

In this thesis, SFDR is most often limited by the lower sideband, the suppression of which varies depending on the input frequency. This is believed to be due to poor balance in the I and Q signals in the Hilbert Filter. H. Scott Axtell has worked on solving this problem in his master’s thesis [48]. Axtell’s work was not available at the time of this thesis. The DSSM behavioral code in Appendix A could be re-implemented with Axtell’s improvements to see what difference it will make in the spurs.
For the 16 and 24-bit designs, another dominant spur is at 2.25 MHz below the input frequency. As already mentioned, this is most likely due to a DDS harmonic. Increasing ROM size increases DDS resolution and reduces this spur. Therefore, for these bit sizes, the largest ROM size needed to meet SFDR specification should be chosen, if possible.

In the 8-bit design, the other dominant spur is at 0.75 MHz. Reducing this spur was not investigated. If this spur limits SFDR specifications, then larger I/O bit width may be needed.

5.7 Math Model

The math model of the cosine wave did not have the harmonics that were present in the gate-level netlist simulations. This suggests that the model was not an accurate representation. This could be recoded to provide a closer comparison.

5.8 Hilbert Filter and Mixer

The Hilbert HDL code that was used had a multiplier for every coefficient. Since the Hilbert Filter is anti-symmetric and linear phase, it only needs half the amount of multipliers. (The absolute value of the first half of the coefficients and the second half are the same.) Again, if the goal was not to maximize circuit area, it could be reduced by using half the multipliers [16].

As mentioned, the Hilbert filter coefficients were not windowed. This caused ripples in the frequency response. The coefficients should be windowed to remove the ripples. Also, if needed, we saw the Hilbert filter can be sped up with piping the multipliers.

The output of the mixer circuit used two adders (one to add, one to subtract). Again, if area was not being maximized, these could be coded so they are combined into one selectable adder/subtractor.
The ROM macros in this project had a chip enable and extra timing margin adjustment (EMA) pins on them. The chip enable was hard-wired “on” since the ROM was always in use. The EMA could be wired to I/O pads, if desired. Although, this would add more pad area.

5.9 VHDL Code Fixes

The fixes that have been made to the VHDL code in appendix A are as follows. In the top-level code, the inputs have now been registered. In the mixer code, Fe_in has been bit extended by two bits for the address counter. Also in the mixer code, Rom_cos_out now has a registered version of cos, which it did not before. Finally, in the ROM wrapper, the address has been registered right before the ROM.

5.10 Characterization Corners

As can be seen in our timing results for the three corner versions, characterization corners make a big difference in timing. If synthesizing to a corner, it can even affect area. Normally, when fabrication is the goal, all characterization corners of concern should be tested before fabrication.

5.11 Parameterization

Having parameters in the scripts made it easy to change values. Not much is suggested for changing the parameterization variables in the scripts or VHDL files. The ROM and cosine-output bit width generics, which were not used, could be removed.

5.12 Automation

The same designs were resynthesized a few times using part or all of the same scripts. This proves the usefulness of scripts in the automation process.
When changing the parameters, the greatest work is upfront in creating new source files (module HDL files and supporting ROM files). If desired, at least the VHDL file generation part could be streamlined even further with a central C/C++ program to take care of generating all of the files.

5.13 Automation Caveats

Automation is a time-saver. However, the design still needs to be double checked, and some parts of the design may need hands-on adjustments and oversight. For example, for at least one design iteration, there were some dangling power rails in one or more designs caused by a ROM macro. And, this was not noticed right away. Likewise, one may want to view multiple design placement suggestions before choosing one in the placement process. Of course, input files and floorplan sizes need to be checked for each design before synthesis as well. With all of that said, scripts are still useful for all points in the design where involvement is not needed.

The file naming convention and file organization are key in the automation process. The scripts should be set up for design names that include all parameter variations. Otherwise, resynthesizing a design with the same name but a different parameter (like a piped Hilbert filter) can end up writing over the files for the old design. This suggests that it would be a good idea to backup some of the designs files periodically, although this can take up a lot of disk space.

5.14 Timing Characterization of Behavioral Files

The simulation speed limitations were not considered accurate as the I/O, standard cell, and ROM Verilog behavioral files only had default timing values (e.g. 1.0 ns). If greater timing accuracy is desired in the simulations, these values could be updated. This would have to be done by simulating some or all of the cells in an analog simulator such as that found in Cadence Virtuoso®, finding the rise and fall times, and entering them back into the Verilog files.
5.15 Next Steps for Fabrication

If these circuits were to be fabricated, they would need a few changes. First of all, there were a limited number of voltage supply pads on the non-hardened circuits in order to be comparable to the RHBD circuit. For manufacture, the number of these pads might need to be increased. (This will increase area.) Also, the placement grid of the design was not on a manufacturing grid. If the circuit needs to be conformed to a manufacturing grid, this should be done. The chip may also need to conform to a standard chip size. If there are dangling power rails, as mentioned in section 5.13, they could also be fixed. Finally, adding metal fill would need to be added to our layout process.

After these changes, the design would be imported into a physical layout program like Cadence Virtuoso®. Here, the design could be extracted and a SPICE Simulation run on at least part of the circuit. The SPICE simulation will give a more accurate analysis of the timing. It also possibly could be used to get a post-PAR power estimate. After all of the appropriate steps in the final layout phase, the design could be converted to a GDS II format and sent for tapeout.

5.16 Radiation Testing the Non-hardened DSSM

The non-hardened DSSM could be tested for total-dose effects in a Cobalt-60 test as in [1]. It could also tested for single-event effects (i.e. soft error rates) through heavy-ion tests. However, it might be hard to characterize the soft error rates in the logic as parts of the circuit could be masked [32]. If it were desired to test the technology used for SET pulse widths for given LET’s, each gate should be tested separately. As mentioned in section 2.11, this could be done by placing multiple copies of one type of gate in a chain (to create a target) and placing a SET pulse capture circuit at its output.
5.17 Future Guard Ring Design

Guard rings and their effectiveness for RHBD were explored in chapter 2. A guard ring’s minimum width scales down with technology. If the minimum width is used, there is less charge mitigation. One could offset this loss by maintaining the same width, but this would, of course, cost more relative area. Also, it is not certain what percentage of charge mitigation is lost due to this versus due to the increased bipolar amplification.

To reduce the well resistance between the guard ring and the diffusion, the ring should be as close as possible to the diffusion (as allowed by design rules) [27]. Of course, ideally there would be a guard ring around each transistor to mitigate the charge sharing and parasitic effects [26, 27]. However, this is not always practical as the cost in area may be too high. This extra area could be minimized by only selectively hardening critical nodes [32]. One or more algorithms have been proposed for finding the sensitivity of each node [32, 33].

For best results, it is suggested that the guard rings use high-density well contacts [34]. Greater ring widths are also better. However, as mentioned in section 2.9, the benefit diminishes after a certain point [27, 33, 47].

5.18 Radiation Testing Guard Rings

Testing could be done of guard ring design. As already mentioned, guard rings reduce in effectiveness for decreasing technologies due to shrinking minimum widths and increased bipolar amplification. Improvement of charge mitigation just based on increasing guard ring widths could be tested in a particular technology by testing for SET pulse widths. This could be done by creating multiple target circuits made of chains of inverters, each chain using a different guard ring width. On the other hand, there could just be a few target circuits with switchable ring widths as done in [36]. However, it first should be confirmed what research there is, if any, on what percentage of the loss in guard ring effectiveness is due to the increased bipolar amplification.
Appendix A. VHDL Code and Top-Level Generator

A.1 DRFM Top-Level Template

[Names of Vendor’s I/O Pads have been removed]

---DRFM_NInum_COEFFnum_TAPnum_ROMnum.vhd
---DRFM_JNP_COEFF_TAP_ROM.vhd

--- The DRFM.vhd is fully parameterizable (NI, NC, NCO, & ROM in names repr. nos.)

--- For changing NI:
--- * Generate new DRFM.vhd file in ./src with new NI, NC, & NCO values
--- (use script ./src/DRFM_gen)
--- * Run rom, wtm (pipelined), and cla generators in ./gen with new bit values
--- Rename new files like this:
---   (cos/sin)Bits_NC_ROM.vhd, wtm_NILNC1.vhd, add2_NILNC.vhd, & claN1.vhd
--- (only the (cos/sin)Bits_NC_ROM.txt file is needed from the rom files)
--- * Generate new vendor ROM files in ./rom/rom_NC_ROM (create directory):
---   Name the entity e.g. ipcosRom_NC_ROM
--- * Copy file and update ROM ports in ./src/rom_wrapper_ROM.vhd
---   and rename to rom_wrapper_ROM.vhd if ROM=512 or smaller
---
--- For changing Hilbert coefficient bit width or no. of taps:
--- * Generate Hilbert vhdl file in ./gen directory
--- * Copy file and change Hilbert component name in ./src/DRFM.vhd
---
--- For changing ROM size used or NC:
--- * Generate new DRFM.vhd file in ./src with new NF value (NF = log2(ROM))
--- (use script ./src/DRFM_gen)
--- * Generate new ROM files in ./gen and ./rom/rom_NC_ROM
--- (Modify romBits.txt files to length of vendor ROM if less than 512)
--- * Copy file and update ROM ports for ./src/rom_wrapper_ROM.vhd for ROM<=512,
---   because the vendor ROM used has a min. size of 512
---
--- For changing NCO:
--- * Generate new DRFM.vhd file in ./src with new NCO value
--- (use script ./src/DRFM_gen)

library ieee.work;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
pragma synthesis_off
--use work.prim_io.all; -- compiled vendor i/o behavioral package
use ieee.vital_timing.all;
use ieee.vital_primitives.all;
pragma synthesis_on

entity DRFM_NInum_COEFFnum_TAPnum_ROMnum is
  generic (
    NF : integer := NFnum;  -- number of Fc bits = log2(num_Words_in_Rom)
    NC : integer := NCnum;  -- number of sine/cosine bits
    NCO : integer := NCOnum;  -- number of output cosine bits
    NI : integer := NInum  -- number of input bits
  );
port (
architecture toplevel of DRFM_Nisnum_COEFFnum_TAPnum_ROMnum is

-- wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
component [input pad name] --input pad
  [generics here]
  
  port(
    Y: out std_logic;
    P: in std_logic;
    IE: in std_logic
  );
end component;

component [Output Pad Name] --output pad
  [generics here]
  
  port(
    P: out std_logic;
    A: in std_logic
  );
end component;

--- SIGNALS:

signal adc, mem, mx_out, sd, so, rcout, Sfinal : std_logic_vector(NI-1 downto 0);
signal mx_sel, rst, add_sub : std_logic;
signal fc : std_logic_vector(NF-1 downto 0);
signal ADC_IN : std_logic_vector(NI-1 downto 0);
signal IE : std_logic;
signal CLK_IN : std_logic;
signal MUX_S : std_logic;
signal RST_IN : std_logic;
signal ADD_SUB_S : std_logic;
signal Fc_IN : std_logic_vector(NF-1 downto 0);
signal MUX_OUT : std_logic_vector(NI-1 downto 0);
signal HIL_I_OUT : std_logic_vector(NI-1 downto 0);
signal HIL_Q_OUT : std_logic_vector(NI-1 downto 0);
signal ROM_COS_OUT : std_logic_vector(NCO-1 downto 0);
signal DRFM_OUT : std_logic_vector(NI-1 downto 0);
signal ONE,DVDD,DVSS,VDD, VSS : std_logic;

begin

--- CORE INSTANTIATIONS:

mux : process(adc, mem, mx_sel)
begin
  if mx_sel = '0' then
    mx_out <= adc;
  else
    mx_out <= mem; -- Replace mem with an input from memory in an actual DRFM.
  end if;
end process mux;

hil_filter : HILBERT,NTTAPnum,NCQOEFFnum
  port map (SI=>mx_out, CLK=>CLK_IN, SD=>sd, SO=>so, regOut=>mem);

mix : mixer
  generic map(NF=>NF, NC=>NC, NCO=>NCO, NI=>NI)
  port map
(rst=>rst, clk=>CLKIN, add_sub=>add_sub, Fc=>fc, Isig=>sd, Qsig=>so, Sfinal=>Sfinal, rom_cos_out=>rout);

io_ports : process(CLKIN) -- clock inputs/outputs
begin
  if CLKIN'event and CLKIN='1' then
    adc <= ADCIN;
    mx_sel <= MUXS;
    rst <= RSTIN;
    add_sub <= ADDSUBS;
    MUXOUT <= mxout;
    -- I/O port
    HILQOUT <= so;
    -- I/O port
    ROMCOSOUT <= rcout;
    -- I/O port
    DRFMOUT <= Sfinal;
  end if;
end process io_ports;

fc_port : process(CLKIN) -- clock fc input on falling edge
begin
  if CLKIN'event and CLKIN='0' then
    fc <= FcIN;
  end if;
end process fc_port;

-- PHILQOUT <= HILQOUT;
-- I/O PAD INSTANTIATIONS:

end toplevel;
A.2 DRFM Top-Level VHDL Generator

#!/bin/csh
# Set parameters
setenv INP 16 # number of input bits
setenv COEFF 14 # number of coeff. bits
setenv TAP 101 # number of filter taps
setenv ROM 256 # ROM word length
setenv NF 8 # number of Fc factor bits = log2(ROM)

setenv NC $INP # number of sin/cos bits, making same as INP
setenv NCO $INP # number of cos out bits, making same as INP

perl -pi -e "s/COEFFnum/$COEFF/g" ./$D_ENTITY.vhd
perl -pi -e "s/TAPnum/$TAP/g" ./$D_ENTITY.vhd
perl -pi -e "s/ROMnum/$ROM/g" ./$D_ENTITY.vhd
perl -pi -e "s/NFnum/$NF/g" ./$D_ENTITY.vhd
perl -pi -e "s/NCnum/$NC/g" ./$D_ENTITY.vhd
perl -pi -e "s/NCOnum/$NCO/g" ./$D_ENTITY.vhd

