Improving the Single Event Effect Response of Triple Modular Redundancy on SRAM FPGAs Through Placement and Routing

Matthew Joel Cannon
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/7551

This Dissertation is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Improving the Single Event Effect Response of Triple Modular Redundancy
on SRAM FPGAs Through Placement and Routing

Matthew Joel Cannon

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Michael Wirthlin, Chair
James Archibald
Jeffrey Goeders
Philip Lundrigan

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2019 Matthew Joel Cannon
All Rights Reserved
ABSTRACT

Improving the Single Event Effect Response of Triple Modular Redundancy on SRAM FPGAs Through Placement and Routing

Matthew Joel Cannon
Department of Electrical and Computer Engineering, BYU
Doctor of Philosophy

Triple modular redundancy (TMR) with repair is commonly used to improve the reliability of systems. TMR is often employed for circuits implemented on field programmable gate arrays (FPGAs) to mitigate the radiation effects of single event upsets (SEUs). This has proven to be an effective technique by improving a circuit’s sensitive cross-section by up to $100\times$. However, testing has shown that the improvement offered by TMR is limited by upsets in single configuration bits that cause TMR to fail.

This work proposes a variety of mitigation techniques that improve the effectiveness of TMR on FPGAs. These mitigation techniques can alter the circuit’s netlist and how the circuit is placed and routed on the FPGA. TMR with repair showed a neutron cross-section improvement of $100\times$ while the best mitigation technique proposed in this work showed an improvement of $700\times$.

This work demonstrates both some causes behind single bit SEU failures for TMR circuits on FPGAs and mitigation techniques to address these failures. In addition to these findings, this work also shows that the majority of radiation failures in these circuits are caused by multiple cell upsets, laying the path for future work to further enhance the effectiveness of TMR on FPGAs.

Keywords: FPGA, reliability, SEE, SEU, radiation effects, TMR, MTTF, fault injection, radiation testing, Markov chain, single bit failure, single point failure, common mode failure, common cause failure, CAD, incremental placement, incremental routing
ACKNOWLEDGMENTS

This dissertation represents the culmination of many years of hard work and preparation. Its completion and my graduation were only possible with the help of many friends and family over the years. I would first like to thank my family. My parents, Bruce and Doreen, have been a constant source of support and encouragement throughout both graduate school and my undergraduate. I would also like to thank all of my siblings, my sister Danielle and my brothers Brian, Wesley and Jordan. I would also like to express my gratitude for my late brother, Joshua, whose example and enthusiasm first interested me in computers and technology.

I would like to thank all my friends who have influenced me over the years. In particular, I would like to thank Doug and Daniela Harding, who have been like family to me throughout my studies here in Utah.

I have had many wonderful colleagues in the Configurable Computing Lab who have provided assistance and support to this work. I would like to thank all of them and especially the members of the TMR team, in particular, Andrew Keller, Hayden Rowberry and Corbin Thurlow. Their contributions have made this work a reality. I would also like to thank my advisor, Dr. Michael Wirthlin who has spent countless hours advising this work, as well as all of the other members on my committee.

Finally, I would like to thank the sponsors of this work that have also made this research a reality. This work has been granted beam time at LANSCE numerous times under proposals NS-2017-7574-A, NS-2017-7574-F and NS-2018-7895-A. I have also received funding through the Utah NASA Space Grant Consortium (UNSGC). This research has also been supported by the I/UCRC Program of the National Science Foundation under Grant Numbers 1265957 and 1738550, through the NSF Center for High-Performance Reconfigurable Computing (CHREC) and the NSF Center for Space, High-performance, and Resilient Computing (SHREC). The opinions, findings, and conclusions or recommendations expressed are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
# TABLE OF CONTENTS

List of Tables ........................................................................... viii

List of Figures ........................................................................... x

Chapter 1  Introduction ................................................................. 1
  1.1 SRAM FPGAs in Space ......................................................... 2
  1.2 Mitigation for FPGAs in Space ............................................. 4
  1.3 Contributions of this Work .................................................. 6
  1.4 List of Publications ............................................................. 7
    1.4.1 Accepted Publications ............................................... 7
    1.4.2 Submitted Publications ............................................. 8

Chapter 2  Radiation Effects in Electronics and FPGAs ................. 9
  2.1 Radiation Effects in Electronics ........................................... 9
    2.1.1 Single Event Upsets .................................................. 9
    2.1.2 Single Event Transients .......................................... 12
    2.1.3 Single Event Functional Interrupt ............................. 15
  2.2 Single Event Effects in FPGAs ........................................... 15
    2.2.1 Single Event Upsets in FPGAs ................................. 16
    2.2.2 Other SEEs in FPGAs ........................................... 18
  2.3 Measuring the Reliability of FPGA Circuits ......................... 19
    2.3.1 Radiation Testing .................................................. 19
    2.3.2 Fault Injection .................................................... 20
    2.3.3 Reliability Metrics ............................................... 21
    2.3.4 Reliability of the Circuit with No Mitigation ................ 26

Chapter 3  Implementation of TMR on FPGAs ............................. 28
  3.1 Basic TMR Insertion ........................................................ 28
    3.1.1 TMR Insertion Strategies ........................................ 31
  3.2 Techniques to Increase TMR Effectiveness ......................... 33
    3.2.1 Circuit Triplication and Partitioning .......................... 33
    3.2.2 Voter Triplication ............................................... 36
    3.2.3 Repair .............................................................. 37
    3.2.4 Cross Clock Domain Timing Synchronization .................. 41
  3.3 Limitations of Current TMR Tools ...................................... 43
  3.4 Summary .......................................................................... 45

Chapter 4  Single Bit Failures in TMR .......................................... 46
  4.1 Types of SBF in TMR Systems .......................................... 47
    4.1.1 Single Point Failures .............................................. 47
    4.1.2 Common Mode Failures ......................................... 49
  4.2 How Single Point Failures Occur in FPGA Circuits .................. 49
4.2.1 Single Voter Failures ................................................. 51
4.2.2 SPF From the Circuit Implementation ................................. 52
4.3 Common Mode Failures .................................................. 55
  4.3.1 FPGA Routing Network ............................................. 55
  4.3.2 Routing CMF ....................................................... 59
  4.3.3 SEUs in a Used Routing Mux .................................... 62
  4.3.4 Clock CMF ......................................................... 64
  4.3.5 Site CMF .......................................................... 66
  4.3.6 CMF Distribution Analysis ....................................... 67
4.4 Previous Studies on Single Bit Failures .............................. 68
4.5 Summary .................................................................. 69

Chapter 5 Mitigation for Single Bit Failures in TMR .................... 71
  5.1 Single Point Failure Mitigation Techniques ............................ 71
    5.1.1 Split-Clock SPF Mitigation Technique .......................... 72
    5.1.2 Split I/O SPF Mitigation Technique .............................. 74
    5.1.3 Output Placement .................................................. 77
    5.1.4 Early Split SPF Mitigation Technique .......................... 78
  5.2 Common Mode Failure Mitigation Techniques .......................... 79
    5.2.1 Fail Sets .......................................................... 80
    5.2.2 Downstream Cells ................................................ 82
    5.2.3 Incremental Placement (PCMF) ................................ 82
    5.2.4 PCMF Algorithm Results ....................................... 87
    5.2.5 Striping ........................................................... 90
    5.2.6 Combining CMF Mitigation with SPF Mitigation Techniques . 92
  5.3 Previous Work ................................................................ 93
  5.4 Summary .................................................................. 96

Chapter 6 Reliability Improvement and Testing Results ............... 97
  6.1 Test Structure and Methodology ...................................... 97
    6.1.1 Benchmark Circuits .............................................. 102
  6.2 Fault Injection ........................................................... 102
  6.3 Neutron Radiation Testing .............................................. 105
  6.4 Replay of Radiation Test ............................................... 111
  6.5 Summary .................................................................. 114