# Make a new DRFM.vhd file and update names and variables

A.3 Hilbert_NT101_NC24 from Pemberton's Generator

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity HILBERT_NT101_NC24 is
generic (
    NI : integer := 16;
    NE : integer := 6);
port ( 
    SI : in std_logic_vector(NI-1 downto 0);
    CLK : in std_logic;
    SD : out std_logic_vector(NI-1 downto 0);
    SO : out std_logic_vector(NI-1 downto 0);
    regOut : out std_logic_vector(NI-1 downto 0)
);
end HILBERT_NT101_NC24;

architecture Behavioral of HILBERT_NT101_NC24 is
type HTCARRAY is array(100 downto 0) of std_logic_vector(23 downto 0);
signal HTCARRAY : HTCARRAY;
type INPUTARRAY_TYPE is array(100 downto 0) of std_logic_vector(NI-1 downto 0);
signal INPUTARRAY : INPUTARRAY_TYPE;

begin

begin coefficient array

SC = 8388606.000000000

   HTCARRAY(1) <= conv_std_logic_vector(-108987,24);
   HTCARRAY(3) <= conv_std_logic_vector(-113625,24);
   HTCARRAY(5) <= conv_std_logic_vector(-118675,24);
   HTCARRAY(7) <= conv_std_logic_vector(-124195,24);
   HTCARRAY(9) <= conv_std_logic_vector(-130253,24);
   HTCARRAY(11) <= conv_std_logic_vector(-136933,24);
   HTCARRAY(13) <= conv_std_logic_vector(-144334,24);
   HTCARRAY(15) <= conv_std_logic_vector(-152582,24);
   HTCARRAY(17) <= conv_std_logic_vector(-161829,24);
   HTCARRAY(19) <= conv_std_logic_vector(-172270,24);
   HTCARRAY(21) <= conv_std_logic_vector(-184151,24);
   HTCARRAY(23) <= conv_std_logic_vector(-197791,24);
   HTCARRAY(25) <= conv_std_logic_vector(-213615,24);
   HTCARRAY(27) <= conv_std_logic_vector(-232190,24);
   HTCARRAY(29) <= conv_std_logic_vector(-254303,24);
   HTCARRAY(31) <= conv_std_logic_vector(-281072,24);
   HTCARRAY(33) <= conv_std_logic_vector(-314139,24);
   HTCARRAY(35) <= conv_std_logic_vector(-356024,24);
   HTCARRAY(37) <= conv_std_logic_vector(-410797,24);
   HTCARRAY(39) <= conv_std_logic_vector(-485487,24);
   HTCARRAY(41) <= conv_std_logic_vector(-593373,24);
   HTCARRAY(43) <= conv_std_logic_vector(-762908,24);
   HTCARRAY(45) <= conv_std_logic_vector(-1068071,24);
   HTCARRAY(47) <= conv_std_logic_vector(-1780118,24);
   HTCARRAY(49) <= conv_std_logic_vector(-5340353,24);
   HTCARRAY(51) <= conv_std_logic_vector(5340353,24);
   HTCARRAY(53) <= conv_std_logic_vector(1780118,24);
   HTCARRAY(55) <= conv_std_logic_vector(1068071,24);
   HTCARRAY(57) <= conv_std_logic_vector(762908,24);
   HTCARRAY(59) <= conv_std_logic_vector(593373,24);
   HTCARRAY(61) <= conv_std_logic_vector(485487,24);
   HTCARRAY(63) <= conv_std_logic_vector(410797,24);
   HTCARRAY(65) <= conv_std_logic_vector(356024,24);
   HTCARRAY(67) <= conv_std_logic_vector(314139,24);
   HTCARRAY(69) <= conv_std_logic_vector(281072,24);
   HTCARRAY(71) <= conv_std_logic_vector(254303,24);


111
CARRAY(73) <= conv_std_logic_vector(232190,24);
CARRAY(75) <= conv_std_logic_vector(213615,24);
CARRAY(77) <= conv_std_logic_vector(197791,24);
CARRAY(79) <= conv_std_logic_vector(184151,24);
CARRAY(81) <= conv_std_logic_vector(172270,24);
CARRAY(83) <= conv_std_logic_vector(161829,24);
CARRAY(85) <= conv_std_logic_vector(152582,24);
CARRAY(87) <= conv_std_logic_vector(144334,24);
CARRAY(89) <= conv_std_logic_vector(136933,24);
CARRAY(91) <= conv_std_logic_vector(130253,24);
CARRAY(93) <= conv_std_logic_vector(124195,24);
CARRAY(95) <= conv_std_logic_vector(118675,24);
CARRAY(97) <= conv_std_logic_vector(113625,24);
CARRAY(99) <= conv_std_logic_vector(108987,24);

-- capture the inputs
process (CLK, SI, INPUTARRAY)
begin
  if rising_edge(CLK) then
    INPUTARRAY(0) <= SI;
    for n in 1 to 100 loop
      INPUTARRAY(n) <= INPUTARRAY(n-1);
    end loop;
  end if;
end process;

-- delayed input to match output
process (CLK, INPUTARRAY)
begin
  if rising_edge(CLK) then
    SD <= INPUTARRAY(50);
    SD(NI-1 downto 0) <= INPUTARRAY(50)(NI-1) & INPUTARRAY(50)(NI-1 downto 1);
    regOut <= INPUTARRAY(100);
  end if;
end process;

-- process the quad-phase signal
process (CLK, INPUTARRAY)
  variable SUMVAL : std_logic_vector(24+NI+NE-1 downto 0);
  variable MULTVAL : std_logic_vector(24+NI+NE-1 downto 0);
begin
  if rising_edge(CLK) then
    SUMVAL := (others => '0');
    for n in 100 downto 0 loop
      if (n rem 2) /= 0 then
        MULTVAL(24+NI-1 downto 0) := INPUTARRAY(n) + CARRAY(n);
        MULTVAL(24+NI+NE-1 downto 24+NI) := (others => MULTVAL(24+NI-1));
        SUMVAL := SUMVAL + MULTVAL;
      end if;
    end loop;
    SO <= SUMVAL(24+NI-2 downto 24-1);
    SO <= SUMVAL(24+NI-1 downto 24);
  end if;
end process;
end Behavioral;
A.4 Piped Version of Hilbert_NT101_NC24

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

downto NC24 is
  
entity HILBERT_NT101_NC24 is
generic (  
  NI : integer := 16;  
  NE : integer := 6 );  

port (  
  SI : in std_logic_vector(NI downto 0);  
  CLK : in std_logic;  
  SD : out std_logic_vector(NI downto 0);  
  SO : out std_logic_vector(NI downto 0);  
  regOut : out std_logic_vector(NI downto 0));  
end HILBERT_NT101_NC24;

architecture Behavioral of HILBERT_NT101_NC24 is

type HTCARRAY is array(100 downto 0) of  
  std_logic_vector(23 downto 0);

signal HTCARRAY: HTCARRAY;

type INPUTARRAY_TYPE is array (100 downto 0) of  
  std_logic_vector(NI downto 0);

signal INPUTARRAY : INPUTARRAY_TYPE;

type ARRAY is array (100 downto 0) of  
  std_logic_vector(24+NI+NE downto 0);

signal ARRAY: MULTARRAY_TYPE;

begin  
  begin coefficient array
  SC = 8388606.000000000
  CARRAY(1) <= conv_std_logic_vector(-108987,24);
  CARRAY(3) <= conv_std_logic_vector(-113625,24);
  CARRAY(5) <= conv_std_logic_vector(-118675,24);
  CARRAY(7) <= conv_std_logic_vector(-124195,24);
  CARRAY(9) <= conv_std_logic_vector(-130253,24);
  CARRAY(11) <= conv_std_logic_vector(-136933,24);
  CARRAY(13) <= conv_std_logic_vector(-144334,24);
  CARRAY(15) <= conv_std_logic_vector(-152582,24);
  CARRAY(17) <= conv_std_logic_vector(-161829,24);
  CARRAY(19) <= conv_std_logic_vector(-172270,24);
  CARRAY(21) <= conv_std_logic_vector(-184151,24);
  CARRAY(23) <= conv_std_logic_vector(-197791,24);
  CARRAY(25) <= conv_std_logic_vector(-213615,24);
  CARRAY(27) <= conv_std_logic_vector(-232190,24);
  CARRAY(29) <= conv_std_logic_vector(-254303,24);
  CARRAY(31) <= conv_std_logic_vector(-281072,24);
  CARRAY(31) <= conv_std_logic_vector(-314139,24);
  CARRAY(33) <= conv_std_logic_vector(-356024,24);
  CARRAY(37) <= conv_std_logic_vector(-410797,24);
  CARRAY(39) <= conv_std_logic_vector(-485487,24);
  CARRAY(41) <= conv_std_logic_vector(-593373,24);
  CARRAY(43) <= conv_std_logic_vector(-762908,24);
  CARRAY(45) <= conv_std_logic_vector(-1068071,24);
  CARRAY(47) <= conv_std_logic_vector(-1780118,24);
  CARRAY(49) <= conv_std_logic_vector(-5340353,24);
  CARRAY(51) <= conv_std_logic_vector(-172270,24);
  CARRAY(53) <= conv_std_logic_vector(-184151,24);
  CARRAY(55) <= conv_std_logic_vector(-197791,24);
  CARRAY(57) <= conv_std_logic_vector(-213615,24);
  CARRAY(59) <= conv_std_logic_vector(-232190,24);
  CARRAY(61) <= conv_std_logic_vector(-254303,24);
  CARRAY(63) <= conv_std_logic_vector(-281072,24);
  CARRAY(65) <= conv_std_logic_vector(-356024,24);

end;
CARRAY(67) <= conv_std_logic_vector(314139,24);
CARRAY(69) <= conv_std_logic_vector(281072,24);
CARRAY(71) <= conv_std_logic_vector(254303,24);
CARRAY(73) <= conv_std_logic_vector(232190,24);
CARRAY(75) <= conv_std_logic_vector(213615,24);
CARRAY(77) <= conv_std_logic_vector(197791,24);
CARRAY(79) <= conv_std_logic_vector(184151,24);
CARRAY(81) <= conv_std_logic_vector(172270,24);
CARRAY(83) <= conv_std_logic_vector(161829,24);
CARRAY(85) <= conv_std_logic_vector(152582,24);
CARRAY(87) <= conv_std_logic_vector(144334,24);
CARRAY(89) <= conv_std_logic_vector(136933,24);
CARRAY(91) <= conv_std_logic_vector(130253,24);
CARRAY(93) <= conv_std_logic_vector(124195,24);
CARRAY(95) <= conv_std_logic_vector(118675,24);
CARRAY(97) <= conv_std_logic_vector(113625,24);
CARRAY(99) <= conv_std_logic_vector(108987,24);

-- capture the inputs
process (CLK, S1, INPUT_ARRAY)
begin
    if rising_edge(CLK) then
        INPUT_ARRAY(0) <= S1;
        for n in 1 to 100 loop
            INPUT_ARRAY(n) <= INPUT_ARRAY(n-1);
        end loop;
    end if;
end process;

-- delayed input to match output
process (CLK, INPUT_ARRAY)
begin
    if rising_edge(CLK) then
        -- SD <= INPUT_ARRAY(51);    -- index added 1 b/c of extra mult register
        SD(NI−1 downto 0) <= INPUT_ARRAY(51) (NI−1) & INPUT_ARRAY(51) (NI−1 downto 1);
        regOut <= INPUT_ARRAY(100);
    end if;
end process;

-- multiply by the coefficients
process (CLK, INPUT_ARRAY)
variable MULT_VAL : std_logic_vector(24+NI+NE−1 downto 0);
begin
    if rising_edge(CLK) then
        for n in 100 downto 0 loop
            if (n rem 2) /= 0 then
                MULT_VAL(24+NI−1 downto 0) := INPUT_ARRAY(n) & CARRAY(n);
                MULT_VAL(24+NI+NE−1 downto 24+NI) := (others => MULT_VAL(24+NI−1));
            end if;
            MULT_ARRAY(n) <= MULT_VAL;
        end loop;
    end if;
end process;

-- sum the clocked multiplications
process (CLK, MULT_ARRAY)
variable SUM_VAL : std_logic_vector(24+NI+NE−1 downto 0);
begin
    if rising_edge(CLK) then
        SUM_VAL := (others => '0');
        for n in 100 downto 0 loop
            if (n rem 2) /= 0 then
                SUM_VAL := SUM_VAL + MULT_ARRAY(n);
            end if;
        end loop;
    end if;
end process;
SO <= SUMVAL(24+NI-2 downto 24-1);
SO <= SUMVAL(24+NI-1 downto 24);
end if;
end process;
end Behavioral;
A.5 Mixer