Chapter 7 Conclusion ........................................................... 116
  7.1 Future Work ................................................................ 117
    7.1.1 Early Split Custom Routing .................................... 117
    7.1.2 Domain Placement ................................................ 117
    7.1.3 SPF In Specialized Resources .................................. 117
    7.1.4 Techniques for Site CMF ....................................... 118
    7.1.5 MCU Mitigation Techniques .................................. 118
    7.1.6 Heavy Ion Testing ................................................ 118
    7.1.7 Applicability to Other Devices ................................. 119
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.8</td>
<td>Algorithm Improvement</td>
<td>119</td>
</tr>
<tr>
<td>7.2</td>
<td>Concluding Remarks</td>
<td>119</td>
</tr>
<tr>
<td>Appendix A</td>
<td>General FPGA CAD</td>
<td>121</td>
</tr>
<tr>
<td>A.1</td>
<td>Synthesis</td>
<td>122</td>
</tr>
<tr>
<td>A.2</td>
<td>Pack/Place</td>
<td>122</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Packing</td>
<td>122</td>
</tr>
<tr>
<td>A.2.2</td>
<td>Placement</td>
<td>122</td>
</tr>
<tr>
<td>A.3</td>
<td>Routing</td>
<td>123</td>
</tr>
<tr>
<td>A.4</td>
<td>TMR</td>
<td>123</td>
</tr>
<tr>
<td>A.4.1</td>
<td>Constrained Variations</td>
<td>124</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Separate CAD Flow Variations</td>
<td>124</td>
</tr>
<tr>
<td>A.5</td>
<td>Open Source CAD Tools</td>
<td>124</td>
</tr>
<tr>
<td>Appendix B</td>
<td>7-Series Architecture Overview</td>
<td>126</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Reliability Metric Derivations and TMR Modeling</td>
<td>131</td>
</tr>
<tr>
<td>C.1</td>
<td>Block Diagrams and Fault Trees</td>
<td>132</td>
</tr>
<tr>
<td>C.2</td>
<td>Markov Chain</td>
<td>134</td>
</tr>
<tr>
<td>C.2.1</td>
<td>Traditional TMR Model</td>
<td>138</td>
</tr>
<tr>
<td>C.2.2</td>
<td>N Modular Redundancy</td>
<td>139</td>
</tr>
<tr>
<td>C.2.3</td>
<td>TMR with Repair Model</td>
<td>143</td>
</tr>
<tr>
<td>C.2.4</td>
<td>TMR Partitioning</td>
<td>144</td>
</tr>
<tr>
<td>Appendix D</td>
<td>Modeling Common Cause Failures in TMR</td>
<td>150</td>
</tr>
<tr>
<td>D.1</td>
<td>TMR with CCF Model</td>
<td>150</td>
</tr>
<tr>
<td>D.1.1</td>
<td>TMR</td>
<td>151</td>
</tr>
<tr>
<td>D.1.2</td>
<td>TMR with Repair</td>
<td>153</td>
</tr>
<tr>
<td>D.1.3</td>
<td>Partitioning</td>
<td>157</td>
</tr>
<tr>
<td>Appendix E</td>
<td>Additional CMF Removal Algorithms</td>
<td>161</td>
</tr>
<tr>
<td>E.1</td>
<td>Fail Set Algorithm for Partitioned Circuits</td>
<td>161</td>
</tr>
<tr>
<td>E.2</td>
<td>Incremental Routing (RCMF)</td>
<td>164</td>
</tr>
<tr>
<td>E.2.1</td>
<td>Identifying Routing CMF</td>
<td>165</td>
</tr>
<tr>
<td>E.2.2</td>
<td>Routing CMF Removal</td>
<td>165</td>
</tr>
<tr>
<td>E.2.3</td>
<td>Fault Injection Results</td>
<td>166</td>
</tr>
<tr>
<td>Appendix F</td>
<td>Implementation and Fault Injection Results for Each Circuit</td>
<td>168</td>
</tr>
<tr>
<td>F.1</td>
<td>Implementation Metrics</td>
<td>168</td>
</tr>
<tr>
<td>F.2</td>
<td>Circuit Specific Fault Injection Results</td>
<td>170</td>
</tr>
<tr>
<td>Appendix G</td>
<td>Glossary &amp; Acronyms</td>
<td>176</td>
</tr>
<tr>
<td>G.1</td>
<td>Glossary</td>
<td>176</td>
</tr>
<tr>
<td>G.2</td>
<td>Differences Between Similar Terms</td>
<td>184</td>
</tr>
<tr>
<td>G.3</td>
<td>Acronyms</td>
<td>185</td>
</tr>
</tbody>
</table>
G.4 TMR Variations ................................................................. 187

Appendix H Scripts Used In This Work ........................................ 188
  H.1 Reliability Graphs .......................................................... 189
  H.2 Find Maximum Frequency ............................................... 190
  H.3 TMR Variation Generator ................................................ 190
      H.3.1 Early Split Constraints .......................................... 191
      H.3.2 Striping Constrains ............................................... 193

References ............................................................................. 195
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Base Circuit Details</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>Unmitigated Fault Injection Results</td>
<td>27</td>
</tr>
<tr>
<td>2.3</td>
<td>Neutron Radiation Testing Results for No Mitigation of b13 Circuit</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>SBF in Trip-IO TMR Circuits</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>SBF in Common-IO TMR Circuits</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Single Voter Failures in TMR Circuits</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>SPF in TMR Circuits</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>CMF in Triplicated I/O TMR Circuits</td>
<td>67</td>
</tr>
<tr>
<td>4.6</td>
<td>Clock CMF Susceptibility of Circuits</td>
<td>68</td>
</tr>
<tr>
<td>5.1</td>
<td>Common-IO (3-Voter) Implementation Metrics</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Routing Nodes Utilized by Untriplicated Nets in Common-IO (3-Voter)</td>
<td>72</td>
</tr>
<tr>
<td>5.3</td>
<td>Split-Clock Routing Nodes Utilized by Untriplicated Nets</td>
<td>73</td>
</tr>
<tr>
<td>5.4</td>
<td>Split-Clock Implementation Metrics</td>
<td>75</td>
</tr>
<tr>
<td>5.5</td>
<td>Split-IO Implementation Metrics</td>
<td>77</td>
</tr>
<tr>
<td>5.6</td>
<td>Split-IO Routing Nodes Utilized by Untriplicated Nets</td>
<td>77</td>
</tr>
<tr>
<td>5.7</td>
<td>Early-Split Implementation Metrics</td>
<td>79</td>
</tr>
<tr>
<td>5.8</td>
<td>Early-Split Routing Nodes Utilized by Untriplicated Nets</td>
<td>79</td>
</tr>
<tr>
<td>5.9</td>
<td>Trip-IO (3-Voter) Implementation Metrics</td>
<td>80</td>
</tr>
<tr>
<td>5.10</td>
<td>PCMF Implementation Metrics</td>
<td>88</td>
</tr>
<tr>
<td>5.11</td>
<td>PCMF Algorithm Metrics</td>
<td>88</td>
</tr>
<tr>
<td>5.12</td>
<td>Striping Implementation Metrics</td>
<td>93</td>
</tr>
<tr>
<td>5.13</td>
<td>Split-Clock-PCMF Implementation Metrics</td>
<td>93</td>
</tr>
<tr>
<td>5.14</td>
<td>Early-Split-PCMF Implementation Metrics</td>
<td>94</td>
</tr>
<tr>
<td>6.1</td>
<td>Fault Injection Geomean Results</td>
<td>105</td>
</tr>
<tr>
<td>6.2</td>
<td>Neutron Radiation Testing Results</td>
<td>109</td>
</tr>
<tr>
<td>6.3</td>
<td>Classification of Failures from Radiation Test</td>
<td>112</td>
</tr>
<tr>
<td>6.4</td>
<td>Neutron Radiation Testing Results Without Accumulation</td>
<td>113</td>
</tr>
<tr>
<td>6.5</td>
<td>Resources Affected by MCUs</td>
<td>113</td>
</tr>
<tr>
<td>6.6</td>
<td>Number of Bit Upsets per SEU</td>
<td>114</td>
</tr>
<tr>
<td>E.1</td>
<td>PCMF Algorithm Metrics</td>
<td>164</td>
</tr>
<tr>
<td>E.2</td>
<td>RCMF Fault Injection Results for b13</td>
<td>167</td>
</tr>
<tr>
<td>F.1</td>
<td>Mean Implementation Metrics</td>
<td>169</td>
</tr>
<tr>
<td>F.2</td>
<td>Implementation Metrics for b13</td>
<td>170</td>
</tr>
<tr>
<td>F.3</td>
<td>Implementation Metrics for md5</td>
<td>170</td>
</tr>
<tr>
<td>F.4</td>
<td>Implementation Metrics for sha3</td>
<td>171</td>
</tr>
<tr>
<td>F.5</td>
<td>Implementation Metrics for aes128</td>
<td>171</td>
</tr>
<tr>
<td>F.6</td>
<td>Fault Injection Results for b13</td>
<td>172</td>
</tr>
<tr>
<td>F.7</td>
<td>Fault Injection Results for md5</td>
<td>173</td>
</tr>
<tr>
<td>F.8</td>
<td>Fault Injection Results for sha3</td>
<td>174</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>General FPGA Architecture</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Triple Modular Redundancy</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Example of an SEU</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of an MBU Event</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Basic SRAM Cell - Cross Coupled Inverter</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Six Transistor SRAM Cell</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of an SET from [1]</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Standard D Flip-Flop</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of Two DFFs Connected Through Logic</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>LUT Contents Define Circuits</td>
<td>16</td>
</tr>
<tr>
<td>2.9</td>
<td>SEUs Change the Design Implemented on an FPGA</td>
<td>17</td>
</tr>
<tr>
<td>2.10</td>
<td>Effects of SEUs in the Routing Network</td>
<td>18</td>
</tr>
<tr>
<td>2.11</td>
<td>FPGA Board in a Neutron Radiation Beam</td>
<td>20</td>
</tr>
<tr>
<td>2.12</td>
<td>Random Fault Injection Flow</td>
<td>21</td>
</tr>
<tr>
<td>2.13</td>
<td>Reliability of Simplex and TMR Systems</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Triple Modular Redundancy on an FPGA</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Reliability of TMR with Repair</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Decomposing a System into Sub-Systems</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>Partitioning - Applying TMR to a Decomposed System</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Multiple Failures in a Partitioned TMR System</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Multiple Failures Causing a System Failure in a Partitioned TMR System</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>TMR with a Single Voter</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>TMR without Feedback</td>