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

-- The center frequency is determined by the scaling factor
-- (Fc) and the clock frequency (clk_freq). Fc is the same bit
-- length as the ROM address which is dependent on the number
-- of words in the ROM (nWords_in_ROM). To calculate the
-- center frequency use:
--
-- Center frequency = Fc * clk_freq / (4 * nWords_in_ROM)
--
-- where, 1 <= Fc <= nWords_in_ROM

entity mixer is
  generic
  (NF : integer := 8; -- number of Fc bits = log2(nSamples_in_ROM)
   NC : integer := 16; -- number of sine/cosine bits
   NCO : integer := 16; -- number of output cosine bits
   NI : integer := 16 -- number of input bits)
  port
  (rst, clk : in std_logic;
   add_sub : in std_logic;
   Fc : in std_logic_vector(NF-1 downto 0);
   Isig, Qsig : in std_logic_vector(NI-1 downto 0);
   Sfinal : out std_logic_vector(NI-1 downto 0);
   rom_cos : out std_logic_vector(NCO-1 downto 0)
  );
end mixer;

architecture Behavioral of mixer is

component rom_wrapper
  generic
  (NF : integer := 8; -- number of address bits
   NC : integer := 16 -- number of sine/cosine bits)
  port
  (clk : in std_logic;
   addr : in std_logic_vector(NF+1 downto 0); -- 2 bits for qtr. wave select
   sin : out std_logic_vector(NC-1 downto 0);
   cos : out std_logic_vector(NC-1 downto 0)
  );
end component;

component wtm
  port
  (A : in std_logic_vector(NI-1 downto 0);
   B : in std_logic_vector(NC-1 downto 0);
   Z : out std_logic_vector(NI+NC-1 downto 0);
   CLK : in std_logic)
end component;

signal load_addr, addr : std_logic_vector(NF+1 downto 0);
signal sin, cos : std_logic_vector(NI-1 downto 0);
signal sin_d, cos_d : std_logic_vector(NI-1 downto 0);
signal Smult1, Smult2 : std_logic_vector(NI+NC-1 downto 0);
signal mult1,mult2 : std_logic_vector(NI+NC−1 downto 0);
signal Sum1,Sum2 : std_logic_vector(NI−1 downto 0);

begin

update_addr : process(clk)
begin
  if clk'event and clk='0' then -- updates on falling edge
    if rst = '1' then
      addr <= (others => '0');
    else
      addr <= addr + ("00" &Fc);
    end if;
  end if;
end process update_addr;

roms : rom_wrapper
generic map(NF=>NF, NC=>NC)
port map
  (clk=>clk, addr=>addr, sin=>sin, cos=>cos);

clk_rom_output : process(clk)
begin
  if clk'event and clk='1' then
    sin_d <= sin;
    cos_d <= cos;
  end if;
end process clk_rom_output;

--Note: replaced 's' with pipelined Wallace Tree Mult. component
-- to try for greater speed
--Smult1 <= Isig * cos_d;
--Smult2 <= Qsig * sin_d;

wtm1 : wtm port map(A=>Isig, B=>cos_d, Z=>Smult1, CLK=>clk);
wtm2 : wtm port map(A=>Qsig, B=>sin_d, Z=>Smult2, CLK=>clk);

sum_prod : process(clk)
begin
  if clk'event and clk='1' then
    mult1 <= Smult1;
    mult2 <= Smult2;
    Sum1 <= mult1(NI+NC−1 downto NC) + mult2(NI+NC−1 downto NC);
    Sum2 <= mult1(NI+NC−1 downto NC) − mult2(NI+NC−1 downto NC);
  end if;
end process sum_prod;

mux : process(add_sub,s, Sum1, Sum2)
begin
  if add_sub = '0' then
    Sfinal <= Sum1;
  else
    Sfinal <= Sum2;
  end if;
end process mux;

rom_cos_out<=cos_d(NC−1 downto NC−NOO);

end Behavioral;
A.6 Rom_wrapper_256

--rom_wrapper_256.vhd

-- NOTE: If ROM size changes, change index values of “A” (and ZERO assignment, if needed) in ROM instances below.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity rom_wrapper is
  generic (
    NF : integer;  -- number of Fe bits
    NC : integer  -- number of sine/cosine bits
  );
  port (
    clk : in std_logic;
    addr : in std_logic_vector(NF+1 downto 0);  -- 2 bits for qtr. wave select
    sin : out std_logic_vector(NC-1 downto 0);
    cos : out std_logic_vector(NC-1 downto 0)
  );
end rom_wrapper;

architecture Behavioral of rom_wrapper is

component ipsinRom
  -- CEN: ROM (“chip”) enable ('0' enables; '1' holds previous output values)
  -- EMA: Extra Margin Adjustment to adjust width of self–timing pulse
  -- (Useful after manufacture)
  port (  
    Q: out std_logic_vector(NC-1 downto 0);
    CLK: in std_logic;
    CEN: in std_logic;                      -- '0' enables ROM
    A: in std_logic_vector(8 downto 0);
    EMA: in std_logic_vector(2 downto 0)  -- Extra Margin Adjustment
  );
end component;

component ipcosRom
  -- CEN: ROM (“chip”) enable ('0' enables; '1' holds previous output values)
  -- EMA: Extra Margin Adjustment to adjust width of self–timing pulse.
  -- (Useful after manufacture)
  port (  
    Q: out std_logic_vector(NC-1 downto 0);
    CLK: in std_logic;
    CEN: in std_logic;                      -- '0' enables ROM
    A: in std_logic_vector(8 downto 0);
    EMA: in std_logic_vector(2 downto 0)  -- Extra Margin Adjustment
  );
end component;

signal rom_sin, rom_cos : std_logic_vector(NC-1 downto 0);
signal ZERO : std_logic;
signal cen : std_logic;
signal ema : std_logic_vector(2 downto 0);
signal max, addr_up, addr_down, addr_in : std_logic_vector(NF-1 downto 0);
signal part, part_d, part_dd : std_logic_vector(1 downto 0);
begin

cen <= '0';
ema <= "000";
ZERO <= '0';

iprom_sin : ipsinRom port map
  (Q => rom_sin, CLK => clk, CEN => cen, A(8) => ZERO, A(7 downto 0) => addr_in, EMA => ema);
iprom_cos : ipcosRom port map
  (Q => rom_cos, CLK => clk, CEN => cen, A(8) => ZERO, A(7 downto 0) => addr_in, EMA => ema);

max <= (others => '1'); -- Assign max explicitly for synthesis

addr_sync : process(clk) -- Synchronize all addr bits (still on falling edge)
begin
  if clk'event and clk='0' then
    part <= addr(NF+1 downto NF); -- select quarter of sin/cos wave
    part_dd <= part;
    addr_up <= addr(NF-1 downto 0); -- delay to match 2x delayed addr
    addr_down <= max - addr(NF-1 downto 0); -- reverse order of readout
  end if;
end process addr_sync;

address_setup : process (clk) -- Ready address for rising clk
begin
  if clk'event and clk='0' then
    if part = "00" then -- first quarter
      addr_in <= addr_up;
    elsif part = "01" then -- second quarter
      addr_in <= addr_down;
    elsif part = "10" then -- third quarter
      addr_in <= addr_up;
    else
      addr_in <= addr_down;
    end if;
  end if;
end process address_setup;

wave_qtr : process (part_dd, rom_sin, rom_cos) -- Flip quarter wave
begin
  if part_dd = "00" then -- first quarter
    sin <= rom_sin;
    cos <= rom_cos;
  elsif part_dd = "01" then -- second quarter
    sin <= not rom_sin + '1';
    cos <= rom_cos;
  elsif part_dd = "10" then -- third quarter
    sin <= not rom_sin + '1';
    cos <= not rom_cos + '1';
  else -- fourth quarter
    sin <= not rom_sin + '1';
    cos <= rom_cos;
  end if;
end process wave_qtr;
end Behavioral;
library ieee, work;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use ieee.math_real.all;
use std.textio.all;

entity DRFM_TB is
  generic (
    NI : integer := INP_num; -- number of input bits
    NC : integer := COEFF_num; -- number of coeff. bits
    NT : integer := TAP_num; -- number of taps
    ROM : integer := ROM_num; -- number of words used in ROM
    NF : integer := NF_num; -- number of bits for Fc factor = log2(ROM)
    CP : time := CP_numns; -- clock period (s)
    CPR : real := CP_numns-9; -- clock period (real)
    f1 : real := Fin_num; -- input freq#1 (Hz)
    f2 : real := Fin_num; -- input freq#2 (Hz)
    Fc : integer := Fc_num; -- ctr. freq. factor ('1' for lowest ctr. freq
    )
    N_DATA : integer := 4096; -- number of test sets per input freq
    N_START : integer := 200; -- number of test sets to initialize ckt
    f1_INCR : real := Fin_num; -- amount to incr. f1 and f2 for a sweep (Hz)
    N_FREQ : integer := NFRQ_num; -- number of freq's for sweep
    CP_INCR : time := CPINC_numns; -- clk period increment for testing clk speed
    N_CLK : integer := 4; -- number of clock increments
    P1 : real := 0.0; -- power level #1 [dB]
    P2 : real := 0.0; -- power level #2 [dB]
    MUX : std_logic := '0'; -- select ADCJN ('0') or memory ('1')
    USB : std_logic := 'USB_num'; -- upper/lower sideband ('1' for USB)
  );

end;

architecture TB of DRFM_TB is

component DRFM_entity
  generic (
    NF : integer := NF; -- number of sine/cosine address bits
    NC : integer := NI; -- number of sine/cosine bits
    NCO : integer := NI; -- number of output cosine bits
    NI : integer := NI -- number of input bits
  );

port ( 
  PADC_JN : in std_logic_vector(NI-1 downto 0); 
  -- digital input
  PIE : in std_logic; 
  -- input enable
  PCLK_JN : in std_logic;

end;
-- global clock
PMUX_S : in std_logic;
-- select ADCIN or "drfm" memory:
PRST_IN : in std_logic;
-- global reset
PADD_SUBS : in std_logic;
-- select upper or lower side band at output
PFc_IN : in std_logic_vector(NF downto 0);
-- ctrl freq scaling value
PMUX_OUT : out std_logic_vector(NI downto 0);
-- test point
PHIL_OUT : out std_logic_vector(NI downto 0);
-- test point
PHILQ_OUT : out std_logic_vector(NI downto 0);
-- test point
PROMCOS_OUT : out std_logic_vector(NCO downto 0);
-- final output
PDFRM_OUT : out std_logic_vector(NI downto 0);
end component;

begin

file OUT_FILE1: text is out "/drfm_ADC_IN.txt";
file OUT_FILE2: text is out "/drfm_MUX_OUT.txt";
-- file OUT_FILE3: text is out "/drfm_HIL1OUT.txt";
file OUT_FILE4: text is out "/drfm_HILQ_OUT.txt";
file OUT_FILE5: text is out "/drfm_COS_OUT.txt";
file OUT_FILE6: text is out "/drfm_DRFM_OUT.txt";
file OUT_FILE7: text is out "/drfm_param.txt";
file OUT_FILE8: text is out "/drfm_param_MATLAB.txt";

type TYPE_INPUT_ARRAY is array (0 to N_FREQ*(N_START+N_DATA)) of std_logic_vector(NI downto 0);
type TYPE_REAL_ARRAY is array (0 to N_FREQ*(N_START+N_DATA)) of real;
signal NDATA_TOT : integer := NrStart + NrData; -- total input data
signal INDEX : integer := 0; -- index for ADCIN
signal INPUT_V : TYPE_INPUT_ARRAY; -- signal for input arry
signal Fs_IN : std_logic_vector(NF downto 0);
signal IE, CLK_IN, MUX_S, RST_IN, ADD_SUBS : std_logic;
signal ADCIN, MUXOUT,
--HIL1_OUT, HILQ_OUT,
ROM_COSOUT, DRFM_OUT : std_logic_vector(NI downto 0);

-- INSTANTIATE UUT & ASSIGN SIGNALS
---drfm_uut DRFM_entity
--generic map (NF=>NF, NC=>NI, NCO=>NI, NI=>NI)
port map(
   PADC_IN => ADC_IN,
   PIE  => IE,
   PCLK_IN => CLK_IN,
   PMUX_S => MUX_S,
   PRST_IN => RST_IN,
   PADD_SUB_S => ADD_SUB_S,
   PFc_IN  => Fc_IN,
   PMUX_OUT => MUX_OUT,
   PHIL_OUT => HIL_OUT,
   PHIL_Q_OUT => HIL_Q_OUT,
   PROM_CO_OUT => ROM_CO_OUT,
   PDRFM_OUT => DRFM_OUT
);

MUX_S <= MUX;  -- select ADC_IN ('0') or memory ('1')
ADD_SUB_S <= USB;  -- select lower sideband ('0', i.e. shift down)
Fc_IN <= conv_std_logic_vector(Fc,NF);  -- set Fc value ('1' for highest cnt. freq.)

--xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-- GENERATE (& test) CLOCK
--vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

generate_clock: process  
    variable x : time := 0 ns;  -- x decreases period of clk to test limits
    begin
        for i in 1 to N_FREQ*N_DATA_TOT/N_CLK loop
            wait for (CP-x)/2;
            CLK_IN <= '0';
            wait for (CP-x)/2;
            CLK_IN <= '1';
        end loop;
        x := x + CP_INCR;
        assert(false) report "Clock period decreased by CP_INCR ns" severity note;
        assert(DRFM_OUT/="XXXXXXXXXXXXXXXX")  -- for 16-bit
            report "Clock speed failed" severity failure;
    end process;

--xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-- RESET CIRCUIT
--vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
reset: process (CLK_IN)
    variable ii : integer := 0;
    begin
        if (ii < 5) then
            RST_IN <= '1';
            ii := ii+1;
        else
            RST_IN <= '0';
        end if;
    end process;

--xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-- INPUT ENABLE (or test IE)
--vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
IE <= '1';

test_ie: process (CLK_IN)
    variable ii : integer := 0;
    begin
        if (ii > 350) and (ii < 370) then
apply_test_vector: process(CLK_IN)
begin
  if CLK_IN='event and CLK_IN='1' then
    ADC_IN <= INPUT_V(INDEX);
    if INDEX = N_FREQ*N_DATA_TOT then
      INDEX <= 0;
    else
      INDEX <= INDEX + 1;
    end if;
  end if;
end process;

generate_input_output: process
begin
  variable W1,W2,W_INCR: real; -- freqs. in radians (omega)
  variable A1,A2,Amax: real := 1.0; -- amplitudes of input signals
  variable P_dynym: real; -- relative power difference
  variable S_REAL: real; -- real-valued signal
  variable VAL_COMP,VAL_DIV: real; -- values for binary conversion
  variable INPUT_R: TYPE_REAL_ARRAY; -- real-valued input array
  variable OUT_LINE: line;
  variable J,K,L: integer;
  variable S_VECTOR: TYPE_INPUT_ARRAY; -- binary signal vector

  -- Input-signal parameters:

  P_dynym := P1 - P2;
  A2 := Amax / (1.0 + exp(log(10.0)*(P_dynym)/10.0));
  A1 := Amax - A2;
  W1 := 2.0 * 3.1415927 * f1; -- convert Hz to radians
  W2 := 2.0 * 3.1415927 * f2; -- convert Hz to radians
  W_INCR := 2.0 * 3.1415927 * f_INCR; -- freq. increment for sweep

  -- Write signal parameters to OUT_FILE7

  write(OUT_LINE,string('"DRFM_entity"'));
  writeln(OUT_FILE7, OUT_LINE);
  write(OUT_LINE,string('""'));
  writeln(OUT_FILE7, OUT_LINE);
  write(OUT_LINE,string('"Clk:"'));
  write(OUT_LINE,1.0/CPR);
  write(OUT_LINE,string('" Hz"'));
  writeln(OUT_FILE7, OUT_LINE);
  write(OUT_LINE,string('"f1:"'));
  write(OUT_LINE,f1);
write(OUTLINE, string '( "Hz f2:" ));
write(OUTLINE, f2);
write(OUTLINE, string '( "Hz") );
write(OUTFILE7, OUTLINE);

write(OUTLINE, string '( "Hz Fc: " ));
write(OUTLINE, real(Fc)/(CPR*4.0*real(ROM)));
write(OUTLINE, string '( "Hz") );
write(OUTFILE7, OUTLINE);

write(OUTLINE, string '( "fINCR: " ));
write(OUTLINE, fincr);
write(OUTLINE, string '( "Hz") );
write(OUTFILE7, OUTLINE);

write(OUTLINE, string '( "NDATA: " ));
write(OUTLINE, NDATA);
write(OUTFILE7, OUTLINE);

-- Write MATLAB(r) parameters to OUT_FILE8
write(OUTLINE, NI);
write(OUTFILE8, OUTLINE);
write(OUTLINE, ROM);
write(OUTFILE8, OUTLINE);
write(OUTLINE, CPR);
write(OUTFILE8, OUTLINE);
write(OUTLINE, f1);
write(OUTFILE8, OUTLINE);
write(OUTLINE, f2);
write(OUTFILE8, OUTLINE);
write(OUTLINE, Fc);
write(OUTFILE8, OUTLINE);
write(OUTLINE, f_incr);
write(OUTFILE8, OUTLINE);
write(OUTLINE, NDATA);
write(OUTFILE8, OUTLINE);
write(OUTLINE, N_freq);
write(OUTFILE8, OUTLINE);
write(OUTLINE, A1);
write(OUTFILE8, OUTLINE);
write(OUTLINE, A2);
write(OUTFILE8, OUTLINE);
write(OUTLINE, conv_integer(USB));
write(OUTFILE8, OUTLINE);
write(OUTLINE, NC);
write(OUTFILE8, OUTLINE);
write(OUTLINE, NT);
write(OUTFILE8, OUTLINE);

-- Generate input values (before simulation):
K:=0;
L:=0;
IN_FREQ_LOOP: while L < N_FREQ loop
  J:=0;
  INPUT_VALUES_LOOP: while J < N_DATA loop
    \[ S_{REAL} := A_1 \sin (W_1 \times \text{REAL}(K) \times \text{CPR}) + A_2 \sin (W_2 \times \text{REAL}(K) \times \text{CPR}) \] ; -- two tones

124
-- Convert signal to n-bit binary (2's Complement):

VAL_DIV := 0.0;
VAL_COMP := 0.0;

if S_REAL < VAL_COMP then  — negative signal value
    S_VECTOR(K)(NI-1) := '1';
    VAL_DIV := -0.5;
    VAL_COMP := -0.5;
else  — positive signal value
    S_VECTOR(K)(NI-1) := '0';
    VAL_DIV := 0.5;
    VAL_COMP := 0.5;
end if;

for I in 1 to NI-1 loop
    if S_REAL < VAL_COMP then
        S_VECTOR(K)(NI-1-I) := '0';
        VAL_DIV := VAL_DIV / 2.0;
        VAL_COMP := VAL_COMP - abs(VAL_DIV);
    else  — S_REAL >= VAL_COMP
        S_VECTOR(K)(NI-1-I) := '1';
        VAL_DIV := VAL_DIV / 2.0;
        VAL_COMP := VAL_COMP + abs(VAL_DIV);
    end if;
end loop;

-- Assign value to input:

INPUT_R(K) := S_REAL;
INPUT_V(K) <= S_VECTOR(K);
J := J + 1;
K := K + 1;
end loop INPUT_VALUES_LOOP;

W1 := W1 + W_INCR;  — increment freq.
W2 := W2 + W_INCR;  — increment freq.
L := L + 1;
end loop IN_FREQ_LOOP;

-- Write simulation results to files (while running):

L := 0; FREQ_LOOP: while L < N_FREQ loop
    K := 0; OUT_FILES_LOOP: while K < N_DATA_TOT loop
        if K > N_START-1 then
            -- OUT_FILE1:
            write(OUT_LINE, real(conv_integer(ADC_IN)));
            writeln(OUT_FILE1, OUT_LINE);

            -- OUT_FILE2:
            --write(OUT_LINE, real(conv_integer(MUX_OUT)));
            --writeln(OUT_FILE2, OUT_LINE);

            -- OUT_FILE3:
            --write(OUT_LINE, real(conv_integer(HIL_OUT)));
            --writeln(OUT_FILE3, OUT_LINE);

            -- OUT_FILE4:

        end if;
    end loop OUT_FILES_LOOP;
end loop FREQ_LOOP;
write(OUT_LINE, real(conv_integer(HIL_Q_OUT)))
write_line(OUT_FILE4, OUT_LINE);

--- OUT_FILE5:
write(OUT_LINE, real(conv_integer(ROM_COS_OUT)))
write_line(OUT_FILE5, OUT_LINE);

--- OUT_FILE6
write(OUT_LINE, real(conv_integer(DRFM_OUT)))
write_line(OUT_FILE6, OUT_LINE);
end if;

wait until CLK_IN'event and CLK_IN = '1';
wait until CLK_IN'event and CLK_IN = '0';
K := K + 1;
end loop OUT_FILES_LOOP;
L := L + 1;
end loop FREQ_LOOP;

assert (false) report "Simulation done!" severity FAILURE;
end process;
end TB;

configuration CFG_TB of DRFM_TB is
for TB
end for;
end;
Appendix C. RTL Compiler & First Encounter® Scripts

[Names of technology files have been removed]

C.1 RTL Compiler Script (rc.cmd)

# Cadence RTL Compiler command script for design synthesis.
# Original Author: Dr. J. Marty Emmert.
# Modified by: Thomas Hopkins, Wright State University 2010,
# for DRFM DSSM thesis project.

####

set INP 16
set CLA 32
set COEFF 14
set TAP 101
set ROM 256

#set CNR "" #set CNR ss_1p08v_125c
set CNR ff_1p32v_125c

#set ROMLIB tt_1p20v_25c_syn.lib
#set ROMLIB ss_1p08v_125c_syn.lib
set ROMLIB ff_1p32v_125c_syn.lib

#set CKTLIB "[90_nm_std_cells]_tt_1p2v_25c.lib [90_nm_io_cells]
_1p2v_2p5v_25c.lib"
#set CKTLIB "[90_nm_std_cells]_ss_1p08v_125c.lib [90_nm_io_cells]
_1p08v_2p3v_125c.lib"
set CKTLIB "[90_nm_std_cells]_ff_1p32v_125c.lib [90_nm_io_cells]
_1p32v_2p7v_125c.lib"

set CKTLEF "[90_nm_rc_values].lef [90_nm_std_cell_macros].lef [90_nm_io_cell_macros].lef"

####

set U .
set DENTITY DRFMUSINPSUSCOEFF$USTAP$USROM
set DESIGN $DENTITY$CNR
echo echo "$DESIGN"
echo

set ONE 1
set WIM $INPSUSINPSUSONE
set ADD2 $INPSUSINP
set NC NC
set HIL NT$TAP$USNC$COEFF

# set Design vhdl files
set DFILES "./.gen/cosRom$INPSUS$ROM.vhd "./.gen/sinRom$INPSUS$ROM.vhd "./.gen/syn.vhd "./.gen/add2$ADD2$ADD2.vhd "./.gen/csa$ADD2.vhd "./.gen/cla$CLA.vhd "./.gen/wns$WTM.vhd "./.gen/hilbert$HIL.vhd "./.src/rom_wrapper$ROM.vhd "./.src/mixer.vhd "./.src/$DENTITY.vhd"
### allow lib vs lef comparison
set_attribute lib_lef_consistency_check enable true

### set dir location for library location
set_attribute lib_search_path ../../../techn_files_dir/

### set library file name in lib_search_path directory
set_attribute library "$SCKT_LIB ../rom/$INP$USROM/ipsinRom.$ROM_LIB ..../rom/
rom$INP$USROM/ipcosRom.$ROM_LIB"

### set lef file names from lib_search_path directory
set_attribute lef_library "$SCKT_LEF ../rom/$INP$USROM/ipsinRom.lef ../rom/
rom$INP$USROM/ipcosRom.lef"

### read in design files
read_hdl vhdl $DFILES

### Do preliminary timing analysis to make sure constraints are realistic
### before synthesis
set_attribute map_timing true

### elaborate designs
elaborate

### set clock period in picoseconds [ps] (10000=>100 MHz)
set_clock [define_clock -design $ENTITY -period 31250 PCLK_IN ]
set_clock [define_clock -design $ENTITY -period 10000 PCLK_IN ]

### clock uncertainty setup rise ... fall (in ps)
set_attribute clock_setup_uncertainty 1 $clock

### clock uncertainty rise ... fall (in ps)
set_attribute clock_hold_uncertainty 1 $clock

### set internal/external delays in picoseconds
external_delay -input 50 -clock $clock -edge_rise {PADC_IN[*] [Input Enable Pad] 
PCLK_IN PMUX_INPUT PRST_IN}
external_delay -input 50 -clock $clock -edge_rise {PADD_SUB$ PFc_IN[*]}
external_delay -output 50 -clock $clock -edge_rise {PMUX_OUTPUT[*] PHIL_Q_OUTPUT[*] 
PHIL_Q_OUTPUT[*] PDRFM_OUTPUT[*]}

### set output pin capacitance loads for buffering [fF]
set_attribute external_pin_cap 10000 CLKTXOUT
set_attribute external_pin_cap 10000 SIG[*]
set_attribute external_pin_cap 10000 CLKTXOUT

### set output pin resistance (KOhms)
set_attribute external_resistance 1.0 signal

### set output wire capacitance loads (fF)
set_attribute external_wire_cap 10 signal

### set output wire resistance (KOhms)
set_attribute external_wire_res 10 signal

### Fixes the minimum design rule costs based on calculations in the
### library, (min capacitance, transmission, fanout, etc...)
set_attribute fix_min_drcs true

#### Synthesis Optimizations####

### Automatic Retiming after synthesis
set_attribute retime true $ENTITY
# Achieves area gain by adder optimization
#set_attribute dp_area_mode true

# Performs architecture downsizing after mapping without
degrading timing
#set_attribute dp_postmap_downsize true

# Performs architecture upsizing after mapping to improve timing
#set_attribute dp_postmap_upsize true

# Single-bit cells are mapped to an appropriate multi-bit cell
# from the technology library (eg. 4 single FF mapped into 1 quad FF)
#set_attribute use_multibit_cells true

##### END Synthesis Optimizations#####

# synthesize the circuit
#synthesize -to_mapped -effort medium -csa_effort medium
synthesize $DENTITY -to_mapped -effort high -csa_effort high

# Physically aware flow with quality of silicon (QoS) prediction
#predict_qos

# Write encounter configuration files
write_encounter $DENTITY -basename ./for_enc/$DESIGN/$DESIGN

# Check for and/or create report directory
if { ![file exists "$DESIGN"] } {
    exec mkdir "$DESIGN"
    puts "Creating directory $DESIGN"
}

# write output files for later timing analysis
#write $DENTITY -mapped > $DESIGN/$DESIGN.v
write_set_load $DENTITY > $DESIGN/$DESIGN.loads
write_sdc $DENTITY > $DESIGN/$DESIGN.sdc
write_script $DENTITY > $DESIGN/$DESIGN.tcl