<td>39</td>
</tr>
<tr>
<td>3.9</td>
<td>TMR with Feedback</td>
<td>40</td>
</tr>
<tr>
<td>3.10</td>
<td>Example of Cross Clock Domain Synchronization Issues in TMR</td>
<td>42</td>
</tr>
<tr>
<td>3.11</td>
<td>Example of How to Fix Cross Clock Domain Synchronization Issues in TMR</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>TMR with SPF</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of TMR FPGA Circuit with SPF in the I/O</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of TMR FPGA Circuit with SPF in a Limited Resource</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>Single Voter TMR Example</td>
<td>51</td>
</tr>
<tr>
<td>4.5</td>
<td>SPFs on a High Fanout Input Net (a) and on a Reduction Output Net (b)</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>Example of Route Tree for a Net with Multiple Sinks</td>
<td>54</td>
</tr>
<tr>
<td>4.7</td>
<td>Source Wire with Multiple Possible Sink Wires</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>Example of a Routing Switchbox with a Valid Configuration</td>
<td>57</td>
</tr>
<tr>
<td>4.9</td>
<td>Example of How Sink Wires are Driven with an Invalid Configuration</td>
<td>57</td>
</tr>
<tr>
<td>4.10</td>
<td>Xilinx Routing Mux</td>
<td>58</td>
</tr>
<tr>
<td>4.11</td>
<td>Example Configuration of a Routing Mux</td>
<td>60</td>
</tr>
<tr>
<td>4.12</td>
<td>Unused Routing Mux</td>
<td>61</td>
</tr>
<tr>
<td>4.13</td>
<td>Row Bit SEU in an Unused Routing Mux</td>
<td>61</td>
</tr>
</tbody>
</table>
4.14 Column Bit SEU in an Unused Routing Mux ........................................ 62
4.15 Row Short in a Routing Mux .......................................................... 63
4.16 Column Short in a Routing Mux ....................................................... 64
4.17 Local Clock Routing. Not All Wires/Muxes are Shown. ....................... 65
4.18 How One SBU Can Affect Multiple TMR Modules from [3] .................. 68
5.1 Triplicating Clocks ........................................................................... 73
5.2 Split I/O Technique .......................................................................... 75
5.3 Pass-Through LUTs .......................................................................... 76
5.4 Untriplicated Output Net .................................................................... 78
5.5 Flow for Implementing CMF Mitigation Techniques .............................. 80
5.6 Downstream Cell Failures ................................................................. 82
5.7 Flow for Implementing PCMF ............................................................. 83
5.8 Example of a Swap During the PFCMF Algorithm ................................. 85
5.9 Example of a Swap with a Carry Chain Dependency ............................ 89
5.10 Example of a Carry Chain Dependency .............................................. 90
5.11 Striping Technique. Each Domain is Constrained to Every Third Column. 92
6.1 DUT with Golden Copy Test Structure ............................................. 98
6.2 TURTLE Setup Using a Master/Slave Test Structure ........................... 99
6.3 TURTLE Pond .................................................................................. 101
6.4 Fault Injection Sensitivity of Designs/Circuits ..................................... 104
6.5 Flight Paths at LANSCE [4] ............................................................... 106
6.8 Neutron Cross-Section of Designs for b13 Circuit ............................... 110
A.1 FPGA CAD Flow ............................................................................. 121
A.2 Netlist TMR CAD Flow ................................................................. 123
A.4 RapidWright/Vivado Interface using a DCP File [7] ............................ 125
B.1 Relationship Between CLBs and Slices [8] ....................................... 126
B.2 Diagram of a SLICE from [8] .......................................................... 128
B.3 Diagram of a SLICEM from [8] ...................................................... 129
B.4 Routing a Clock to a Logic Slice ..................................................... 130
C.1 Simplex Block Diagram Model ......................................................... 132
C.2 TMR Block Diagram Model .............................................................. 132
C.3 Simplex Reliability Model ............................................................... 135
C.4 Simplified Simplex Reliability Model ............................................. 135
C.5 Traditional TMR Reliability Model .................................................. 138
C.6 NMR Reliability Model for n Modules ............................................ 140
C.7 Comparison of NMR systems ......................................................... 142
C.8 TMR with Repair Reliability Model ............................................... 143
CHAPTER 1. INTRODUCTION