# write Verilog and sdc files for First Encounter
write $DENTITY -mapped > ../enc/des_files/$DESIGN.v
write_sdc $DENTITY > ../enc/des_files/$DESIGN.sdc

# Write reports
cd designs
cd $DENTITY
#report clocks -ideal -generated > ./DESIGN/$DESIGN.clocks
report clocks > ./DESIGN/$DESIGN.clocks
report timing > ./DESIGN/$DESIGN.timing
report timing -lint > ./DESIGN/$DESIGN.timinglint
report gates > ./DESIGN/$DESIGN.gates
report area > ./DESIGN/$DESIGN.area
report power > ./DESIGN/$DESIGN.power
report power -clock_tree $clock > ./DESIGN/$DESIGN.power_clk
report summary > ./DESIGN/$DESIGN.summary
check_design $DENTITY > ./DESIGN/$DESIGN.check
quit
C.2 First Encounter® Command Script (build_all.cmd)

# Cadence First Encounter(r) layout and route command file
# Original Author: Dr. J. Marty Emmert.
# Modified by: Thomas Hopkins, Wright State University 2010,
# for DRFM DSSM thesis project.

```
set D_ENTITY DRFM 16 14 101 256
set CNR sf1p32v 125c
set DESIGN $D_ENTITY$CNR

## floorplan.cmd (set design name above & choose fp size below) ####
loadConfig des_files/$DESIGN.conf 0
commitConfig
clearGlobalNets
globalNetConnect VDD -type ppgpin -pin VDD -inst {{VDD Pad}*}
globalNetConnect VSS -type ppgpin -pin VSS -inst {{VSS Pad}*}
globalNetConnect VDD -type ppgpin -pin VDD -inst {}
globalNetConnect VSS -type ppgpin -pin VSS -inst {}
globalNetConnect VDD -type net -net VDD
globalNetConnect VSS -type net -net VSS

## Choose floorplan size (already calculated here for I/O used in design)
## (Sets chip and core sizes)
## 8-bit fp:
#floorPlan -site cms9flpsite -b 0.0 0.0 1443.0 1443.0 174.0 174.0 1269.0 1269.0 269.0 269.0 1174.0 1174.0
## 16-bit 128/256 ROM fp:
#floorPlan -site cms9flpsite -b 0.0 0.0 2246.0 2246.0 174.0 174.0 2072.0 2072.0 269.0 269.0 1977.0 1977.0
## 16-bit 512 ROM fp:
#floorPlan -site cms9flpsite -b 0.0 0.0 2319.0 2319.0 174.0 174.0 2145.0 2145.0 269.0 269.0 2050.0 2050.0
## 24-bit fp:
#floorPlan -site cms9flpsite -b 0.0 0.0 3195.0 3195.0 174.0 174.0 3021.0 3021.0 269.0 269.0 2926.0 2926.0

## Add Halos around ROM macros to keep std cells from being placed too close
addHaloToBlock 25 25 25 25 mix/roms/iprom_sin
addHaloToBlock 25 25 25 25 mix/roms/iprom_cos

setRoutingStyle -top -style m
uiSetTool select

## Fill in space betw I/O Pads
addIoFiller -cell PFILL1 -prefix PFILL
saveDesign $DESIGN/$DESIGN.fp.enc
fit

## place.cmd (places design) ##################################################
#restoreDesign $DESIGN/$DESIGN.fp.enc.dat $D_ENTITY

## End-cap cells & well taps left out since not being fabbed
#addEndCap -preCap ENDCAPMH -postCap ENDCAPMNH -prefix ENDCAP
#addWellTap -cell FILLTIEVH -maxGap 39.2 -prefix FILLTIE

#placeDesign -prePlaceOpt
placeDesign -noPrePlaceOpt

## Fix Macro placement so they don’t get moved again
setBlockPlacementStatus -allHardMacros -status fixed
```

130
## Tie '0's and '1's in design
addTieHiLo -cell {TIELOMVH TIEHIMVH} -prefix LTIE
saveDesign $DESIGN/$DESIGN.p_enc

## power.cmd (add power rings, stripes, & structure) #################################
#restoreDesign $DESIGN/$DESIGN.p_enc.dat $ENTITY
setAddStripeOption -merge_with_all_layers 1
addStripe -block_ring_bottom_layer_limit MI -block_ring_top_layer_limit OL - max_same_layer_jog_length 4 -merge_stripes_value 10 - padcore_ring_bottom_layer_limit M5 -set_top_set_distance 146.0 - stacked_via_top layer LD -padcore_ring_top_layer_limit OL -spacing 64.04 - xleft_offset 89.48 -xright_offset 50.00 -layer MI2B -width 8.96 -nets {VSS VDD} -stacked_via_bottom_layer MI

## Add power structures
sroute -noBlockPins -jogControl { preferWithChanges preferDifferentLayer } -layerChangeTopLayer 7
saveDesign $DESIGN/$DESIGN.pwr.enc

## clktree.cmd (synthesize clock tree) ##############################################
#restoreDesign $DESIGN/$DESIGN_pwr_enc.dat $ENTITY

clockDesign -specFile des_files/design.ctstch -outDir ./$DESIGN/clock_report -fixedInstBeforeCTS
createSaveDir $DESIGN/$DESIGN.cts
saveNlist $DESIGN/$DESIGN.cts.$DESIGN.cts.v
savePlace $DESIGN/$DESIGN.cts/$DESIGN.cts.place
saveDesign $DESIGN/$DESIGN.clk.enc

## route.cmd (route global and detail signal networks) ###############################
## (routinig is timing and SI driven)

#restoreDesign $DESIGN/$DESIGN.clk.enc.dat $ENTITY
routeDesign -globalDetail
saveDesign $DESIGN/$DESIGN.par.enc

## fill.cmd (add filler cells) ####################################################
#restoreDesign $DESIGN/$DESIGN_par_enc.dat $ENTITY

addFiller -cell FILL16MVH FILL1MVH FILL2MVH FILL8MVH FILL4MVH FILL32MVH FILL64MVH -prefix FILL -markFixed
saveDesign $DESIGN/$DESIGN.fill.enc

## save.cmd (write design to def file & save final version) ##########################
#restoreDesign $DESIGN/$DESIGN.fill.enc.dat $ENTITY

global dbgLefDefOutVersion
set dbgLefDefOutVersion 5.5
defOut -floorplan -netlist -routing $DESIGN/$DESIGN.def
set dbgLefDefOutVersion NULL
saveDesign $DESIGN/$DESIGN.$ENTITY.enc

## verify.cmd (check & verify design; write netlist) #################################
checkDesign -outfile $DESIGN/$DESIGN.checkDesign.rpt -io -netlist -physicalLibrary -powerGround -tieHilo -timingLibrary -floorplan -place -noHtml

#verifyGeometry -report $DESIGN/$DESIGN.geom.rpt -area { 176.0 176.0 2946.0 2946.0 }
verifyGeometry -report $DESIGN/$DESIGN.geom.rpt

#verifyMetalDensity -layers { OL LD } -reportfile $DESIGN/$DESIGN.density.rpt
verifyConnectivity -report $DESIGN/$DESIGN.conn.rpt -type all -error 1000 -warning 50
verifyProcessAntenna -reportfile $DESIGN/$DESIGN.antenna.rpt -leffile $DESIGN/$DESIGN.antenna.lef -error 10000

violationBrowserReport -report $DESIGN/$DESIGN.viol.rpt

timeDesign -postRoute -outdir $DESIGN/$DESIGN.timingReports
report_timing > $DESIGN/$DESIGN.timingReports/report_timing.rpt
check_timing >> $DESIGN/$DESIGN.timingReports/check_timing.rpt
summaryReport -outdir ./$DESIGN -outfile ./$DESIGN/$DESIGN.summary.rpt

genPinText $DESIGN/LVS.txt -cells {*}
saveNetlist $DESIGN/LVS.v -includePowerGround -phys
exit
C.3  First Encounter® Configuration File ([design].conf)

```
set INP 16
set COEFF 14
set TAP 101
set ROM 256

###set CNR ""
##set CNR .ss_tt1p08v_125c
##set CNR .ss1p08v_125c
set CNR .ff1p32v_125c

###set STDLIB [90 nm_std_cells] ss_tt1p02v_25c.lib
##set STDLIB [90 nm_std_cells] ss1p08v_125c.lib
set STDLIB [90 nm_std_cells] ff1p32v_125c.lib

###set IOLIB [90 nm_io_cells] ss_tt1p02v_25v_2p5v_25c.lib
##set IOLIB [90 nm_io_cells] ss1p08v_2p3v_2p0v_25c.lib
set IOLIB [90 nm_io_cells] ff1p32v_2p0v_2p5v_25c.lib

###set ROMLIB tt1p20v_25c_syn.lib
##set ROMLIB ss1p08v_125c_syn.lib
set ROMLIB ff1p32v_25c_syn.lib

set CKTLEF "./[techn_files_dir]/[90 nm_std_cells].lef ../[techn_files_dir]/[90 nm_std_cells Macros].lef ../[techn_files_dir]/[90 nm_io_cells Macros].lef"

set DESIGN SDENTITYSCNR
set rda_input global
set rda_input.cwd .
set rda_input.import_mode 
set rda_input.ui_netlist "des_files/$DESIGN.v"
set rda_input.ui_netlisttype {Verilog}
set rda_input.ui_rtllist ""
set rda_input.ui_lmilist ""
set rda_input.ui_imspef ""
set rda_input.ui_settop {0}
set rda_input.ui_topcell {}
set rda_input.ui_celllib ""
set rda_input.ui_iolib ""
set rda_input.ui_areaiolib ""
set rda_input.ui_blklib ""
set rda_input.ui_bboxlib ""
set rda_input.ui_sds_file ""
set rda_input.ui_oxid {
set rda_input.ui_view_definition_file ""
set rda_input.ui_timeLib.max ""
set rda_input.ui_timelib.min ""
set rda_input.ui_timelib .../[techn_files_dir]/$STDLIB ../[techn_files_dir]/$IOLIB ../[rom/rom$INPS$US$ROM/ipsinRom$ROM$LIB ../[rom/rom$INPS$US$ROM/pcsRom$ROM$LIB"
set rda_input.ui_smodDef ""
set rda_input.ui_smodData ""
set rda_input.ui_dpath ""
set rda_input.ui_tech_file ""
set rda_input.ui_tech_file "des_files/DRFM$INPS$US$ROM.io"
set rda_input.ui_tech_file "des_files/$DESIGN.sdc"
set rda_input.ui_latency_file ""
set rda_input.ui_scheduling_file ""
set rda_input.ui_buf_footprint {{Buffer Cell x18}}
```

133
set rda_input (ui_delay_footprint) {}
set rda_input (ui_inv_footprint) {{Inverter Cell x8}}
set rda_input (ui_leffile) "$SCKTLEF .../rom/rom$SROM/ipsinRom.vclef .../rom/
set rda_input (ui_VISIBLE) "INP$U/B0/ROM/ipcosRom.vclef"
set rda_input (ui_cts_cell_footprint) {}
set rda_input (ui_cts_cell_list) {}
set rda_input (ui_aspect_ratio) {1.0}
set rda_input (ui_aspect_util) {0.7}
set rda_input (ui_core_height) {}
set rda_input (ui_core_width) {}
set rda_input (ui_core_to_left) {}
set rda_input (ui_core_to_right) {}
set rda_input (ui_core_to_top) {}
set rda_input (ui_core_to_bottom) {}
set rda_input (ui_max_to_height) {0}
set rda_input (ui_row_height) {}
set rda_input (ui_isHorTrackHalfPitch) {0}
set rda_input (ui_isVerTrackHalfPitch) {1}
set rda_input (ui_ipOri) (R0)
set rda_input (ui_isOriginCenter) {0}
set rda_input (ui_isVerticalRow) {0}
set rda_input (ui_exc_net) ""
set rda_input (ui_delay_limit) (1000)
set rda_input (ui_net_delay) (1000.0 ps)
set rda_input (ui_nets_load) {0.5 pF}
set rda_input (ui_nets_trant_limit) {0.0 ps}
set rda_input (ui_captbli_file) ""
set rda_input (ui_defcap_scale) {1.0}
set rda_input (ui_defcap_scale) {1.0}
set rda_input (ui_defcap_scale) {1.0}
set rda_input (ui_res_scale) {1.0}
set rda_input (ui_shr_scale) {1.0}
set rda_input (ui_refc_thres) {0.03}
set rda_input (ui_totc_thres) {5.0}
set rda_input (ui_time_unit) {none}
set rda_input (ui_cap_unit) {}
set rda_input (ui_oa_reflib) ""
set rda_input (ui_oa_abstractname) {}
set rda_input (ui_oa_layoutname) {}
set rda_input (ui_sigstormlib) ""
set rda_input (ui_cdb_file , min) ""
set rda_input (ui_cdb_file , max) ""
set rda_input (ui_cdb_file) ""
set rda_input (ui_echo_file , min) ""
set rda_input (ui_echo_file , max) ""
set rda_input (ui_echo_file) ""
set rda_input (ui_xtwf_file) ""
set rda_input (ui_qxtech_file) ""
set rda_input (ui_qxlayermap_file) ""
set rda_input (ui_qxlib_file) ""
set rda_input (ui_qxconf_file) ""
set rda_input (ui_pwrnet) {VDD}
set rda_input (ui_gndnet) {VSS}
set rda_input (flip_first) {1}
set rda_input (double_back) {1}
set rda_input (assign_buffer) {1}
set rda_input (ui_pg_connections) ""
set rda_input (ui_gen FOOTPRINT) {0}
# Chip South Side
Offset: 0.000
Spacing: 0.000
Pad: GCNRS S [Corner Pad Name]
Pad: GHIL.Q.OUT.15 S [Output Pad Name]
Pad: GHIL.Q.OUT.14 S [Output Pad Name]
Pad: GHIL.Q.OUT.13 S [Output Pad Name]
Pad: GHIL.Q.OUT.12 S [Output Pad Name]
Pad: GHIL.Q.OUT.11 S [Output Pad Name]
Pad: GHIL.Q.OUT.10 S [Output Pad Name]
Pad: GHIL.Q.OUT.9 S [Output Pad Name]
Pad: GHIL.Q.OUT.8 S [Output Pad Name]
Pad: GHIL.Q.OUT.7 S [Output Pad Name]
Pad: GHIL.Q.OUT.6 S [Output Pad Name]
Pad: GHIL.Q.OUT.5 S [Output Pad Name]
Pad: GHIL.Q.OUT.4 S [Output Pad Name]
Pad: GHIL.Q.OUT.3 S [Output Pad Name]
Pad: GHIL(Q.OUT.2 S [Output Pad Name]
Pad: GHIL.Q.OUT.1 S [Output Pad Name]
Pad: GHIL.Q.OUT.0 S [Output Pad Name]
Pad: GROM.COS.OUT.0 S [Output Pad Name]
Pad: GROM.COS.OUT.1 S [Output Pad Name]
Pad: GROM.COS.OUT.2 S [Output Pad Name]
Pad: GROM.COS.OUT.3 S [Output Pad Name]
Pad: GROM.COS.OUT.4 S [Output Pad Name]
Pad: GROM.COS.OUT.5 S [Output Pad Name]
Pad: GROM.COS.OUT.6 S [Output Pad Name]
Pad: GROM.COS.OUT.7 S [Output Pad Name]
Pad: GDVSSS S [Pad−VSS Pad]
Pad: GDVDES S [Pad−VDD Pad]