Space, sometimes referred to as the “final frontier”, offers an exciting new environment full of exploratory and research opportunities. In the near future, NASA plans to study our solar system and beyond, send manned missions back to the moon, continue the research on the international space station, improve flight and satellite technology and to continue studying our planet from the vantage of space [9]. This type of exploration can lead to answers about the history of the universe, help create new industries and push the boundaries of current scientific and technical knowledge [10].

Current satellites have an integral role in current life. Satellites are used everyday to help farmers with irrigation and harvesting. Satellites are also used to broadcast entertainment media, make phone calls and access the internet. They can be used to monitor the transportation infrastructure and current weather conditions [11]. As more and more satellites are deployed each day their influence on daily life will continue to grow.

Current and future satellites require complex electronics and systems in order to function correctly. Common systems on satellites include propulsion, power, communications, thermal, attitude and the telemetry. These systems rely on the data provided by internal and external sensors on the satellite to continue to operate [12]. These sensors can range from radars and accelerometers to temperature sensors. The outputs of these sensors can be used to adjust the attitude of the satellite and to assess the overall health of the satellite.

The output of these sensors is also crucial to the overall mission of the satellite. Satellites are deployed to collect, process and transmit data back to the ground users. One well known satellite application is to capture pictures of the Earth’s surface. Such satellites are responsible for capturing the images and then transmitting back to the ground. Weather satellites are another application that captures sensor data about atmospheric conditions such as the current temperature and precipitation levels.
Once the sensor data has been captured it needs to be transmitted back to the ground. In most satellite systems there is often a disparity between how much data is produced and how much data can actually be transmitted back to the ground station. For example, the New Horizons space probe recently completed a mission of taking measurements of Pluto during a flyby and then transmitting the data back to Earth. In total about 6.25 GB of data was transmitted back to Earth at a rate of 1-2 kbps, taking about 15 months in total [13]. This was acceptable for this mission as the probe was not continuously collecting data, but this would be a problem for more real-time systems that are simultaneously collecting and transmitting data.

The problem of processing and transmitting data is exacerbated as modern satellites continue to add more, high-resolution sensors, increasing the amount of raw data being produced. In a study from 2009, JPL estimates that their spectroscopy instruments can produce 1-5 TB of raw data per day, which is two orders of magnitude more data than can typically be downlinked [14]. In another satellite launched in 2018, the Polarimetric and Helioseismic Imager instrument (PHI) was deployed in the scientific payload with an active pixel sensor (APS) with a resolution of $2048 \times 2048$ pixels. This particular application required a cadence of only one minute to record a complete data set of 3.22 Gbit and a peak data rate of 300 Mbits/s [15].

The increasing amount of raw data being generated by these space systems creates a requirement for a computational system that can handle this amount of data. This computation system would need to collect, preprocess and compress the raw data before it is transmitted back to Earth. The European Space Agency (ESA) has estimated one future mission could require 10 GOPS (Giga-operations per second) of processing power for both fixed and floating point operations at a peak data rate of 2.2 Gbit/s. Another Earth observation mission could require up to 1000 GOPS of processing power for fixed point operations with a peak data rate of 500 Gbit/s [16].

1.1 SRAM FPGAs in Space

SRAM field programmable gate arrays (FPGA) would be ideal for this computational system handling large amounts of data. FPGAs contain a large array of digital signal processors (DSP), high speed input/output (I/O) ports, large block memories (BRAM) and a large configurable fabric which can be reprogrammed to tailor the FPGA for a specific task, as shown in Figure 1.1. The current function of the FPGA is defined by the contents of the SRAM configuration memory.
The combination of DSPs, BRAMs, high speed I/O and configurable fabric make FPGAs ideal for processing the raw sensor data [17]. For Xilinx 7-Series devices, built on a 28-nm process, the Virtex-7 690T is able to achieve 3.12 TFLOPS of processing power. Even the smaller Artix-7 200T device (used throughout this work) is able to achieve a processing power of 648 GFLOPS while consuming only 9 W of power, a power efficiency of 72 GFLOPS/W [18].