# Chip West Side
Offset: 174.000
Spacing: 0.000
Pad: GADC.IN.15 W [Input Pad Name]
Pad: GADC.IN.14 W [Input Pad Name]
Pad: GADC.IN.13 W [Input Pad Name]
Pad: GADC.IN.12 W [Input Pad Name]
Pad: GADC.IN.11 W [Input Pad Name]
Pad: GADC.IN.10 W [Input Pad Name]
Pad: GADC.IN.9 W [Input Pad Name]
Pad: GADC.IN.8 W [Input Pad Name]
Pad: GADC.IN.7 W [Input Pad Name]
Pad: GADC.IN.6 W [Input Pad Name]
Pad: GVSSW W [VSS Pad]
Pad: GVDDW W [VDD Pad]
Pad: GADC.IN.5 W [Input Pad Name]
Pad: GADC.IN.4 W [Input Pad Name]
Pad: GADC.IN.3 W [Input Pad Name]
Pad: GADC.IN.2 W [Input Pad Name]
Pad: GADC.IN.1 W [Input Pad Name]
Pad: GADC.IN.0 W [Input Pad Name]
Pad: GCLK W [Input Pad Name]
Pad: GRST.IN W [Input Pad Name]
Pad: GIE W [Input Enable Pad]
Pad: GMUX.IN S W [Input Pad Name]
Pad: GADD.SUB.L W [Input Pad Name]
Offset: 1926.000
Pad: GDVSSW W [Pad−VSS Pad]
Pad: GDVDDW W [Pad−VDD Pad]
Pad: GCNRW W [Corner Pad Name]
# Chip East Side

Offset: 0.000  
Spacing: 0.000

<table>
<thead>
<tr>
<th>Pad</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCNRE</td>
<td>Corner Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_15</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_14</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_13</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_12</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_11</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_10</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_9</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_8</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_7</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_6</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GDRFM_OUT_5</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GVSSE</td>
<td>VSS Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GVDD</td>
<td>VDD Pad</td>
<td>Name</td>
</tr>
</tbody>
</table>

# Chip North Side

Offset: 174.000  
Spacing: 0.000

<table>
<thead>
<tr>
<th>Pad</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFc_IN_7</td>
<td>Input Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GFc_IN_6</td>
<td>Input Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GFc_IN_5</td>
<td>Input Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GFc_IN_4</td>
<td>Input Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GFc_IN_3</td>
<td>Input Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GFc_IN_2</td>
<td>Input Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GFc_IN_1</td>
<td>Input Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GFc_IN_0</td>
<td>Input Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_15</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_14</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_13</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_12</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_11</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_10</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_9</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_8</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_7</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_6</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_5</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_4</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_3</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_2</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GMUX_OUT_1</td>
<td>Output Pad</td>
<td>Name</td>
</tr>
<tr>
<td>GCRN</td>
<td>Corner Pad</td>
<td>Name</td>
</tr>
</tbody>
</table>

136
C.5 NCSIM® Simulation Script

# UNIX–shell script for simulating designs.
#!/bin/csh
source set_cadence_sim_env

# Set bit widths (CLA = 2x INP), no. of taps, ROM size,
# and characterization corner (ignored for behavioral simulation):
setenv INP 16       # number of input bits
setenv CLA 32       # carry lookahead adder label (2x INP)
setenv COEFF 14     # number of coefficient bits
setenv TAP 101      # number of filter taps
setenv ROM 256      # number of ROM words used
setenv NF 8         # number of bits for Fc factor (log2(ROM))
#setenv CNR ""      # characterization corner ("" for typical)
##setenv CNR _ss_l0p08_s12c # (for DRFM_16_14_101_256_r8...)
setenv CNR _ff_l0p32v_12c # (for DRFM_16_14_101_256_tt...)

setenv CP 65.104167  # clk period (ns)
setenv CP_INC 0     # CP increments for testing clk (ns)
setenv Fin 750.0e3  # initial input freq (Hz)
setenv Fc 50        # ctr freq factor (1 to ROM/2) => Fc/1-ROM*CP Hz
setenv NFREQ 150.0e3 # input freq increments for sweep test (Hz)
setenv N_FREQ 41    # number of freq's for sweep test
setenv USB 1        # sideband select ('1' is USB)

# Set simulation types NOT wanted to "#":
setenv BEH ""
setenv SYN ""
setenv PAR ""

####################################################

setenv ENTITY DRFM$INP""$COEFF""$TAP""$ROM
setenv DESIGN $ENTITY$CNRPIC
echo "$DESIGN"

$BEH setenv EXT ""
$SYN setenv EXT ""syn"
$PAR setenv EXT ""par"

# Make a temporary new test bench file (and update names and variables)
cp .. /src/1DRFM_template.vhd .. /src/$DESIGN$tb.vhd
perl -pi -e "s/DRFM/entity/$ENTITY/g" .. /src/$DESIGN$tb.vhd
perl -pi -e "s\d/dfm/\d/$DESIGN$EXT/dfm/g" .. /src/$DESIGN$tb.vhd
perl -pi -e "s/INP_num/$INP/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/COEFF_num/$COEFF/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/TAP_num/$TAP/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/RM_num/$RM/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/NF_num/$NF/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/CP_num/$CP/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/FM_num/$FM/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/Fc_num/$Fc/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/FN_num/$FN/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/FINC_num/$INC/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/NFREQ_num/$NF/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/CPINC_num/$CPINC/g" ... /src/$DESIGN$tb.vhd
perl -pi -e "s/USB_num/$USB/g" ... /src/$DESIGN$tb.vhd

# Make directory for results
mkdir $DESIGN$EXT

# Remove old files
rm * .log

137
rm -rf *.rcf
rm -rf INCA_libs

# Link to ROM's .rcf files
ln -s ../rom/$INP" "$ROM/ipcosRom_verilog.rcf
ln -s ../rom/$INP" "$ROM/ipsinRom_verilog.rcf

# I/O PAD MODELS
ncvhdl -64BIT -NOCOPYRIGHT -MESSAGES .. /

# BEHAVIORAL SIMULATION:
$BEH ncvhdl -64BIT -NOCOPYRIGHT -MESSAGES .. /
genadd2_$INP "$INP.vhd

gen/cla$INP "$INP.vhd

gen/ctn$INP "$INP.vhd

$BEH ncvhdl -64BIT -NOCOPYRIGHT -MESSAGES .. /gen/hilbert$INP "$INP.vhd

$BEH ncvhdl -64BIT -NOCOPYRIGHT -MESSAGES .. /src/rom_wrapper$INP "$INP.vhd

# SYNTHESIZED SIMULATION:
$SYN cat ./timescale.txt .. /

# POST-PAR SIMULATION:
$PAR cat ./timescale.txt .. /

# ROM MODELS:
ncvlog -64BIT -NOCOPYRIGHT -MESSAGES .. /rom/rom/$INP "$INP.v

# ELABORATION & SIMULATION:
ncvhdl -64BIT -NOCOPYRIGHT -MESSAGES .. /src/$DESIGN "tb.vhd
ncelab -64BIT -NOCOPYRIGHT -MESSAGES CFG_TB --access rwc
ncsim -64BIT -MESSAGES CFG_TB

rm .. /src/$DESIGN "tb.vhd
Appendix D. MATLAB® Scripts

D.1 Math Model and Frequency Sweep Plotting

```matlab
% drfm.m
% Calculates and plots DRFM signals and spur free dynamic range (SFDR)
% (both in dBc) in frequency domain for given parameters and data files.
% Data and parameter files are assumed to have been generated by the DRFM
% test bench simulation.
%
% Original created by Thomas Pemberton, Wright State University graduate
% student. Modification by Thomas Hopkins, Wright State University,
% spring/summer 2010.

close all;
clear all; clc;

% select folder and data source
design = 'DRFM_16_14_101_256_syn';
data_file = 'drfm_DRFM_OUT.txt';
use_MATLAB = 0; % '1' to use MATLAB to generate data
qntz = 1; % '1' to quantize MATLAB data
dbFloor = -140; % lowest plot level (dBc)

% load parameters
param_file = 'drfm_param_MATLAB.txt'; % generated by test bench
p = load(fullfile(design,param_file));

% assign MATLAB data parameters
if use_MATLAB
    NI = p(1); % number of input bits
    NC = p(13); % number of coeff. bits
    NT = p(14); % number of taps
    ROM = p(2); % number of words used in ROM
    Ts = p(3); % sampling period used (s)
    f1 = p(4); % initial input freq#1 (Hz)
    f2 = p(5); % initial input freq#2 (Hz)
    Fc = p(6); % center frequency factor (from 1 to ROM/2)
    f_incr = p(7); % input freq. increments (Hz)
    N = p(8); % number of data points for each input freq.
    N_freq = p(9); % number of input freqs. in sweep
    A1 = p(10); % amplitude of input#1
    A2 = p(11); % amplitude of input#2
    USB = p(12); % sideband select: '1' for USB, '0' for LSB
else
    NI = p(1); % number of input bits
    ROM = p(2); % number of words used in ROM
    Ts = p(3); % sampling period used (s)
    f1 = p(4); % initial input freq#1 (Hz)
    f2 = p(5); % initial input freq#2 (Hz)
    Fc = p(6); % center frequency factor (from 1 to ROM)
    f_incr = p(7); % input freq. increments (Hz)
    N = p(8); % number of data points for each input freq.
    N_freq = p(9); % number of input freqs. in sweep
    A1 = p(10); % amplitude of input#1
    A2 = p(11); % amplitude of input#2
    USB = p(12); % sideband select: '1' for USB, '0' for LSB
```

139
end

%%% setup internal parameters and variables
dla bel = strrep(d es i gn , ' ', ' \backslash r ' ); % convert design to label for plot title
if use_MATLAB
    titl = 'MATLAB(r) Model ';
else
    titl = ' ';
end

% Nfft = 2^nextpow2(N); % makes N the next largest factor of 2 (for fft)
Fc_hz = Fc/(Ts*4*ROM); % center/shift frequency (Hz)
fs = 1/Ts; % sampling/clock freq
fstep = Fs/fs; % freq plot step size
f = 0 : fstep : Fs/2; % output freq range (0 − Fs/2) to plot
Fin = f1 : f_incr : f1 + (Nfreq−1)*f_incr; % input freq sweep range to plot
if USB
    SB = 'USB';
else
    SB = 'LSB';
end

% text for xlabel:
%text_range_text = ['from ', num2str(f1, '%3.0f'), ' to ', num2str((f1+f_incr*(Nfreq−1)), '%3.0f'), ' Hz, every ', num2str(f_incr, '%3.0f'), ' Hz'];

% generate data with MATLAB
if use_MATLAB
    q = quantizer ('mode', 'fixed', 'roundmode', 'floor', 'format', [NI,NI−1]);
    qh = quantizer ('mode', 'fixed', 'roundmode', 'floor', 'format', [NC,NC−1]);
    % warning ('off', 'last');
    if qntz, h = quantize(qh,h); end
    addr = zeros(1,Nfreq*N);

    % Generate Hilbert filter coeffs.
    n = −(NT−1)/2:(NT−1)/2;
    h = 2*(sin(pi*n/2)).*2 ./ (pi*n);
    for i = 1:NT
        if rem(i,2) = 0, h(i)=0; end
    end
    if qntz, h = quantize(q,h); end
    adc_in = [];
    for i = 1:N_Nfreq
        % ADC
        W1 = (f1 + (i−1)*f_incr)*2*pi;
        W2 = (f2 + (i−1)*f_incr)*2*pi;
        adc = A1.*sin(W1*KsTs) + A2.*sin(W2*KsTs);
        if qntz, adc = quantize(q,adc); end
        adc_in = [adc_in adc(extra/2+1 : end − extra/2)]; % for plot

        % HILBERT FILTER (Discrete Hilbert Transform)
        Q = conv(h,adc);
        Q = Q(extra/2+1 : (NT−1)/2 : end − extra/2 − (NT−1)/2);
        I = adc(extra/2+1 : end − extra/2); % both Q & I now length N
        if qntz, Q = quantize(q,Q); end
        if qntz, I = quantize(q,I); end

        % ROM's
        J=1:N;
        addr = mod(Fc.*s(J−1), 4*ROM);
        cosine = cos(2*pi.*addr.(4*ROM));
        sine = sin(2*pi.*addr.(4*ROM));
        if qntz, cosine = quantize(q,cosine); end
        if qntz, sine = quantize(q,sine); end

end

140
```matlab
% MIXER
mult1 = I.*cosine;
mult2 = Q.*sine;
if qntz, mult1 = quantize(q,mult1); end
if qntz, mult2 = quantize(q,mult2); end
sum1 = mult1 + mult2; %LSB
sum2 = mult1 - mult2; %USB
if qntz, sum1 = quantize(q,sum1); end
if qntz, sum2 = quantize(q,sum2); end
if USB
data(1+(i-1)*N : i*N, 1) = sum2(length(I)-N+1:length(I))'; %upper SB
else
data(1+(i-1)*N : i*N, 1) = sum1(length(I)-N+1:length(I))'; %lower SB
end
end
adc_in = adc_in';
data=adc_in(1:N)';
data=cosine';
data=mult1';

% load data from simulation files
else
adc_in = load(fullfile('drfmADCIN.txt'));
I = load(fullfile('drfmHILIOUT.txt'));
Q = load(fullfile('drfmHILQOUT.txt'));
data = load(fullfile('datafile'));
if Nfreq == length(data)/N
'N_freq ~ length(data)/N'
break;
end
end

% convert data to freq domain and plot
SFDR = zeros(4,N_freq); % initialize spur free dynamic range (SFDR) data
% SFDR(1,:) = Input frequency
% SFDR(2,:) = SFDR (dBc)
% SFDR(3,:) = Spur frequency
% SFDR(4,:) = Spur freq. relative to input freq.
SFDR2 = zeros(4,N_freq); % initialize "SFDR" data for 2nd peak
FFOUT = zeros(1,N_freq); % initialize output peaks (magnitude)
FFIN = zeros(1,N_freq); % initialize input peaks (magnitude)
SFDRIN = zeros(1,N_freq); % initialize input SFDR
SFDR(1,:) = Fin;
SFDR2(1,:) = Fin;
for i=1:N_freq
% truncate the data and window it
output = data(1+(i-1)*N : i*N).*hanning(N);
input = adc_in(1+(i-1)*N : i*N).*hanning(N);

% perform N-pt FFT and put in dBc (dB in ref. to carrier, i.e. max. output)
ff_out = abs(fft(output,N));
ff_in = abs(fft(input,N));
ff_outd = 20*log10(ff_out/max(ff_out));
ff_ind = 20*log10(ff_in/max(ff_in));

% store max peak (magnitude)
n = find(ff_outd==0);
FFOUT(i) = ff_outd(n(1));
nl = find(ff_ind==0);
FFIN(i) = ff_in(nl(1));
```

141
% store spur free dynamic range (SFDR)
outpks = findpeaks(ff_outd(1:N/2));
b = sort([ff_outd(1) outpks],'descend');
for j = 1:length(b)
    if b(j)=0, break; end
end
if b(j)==0, SFDR(2,i)=500; %representing infinity
else
    SFDR(2,i)=abs(b(j));
end
x=find(ff_outd==b(j));
SFDR(3,i)=fstep*(x(1)-1);
SFDR(4,i)=SFDR(3,i)-SFDR(1,i);
if qntz
    SFDR2(2,i)=abs(b(j+1)); %SFDR to 2nd peak
    x=find(ff_outd==b(j+1));
    SFDR2(3,i)=fstep*(x(1)-1);
    SFDR2(4,i)=SFDR2(3,i)-SFDR2(1,i);
end
inpks = findpeaks(ff_ind);
c = sort([ff_ind(1) inpks],'descend');
for j = 1:length(c)
    if c(j)=0, break; end
end
if c(j)==0, SFDR_IN(i)=500; %representing infinity
else
    SFDR_IN(i)=abs(c(j));
end

%% design the xlabels for the plots
labels = [];
for ii=0:Fs/16:Fs/2
    a = floor(((ii/1e6)*1e3))/1e3;
    labels = [labels a];
end

%% plot output spectrum (dBC)
figure; plot(f,ff_outd(1:N/2),'r'); hold on;
pplot(f,ff_outd(1:N/2));
legend('DRFM Input','DRFM Output');
title([titl,dlabef,' [Fs=',num2str(Fs,'%3.0 f'),', ' Hz; Fc= ',num2str(Fc_hz,'%3.0 f'),', ' Hz ; ',SB,' ]']);
axis([0 Fs/2 (min(FF_OUTD)-5) (max(FF_OUTD)+2)]);
set(gca,'XTick',0:Fs/16:Fs/2);
set(gca,'XTickLabel',labels);
xlabel('Frequency (MHz)');
ylabel(' (dBC)');

%% pause
end;

%% plot input freq vs output peak (dBC)
FF_OUTD = 20*log10(FF_OUT/max(FF_OUT));
FF_IND = 20*log10(FF_IN/max(FF_IN));
figure; plot(Fin,FF_OUTD,'-o'); hold on;
pplot(Fin,FF_IND,'-o');
title([titl,dlabef,' [Fs= ',num2str(Fs,'%3.0 f'),', ' Hz; Fc= ',num2str(Fc_hz,'%3.0 f'),', ' Hz ]

axis([0 Fs/2 (min(FF_OUTD)-5) (max(FF_OUTD)+2)]);
set(gca,'XTick',0:Fs/16:Fs/2);
set(gca,'XTickLabel',labels);
xlabel('Input Frequency (MHz)');
ylabel(' (dB)');

%% plot input freq vs SFDR
figure; plot(Fin,SFDR_IN,'-o'); hold on;
pplot(Fin,SFDR(2,:),'-o');
legend('ADC\_IN',dlabef);
title(’SFDR [Fs=’, num2str(Fs, ’%3.0f’), ’ Hz; Fc=’, num2str(Fc_hz, ’%3.0f’), ’ Hz]’);
axis([0 Fs/2 (min(SFDR(2,:))-5) (max(SFDR_IN)+5)]);
set(gca,’XTick’,0:Fs/16:Fs/2);
set(gca,’XTickLabel’,labels);
xlabel([’Input Frequency (MHz) ’ input_range_txt ’ ’]);
ylabel(’SFDR (dBc)’);

%% write sweep data to files
peak(1,:)=Fin;
peak(2,:)=FF_OUTD;
sfdr=SFDR;
sfdr2=SFDR2;
if use_MATLAB
    peak_file=’peak_matlab.txt’;
sfdr_file=’sfdr_matlab.txt’;
sfdr2_file=’sfdr2_matlab.txt’;
else
    peak_file=’peak_matlab_noqntz.txt’;
sfdr_file=’sfdr_matlab_noqntz.txt’;
sfdr2_file=’sfdr2_matlab_noqntz.txt’;
end
else
    peak_file=’peak.txt’;
sfdr_file=’sfdr.txt’;
sfdr2_file=’sfdr2.txt’;
end
dlmwrite(fullfile( design , peak_file ), peak , ’newline’ , ’pc’);
dlmwrite(fullfile( design , sfdr_file ), sfdr , ’newline’ , ’pc’);
dlmwrite(fullfile( design , sfdr2_file ), sfdr2 , ’newline’ , ’pc’);

%% plot time domain sample
T=Ts=10^6;
t=0:T:T*N-T;
adc_in1=adc_in/2^(NI-1);
data=data/2^(NI-1);
figure; subplot(211); plot(t,adc_in1(1:N),’r’);
axis([0 T*100 -1.2 1.2]);
title([’ADC\_IN [Fin=’, num2str(f1, ’%3.0f’), ’ Hz]’]);
xlabel(’Time (\mu s)’); ylabel([’ADC\_IN’]);
subplot(212); plot(t,data(1:N));
axis([0 T*100 -3 .3]);
title([’DRFM\_OUT for’, dlab, [’Fc=’, num2str(Fc_hz, ’%3.0f’), ’ Hz; Fout=’, num2str(Fc_hz+f1, ’%3.0f’), ’ Hz]’]);
xlabel(’Time (\mu s)’); ylabel(’DRFM\_OUT’);
figure; plot(1:200,I(length(I)-199:end)); hold on;
plot(1:200,Q(length(I)-199:length(I)),’:r’);
title(’I and Q signals’)
D.2 Summary Plotting

```matlab
% drfm_plot.m
% Plots DRFM outputs and SFDR's for input frequency-sweeps. Freq and % sideband parameters are assumed to be the same for all designs.

close all;
clear all; clc;

% choose plots
plt1 = 0; % output for separate designs
plt2 = 1; % output for design comparisons
plt4 = 0; % output for all designs
plt5 = 1; % sfdr for separate designs
plt5a = 1; % 8, 16, 24 input bits
plt5b = 1; % 14, 16 coeff bits
plt5c = 1; % 33, 101, 153 taps
plt5d = 1; % 128, 256, 512 ROM words
plt5e = 1; % DRFM 16, 14, 101, 256 corners
plt5f = 1; % DRFM 24, 24, 101, 256 unpiped and piped
plt6 = 1; % sfdr for all designs

% list designs
b(1) = {'DRFM_8_8_101_256'};
b(2) = {'DRFM_16_14_101_256'};
b(3) = {'DRFM_16_16_101_256'};
b(4) = {'DRFM_16_16_33_256'};
b(5) = {'DRFM_16_16_153_256'};
b(6) = {'DRFM_16_16_101_128'};
b(7) = {'DRFM_16_16_101_512'};
b(8) = {'DRFM_24_24_101_256'};
b(9) = {'DRFM_16_14_101_256_ff1p32v_125c'};
b(10) = {'DRFM_16_14_101_256_ss1p08v_125c'};
b(11) = {'DRFM_24_24_101_256_1'};

s(1) = {'DRFM_8_8_101_256_syn'};
s(2) = {'DRFM_16_14_101_256_syn'};
s(3) = {'DRFM_16_16_101_256_syn'};
s(4) = {'DRFM_16_16_33_256_syn'};
s(5) = {'DRFM_16_16_153_256_syn'};
s(6) = {'DRFM_16_16_101_128_syn'};
s(7) = {'DRFM_16_16_101_512_syn'};
s(8) = {'DRFM_24_24_101_256_syn'};
s(9) = {'DRFM_16_14_101_256_ff1p32v_125c_syn'};
s(10) = {'DRFM_16_14_101_256_ss1p08v_125c_syn'};
s(11) = {'DRFM_24_24_101_256_1_syn'};

% load & assign parameters
% assumes: same Ts & sideband were used for all designs, and Fc was % adjusted proportionately for changing ROM sizes.
p = load(fullfile('DRFM_8_8_101_256_syn', 'drfm_param_MATLAB.txt'));
ROM = p(2); % number of words used in ROM
Ts = p(3); % sampling period used (s)
f1 = p(4); % initial input freq#1 (Hz)
f2 = p(5); % initial input freq#2 (Hz)
Fc = p(6); % center frequency factor (from 1 to ROM/2)
fincr = p(7); % input freq. increments (Hz)
Nfreq = p(9); % number of input freqs. in sweep
USB = p(12); % sideband select: '1' for USB, '0' for LSB
if USB
    SB = 'USB';
else
    SB = 'LSB';
end
Fin = f1;
```
% MATLAB

Fc_hz = Fs/(Ts*4*ROM);
Fs = 1/Ts;
input_range_txt = ['from ',num2str(Fin,'%.3f'), ' to ',num2str((Fin+f_incr*(N_freq-1)), '%3.0f')+', Hz, every '+num2str(f_incr, '%3.0f'), ' Hz'];

n = length(b);
for i=1:n  % convert design names for use in titles and legends
    d(i) =strrep(char(b(i)),'-','\-');
end

% load data
for i=0:n-1
    beh_dir = char(b(i+1));
    syn_dir = char(s(i+1));
    peak_beh(1+i*2:2+i*2,:) = dlmread(fullfile(beh_dir,'peak.txt'));
    peak_syn(1+i*2:2+i*2,:) = dlmread(fullfile(syn_dir,'peak.txt'));
    peak_matlab(1+i*2:2+i*2,:) = dlmread(fullfile(syn_dir,'peak_matlab.txt'));

    %peak_matlab_noqntz(1+i*2:2+i*2,:) = dlmread(fullfile(syn_dir,'peak_matlab_noqntz.txt'));
    sfdr_beh(1+i*2:2+i*2,:) = dlmread(fullfile(beh_dir,'sfdr.txt'));
    sfdr_syn(1+i*2:2+i*2,:) = dlmread(fullfile(syn_dir,'sfdr.txt'));
    sfdr_matlab(1+i*2:2+i*2,:) = dlmread(fullfile(syn_dir,'sfdr_matlab.txt'));
end

% design the xlabels for the plots
labels = [];
for ii=0:Fs/16:Fs/2
    a = floor(((ii/1e6)*1e3))/1e3;
    labels = [labels a];
end