The great processing power and power efficiency of FPGAs have led researchers to study how to use FPGAs in space-based systems. The JPL study from 2009 proposed an FPGA solution to reduce the amount of data from 1-5 TB per day to 100s of GB per day [14]. In this work, the authors studied a possible architecture using a Xilinx Virtex-4 FPGA. The authors first considered using a more typical computing device, but did not go with this option due to concerns about the power and available on-chip memories. The authors also considered a DSP solution, but chose not to pursue this due to limited knowledge about whether fixed-point or floating point arithmetic would be needed.

An FPGA was chosen because it overcame the obstacles raised by the other device options. As stated above, FPGAs have great power efficiency and available on-chip memories. They also contain DSPs that can be configured for either floating-point or fixed-point arithmetic operations, providing design flexibility when all of the mission parameters are not known early in the design
cycle. In addition to these features, it is relatively easy to configure the FPGA to work with many different communication interfaces and protocols.

One of the concerns with using a FPGAs in space is their susceptibility radiation induced single event upsets (SEU), which constitute a change in a memory cell [19, 20]. SEUs are of a particular concern for SRAM FPGAs because a change in one of the SRAM configuration cells signifies a change in the configuration of the device. SEUs can have varying affects on the circuit depending on what the affected bit controls. An SEU could occur in a memory element, corrupting the current state of the circuit or in a architectural bit which could change the implemented circuit behavior. Depending on where the SEU occurs, it could disrupt normal circuit behavior by corrupting the output data stream or causing other undesirable behavior. The SEU problem needs to be properly addressed before adopting an FPGA in any space system.

1.2 Mitigation for FPGAs in Space

There has been great interest in overcoming the radiation problem, so FPGAs can be more widely adopted for use in space. Many research studies have investigated possible mitigation strategies to make FPGAs more radiation resilient [21]. There are two categories of mitigation strategies: failure recovery and failure masking.

The goal behind failure recovery techniques is not to prevent failures from occurring, but to correct the circuit so that the failure ceases. This is done by applying a repair mechanism called scrubbing, where the configuration memory of the FPGA is continually scrubbed to correct any erroneous bits that resulted from the effects of SEUs. By applying scrubbing, the original circuit is restored to the FPGA device.

The goal behind failure masking techniques is to actively mask the effects of SEUs so that failures are tolerated for some amount of time. This is achieved by adding some form of redundancy to the circuit. The most popular form of redundancy is triple modular redundancy (TMR), where the circuit is replicated three times and a voter is placed on the output, as shown in Figure 1.2. The voter is used so that only the majority value from the three redundant circuits is propagated to the rest of the system. TMR allows one of the redundant circuits to fail without affecting the rest of the system as the other two redundant circuits will out-vote the failed circuit.
If another one of the redundant circuits fails, then two of the redundant circuits have failed and the voter can no longer propagate the correct output.

The most effective mitigation strategy is to combine the failure masking and failure recovery techniques and implement TMR with repair. In TMR with repair, the circuit can both mask and recover from failures. When one of the redundant circuits fails, the system has an opportunity to repair that circuit. The system remains in a correct state if every singular redundant circuit failure is repaired before a failure occurs in another one of the redundant circuits. Using TMR with repair on FPGAs has shown to improve the average lifetime in a radiation environment by up to $100\times$ [22].

Increasing the average lifetime of a circuit in space by $100\times$ is a great improvement, but may not be enough improvement for all missions. Some missions may require a circuit lifetime with higher lifetime or reliability. For example, some long term probe missions may be sent into deep space and should collect and transmit data for as long as possible, or until the devices themselves fail (for reasons unrelated to radiation). Another example would be safety critical systems, where devices need to operate correctly with near certainty. For such missions, the system designers need to carefully consider the cost, power, computational performance and reliability constraints of the mission against potential devices available to them.

FPGA devices offer great power and computational performance and are affordable in small quantities, but are still not widely deployed due to their radiation sensitivity. These missions where FPGAs are not deployed may be sacrificing computational performance and/or cost in favor of
radiation sensitivity. If the radiation resiliency of FPGA devices was improved, it could lead to wider adoption of FPGAs in space.

The focus of this dissertation is to identify how upsets in single configuration bits can cause TMR to fail and then to provide mitigation techniques for these single bit failures (SBF). As the results section will show, the proposed mitigation techniques can improve the radiation reliability by several orders of magnitude. By successfully improving the radiation resiliency of FPGAs in space, this dissertation enables advances in the processing capabilities of future space systems.

1.3 Contributions of this Work

This work extends on the state of the art by offering four major contributions. The first contribution is this work provides an architecture-level theory into how single bits cause failures in TMR designs on FPGAs. This is achieved by examining specific architectural structures on the FPGA device and exploring how single bit upsets in these structures lead to multiple TMR domain failures.