% linespec notes for reference
% color=['r';'g';'b';'c';'m';'k';'r';'g'];
% mark=['+';'o';'s';'x';'s';'d';'p';'h'];
% line=['-.';'-.';'-.';'-.';'-.';'-.';'-.'];

% plot outputs for each separate design (compare beh, syn, and matlab)
if plt
    for i=0:n-4
        figure;
        plot(peak_matlab(1+i*2,:),peak_matlab(2+i*2,:),':kx'); hold all;
        plot(peak_beh(1+i*2,:),peak_beh(2+i*2,:),'-b'); hold all;
        plot(peak_syn(1+i*2,:),peak_syn(2+i*2,:),'-ro'); hold all;
        legend(['MATLAB(r)', 'RTL Code', 'Gate-Level Netlist', 'Location', 's']);
        title([(char(d(i+1)),' Output [Fs=',num2str(Fs,'%.3f'),', Hz; Fc=',num2str(Fc_hz,'%.3f'),' Hz]']);
        axis([0 Fs/2 -75 .05]);
        %axis([0 Fs/2 (min(peak_matlab(2+i*2,:))-.5) (max(peak_syn(2+i*2,:))+.5)]);
        set(gca,'XTick',0:Fs/16:Fs/2);
        set(gca,'XTickLabel',labels);
        xlabel(['Input Frequency (MHz) ('input_range_txt,')']);
        ylabel(['DRFM\_OUT (dBc')]);
    end
end

% plot output for design comparisons
if plt2
    % plot output for 8, 16, & 24 input bits
    figure;
    plot(peak_syn(15,:),peak_syn(16,:),'-'); hold all;
end
plot(peak_syn(19,:),peak_syn(20,:),'--'); hold all;
plot(peak_syn(3,:),peak_syn(4,:),':'); hold all;
plot(peak_syn(17,:),peak_syn(18,:),'--'); hold all;

design(2) = strcat(d(2), '\\tlp02v\\25c');
design = [d(10) design(2) d(9)];
legend(design,'Location','s');
title(['Output for Characterization Corners [Fs=\',num2str(Fs,'%3.0f'),'Hz; Fc=\',num2str(Fc_hz,'%3.0f'),'Hz]']);
xlim([0 Fs/2]);
ylim([-4.05]);
set(gca,'XTick',0:Fs/16:Fs/2);
set(gca,'XTickLabel',labels);
xlabel(['Input Frequency (MHz) (',input_range_txt, ')']);
ylabel('DRFM_OUT (dBc)');

end

%% plot output for DRFM\_24,\_24,\_101,\_256 with unpiped and piped Hilbert figure;
plot(peak_syn(15,:),peak_syn(16,:),'--'); hold all;
design(8) = strcat(d(8), 'Unpiped');
plot(peak_syn(21,:),peak_syn(22,:),'-r'); hold all;
design(11) = strcat(d(8), 'Piped');
design = [design(8) design(11)];
legend(design,'Location','s');
title(['Output, Unpiped vs. Piped Filter [Fs=\',num2str(Fs,'%3.0f'),'Hz; Fc=\',num2str(Fc_hz,'%3.0f'),'Hz]']);
xlim([0 Fs/2]);
ylim([-4.05]);
set(gca,'XTick',0:Fs/16:Fs/2);
set(gca,'XTickLabel',labels);
xlabel(['Input Frequency (MHz) (',input_range_txt, ')']);
ylabel('DRFM\_OUT (dBc)');

end

%% plot output for all designs
if plt3
figure;
design={};
for i=0:n-3
plot(peak_syn(1+i*2,:),peak_syn(2+i*2,:),'--'); hold all;
design(i+1,:) = d(i+1);
end
legend(design);
title(['Fs=\',num2str(Fs,'%3.0f')', Hz; Fc=\',num2str(Fc_hz,'%3.0f'), 'Hz']);
xlim([0 Fs/2]);
ylim([-8 1]);
set(gca,'XTick',0:Fs/16:Fs/2);
set(gca,'XTickLabel',labels);
xlabel(['Input Frequency (MHz) (',input_range_txt, ')']);
ylabel('DRFM\_OUT (dBc)');
end

%% plot sfdr for each design separately (compare beh, syn, and matlab)
if plt4
for i=0:n-1
figure;
plot(sfdr_matlab(1+i*2,:),sfdr_matlab(2+i*2,:),'kx'); hold all;
plot(sfdr_beh(1+i*2,:),sfdr_beh(2+i*2,:),'b.'); hold all;
plot(sfdr_syn(1+i*2,:),sfdr_syn(2+i*2,:),'ro'); hold all;
avg_sfdr_matlab = mean(sfdr_matlab(2+i*2,:));
avg_sfdr_beh = mean(sfdr_beh(2+i*2,:));
avg_sfdr_syn = mean(sfdr_syn(2+i*2,:));
end
legend(['MATLAB(r) (mean = ', num2str(avg_sfdr_matlab, '%3.0f'), ', dBc)'], ['RTL Code (mean = ', num2str(avg_sfdr_beh, '%2.2f'), ', dBc)'], ['Gate-Level Netlist (mean = ', num2str(avg_sfdr_syn, '%2.2f'), ', dBc)']);

title(['SFDR for ', char(d(i+1))], ['Fs=', num2str(Fs, '%3.0f'), ', Hz; Fc=', num2str(Fc, '%3.0f')]);

axis([0 Fs/2 (min(sfdr_syn(2+i*2,:)) - 12) (max(sfdr_matlab(2+i*2,:)) + 5)]);

set(gca, 'XTickLabel', 's');

set(gca, 'XTickLabel', labels);

xlabel(['Input Frequency (MHz) (', input_range_txt, ')']);
ylabel('SFDR (dBc)');

if plt55
    plot sfdr design comparisons
    legend([design(1) design(3) design(8)], [design, 'Location', 's']);
    title(['SFDR for 8, 16, & 24 Input Bits [Fs=', num2str(Fs, '%3.0f'), ', Hz; Fc=', num2str(Fc, '%3.0f')]);
    xlim([0 Fs/2]);
    ylim([15 65]);
    set(gca, 'XTickLabel', labels);
    xlabel(['Input Frequency (MHz) (', input_range_txt, ')']);
ylabel('SFDR (dBc)');
end

if plt5b
    figure;
    plot(sfdr_syn(3,:), sfdr_syn(4,:), 'r');
    hold all;
    design(2) = strcat(d(2), [' ', mean = ', num2str(mean(sfdr_syn(4,:)), ' dBc')]);
    plot(sfdr_syn(5,:), sfdr_syn(6,:), 'r');
    hold all;
    design(3) = strcat(d(3), [' ', mean = ', num2str(mean(sfdr_syn(6,:)), ' dBc')]);
    design = [design(2) design(3)];
    legend([design, 'Location', 's']);
    title(['SFDR for 14 & 16 Coeff Bits [Fs=', num2str(Fs, '%3.0f'), ', Hz; Fc=', num2str(Fc, '%3.0f')]);
    xlim([0 Fs/2]);
    ylim([15 65]);
    set(gca, 'XTickLabel', labels);
    xlabel(['Input Frequency (MHz) (', input_range_txt, ')']);
ylabel('SFDR (dBc)');
end

if plt5c
    figure;
    plot(sfdr_syn(9,:), sfdr_syn(10,:), 'r');
    hold all;
    design(5) = strcat(d(5), [' ', mean = ', num2str(mean(sfdr_syn(10,:)), ' dBc')]);
    plot(sfdr_syn(5,:), sfdr_syn(6,:), 'r');
    hold all;
    design(3) = strcat(d(3), [' ', mean = ', num2str(mean(sfdr_syn(6,:)), ' dBc')]);
end

if plt5d
    figure;
    plot(sfdr_syn(11,:), sfdr_syn(12,:), 'r');
    hold all;
    design(5) = strcat(d(5), [' ', mean = ', num2str(mean(sfdr_syn(12,:)), ' dBc')]);
    plot(sfdr_syn(5,:), sfdr_syn(6,:), 'r');
    hold all;
    design(3) = strcat(d(3), [' ', mean = ', num2str(mean(sfdr_syn(6,:)), ' dBc')]);
end

148
hold all

plot(sfdr.syn(7,:), sfdr.syn(8,:), 'r-',  'dBC'); hold all;

 designers = [ design(5), design(3), design(4)];

 legend('SFDR for 33, 101, 153 Filter Taps [Fs=’, num2str(Fs,’%3.0f’), ’Hz; Fc=’, num2str(Fc_hz, ’%3.0f’), ’Hz] ’);
xlim([0 Fs/2]);
ylim([1 5 6 5]);
set(gca,’XTick’,0:Fs/16:Fs/2);
set(gca,’XTickLabel’,labels);
xlabel(’Input Frequency (MHz) ’,’input_range_txt ’, ’ ’);
ylabel(’SFDR (dBC) ’);

end

%% sfdr for 128, 256, & 512 ROM words
if plt5d

 figure;

 plot(sfdr.syn(13,:), sfdr.syn(14,:), ‘r-’); hold all;

 designers = [ design(7), design(3), design(6)];

 legend(’SFDR for 128, 256, & 512 ROM words ’,
 Fs=’, num2str(Fs,’%3.0f’), ’Hz; Fc=’, num2str(Fc_hz, ’%3.0f’), ’Hz] ’);
xlim([0 Fs/2]);
ylim([1 5 6 5]);
set(gca,’XTick’,0:Fs/16:Fs/2);
set(gca,’XTickLabel’,labels);
xlabel(’Input Frequency (MHz) ’,’input_range_txt ’, ’ ’);
ylabel(’SFDR (dBC) ’);

end

%% plot sfdr for DRFM_16_14_101_256 characterization corners
if plt5e

 figure;
 plot(sfdr.syn(19,:), sfdr.syn(20,:), ’r-’); hold all;

 designers = [ design(10), design(3), design(2)];

 legend(’SFDR for Characterization Corners ’,
 Fs=’, num2str(Fs,’%3.0f’), ’Hz; Fc=’, num2str(Fc_hz, ’%3.0f’), ’Hz] ’);
xlim([0 Fs/2]);
ylim([1 5 6 5]);
set(gca,’XTick’,0:Fs/16:Fs/2);
set(gca,’XTickLabel’,labels);
xlabel(’Input Frequency (MHz) ’,’input_range_txt ’, ’ ’);
ylabel(’SFDR (dBC) ’);

end

%% plot sfdr for DRFM_24_24_101_256 with unpiped and piped Hilbert
if plt5f

 figure;

 plot(sfdr.syn(15,:), sfdr.syn(16,:), ’r-’); hold all;

 designers = [ design(8), design(9), design(11), design(12)];

 legend(’Unpiped ’, ’Piped ’, ’dBC ’);

 end
design(11) = strcat(d(8), [' Piped , mean = ', num2str(mean(sfdr_syn(22,:))),' dBc']);

design = [design(8) design(11)];

legend([’SFDR, Unpiped vs. Piped Filter [Fs=’,num2str(Fs,’%3.0f’),’ Hz; Fc=’,num2str(Fc_hz,’%3.0f’),’ Hz’]);

xlim([0 Fs/2]);
ylim([-5 62]);

% % plots sfdr for all designs
if plt6
    figure;
    plot(sfdr_syn(9,:),sfdr_syn(10,:),’- . ’); hold all;
    design(5) = strcat(d(5),[' ', mean = ', num2str(mean(sfdr_syn(10,:))),' dBc']);
    plot(sfdr_syn(13,:),sfdr_syn(14,:),’- . ’); hold all;
    design(7) = strcat(d(7),[' ', mean = ', num2str(mean(sfdr_syn(14,:))),' dBc']);
    plot(sfdr_syn(15,:),sfdr_syn(16,:),’- . ’); hold all;
    design(8) = strcat(d(8),[' ', mean = ', num2str(mean(sfdr_syn(16,:))),' dBc']);
    plot(sfdr_syn(5,:),sfdr_syn(6,:),’- . ’); hold all;
    design(3) = strcat(d(3),[' ', mean = ', num2str(mean(sfdr_syn(6,:))),' dBc']);
    plot(sfdr_syn(3,:),sfdr_syn(4,:),’- . ’); hold all;
    design(2) = strcat(d(2),[' ', mean = ', num2str(mean(sfdr_syn(4,:))),' dBc']);
    plot(sfdr_syn(11,:),sfdr_syn(12,:),’- . ’); hold all;
    design(6) = strcat(d(6),[' ', mean = ', num2str(mean(sfdr_syn(12,:))),' dBc']);
    plot(sfdr_syn(1,:,),sfdr_syn(2,:),’- . ’); hold all;
    design(1) = strcat(d(1),[' ', mean = ', num2str(mean(sfdr_syn(2,:))),' dBc']);
    plot(sfdr_syn(7,:),sfdr_syn(8,:),’- . ’); hold all;
    design(4) = strcat(d(4),[' ', mean = ', num2str(mean(sfdr_syn(8,:))),' dBc']);

    design = [design(5) design(7) design(8) design(3) design(2) design(6) design(1)
        design(4)];

    legend(design,’Location’,’s’);
    title([’SFDR for All Designs (highest to lowest avg) [Fs=’,num2str(Fs,’%3.0f’),’ Hz;
        Fc=’,num2str(Fc_hz,’%3.0f’),’ Hz’]);
    xlim([0 Fs/2]);
ylim([-5 62]);

end

end

end
Bibliography


11. B. P. Lathi, Signal processing and linear systems. New York: Oxford University Press, 1998. B.P. Lathi; ;ill. ;26 cm; Includes bibliographical references (p. 837-838) and index.