The second contribution is that this work offers several techniques to mitigate single bit failures in FPGAs. Specific algorithms to perform these techniques are shown based on the architecture-level theory of how single bits cause failures. These techniques are automated using these algorithms and require minimal user input.

The third contribution of this work is the extensive fault injection and neutron radiation data on designs that use the proposed mitigation techniques of this work. Fault injection data is collected for all of the TMR mitigation strategies across four different benchmark circuits and is used to show how effective the proposed mitigation techniques are for a variety of circuit designs. Neutron radiation testing is then used to validate the efficacy of a selection of the proposed techniques in a radiation environment, showing a $700 \times$ improvement over no mitigation.

The fourth contribution of this work is the development of theoretical Markov chains that model single bit failures in TMR systems. Single bit failures are modeled for multiple TMR circuits. These models are also applicable to general systems and not just the FPGA systems used in this work.

This work offers a wide variety of mitigation techniques, and theory behind why these mitigation techniques are effective, for radiation induced failures. By offering a wide variety of
mitigation techniques, the design engineer can choose a technique that is best for their particular mission. This will enable the engineer to design a circuit that is both high performance and high reliability.

1.4 List of Publications

This work has resulted in several publications as outlined below. Some of these publications have been accepted and published, while others are still in the review process.

1.4.1 Accepted Publications


  This paper explored some causes into how common mode failures occur on Xilinx 7-Series FPGAs followed by several mitigation strategies to mitigate common mode failures.

- “Improving the Effectiveness of TMR Designs on FPGAs with SEU-Aware Incremental Placement,” Presented at the 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), April, 2018. [24]

  This paper explored the cause behind clock common mode failures in Xilinx 7-Series FPGAs and then explored one way to mitigate clock common mode failures.


  This paper explored the reliability of multi-gigabit transceivers (MGT) while exposed to proton radiation. The paper concluded that the FPGA’s configurable fabric was more sensitive to failures than the MGTs themselves.
1.4.2 Submitted Publications


  This paper explores some of the causes behind single point failures in SRAM FPGAs and then proposes several mitigation techniques for single point failures. These techniques are tested on Xilinx 7-Series FPGAs.


  This paper explores the theoretical limitations imposed by common cause failure in TMR systems.
CHAPTER 2. RADIATION EFFECTS IN ELECTRONICS AND FPGAS

Commercial FPGAs, like all electronics, are subject to radiation effects when struck by an ionizing particle. Depending on where the particle strikes, the angle of the strike, the energy of the particle, what is implemented on the device, etc., there can be a wide range of effects. Before any device is deployed in a high radiation environment such as space [26], these events should be characterized and understood to ensure correct operation during the intended lifetime of the device.

The goal of this chapter is to introduce basic radiation effects concepts, especially those that are pertinent to FPGAs. After the basic concepts are introduced, the chapter will then focus on how these radiation effects can lead to observable failures on an FPGA. This chapter then concludes by introducing testing methods and metrics used to measure a device’s and circuit’s radiation reliability. Throughout this chapter and dissertation, terminology will be used assuming the reader knows their meaning. Appendix G.1 contains a comprehensive glossary for the reader that is unfamiliar with some of the terms.

2.1 Radiation Effects in Electronics

When a high energy particle comes into contact with an electronic component, the particle has the potential to deposit charge within the device. Depending on the location of the particle strike an immediate effect can be observed on the circuit, called a single event effect (SEE). There are many types of SEEs: single event upset (SEU), single event transient (SET), single event functional interrupt (SEFI) and others. The listed SEEs are pertinent to this work and summarized below. Other types of SEEs are described in detail in [27].

2.1.1 Single Event Upsets

SEUs represent a change in a memory cell in the circuit. For example, if the content of the memory cell is a logic ‘0’, an SEU would change that to a logic ‘1’, as shown in Figure 2.1a. Or,
if the content of a memory cell is a logic ‘1’, an SEU would change that to a logic ‘0’, as shown in Figure 2.1b. When an SEU only changes one memory cell, it is referred to as a single bit upset (SBU) or a single cell upset (SCU)\(^1\).

When an SEU affects multiple memory cells, it is called a multi-bit upset (MBU). Since an MBU is an SEU, it is caused by a single particle strike. An MBU event can affect any number of multiple cells, such as two cells, three cells, four cells, etc. A single MBU event is not limited to just changing logic ‘0’\(^{s}\) to ‘1’\(^{s}\) or vice-versa, but can include both kinds of logic flips, as shown in Figure 2.2.

Similar to an MBU is a multi-cell upset (MCU). The difference between an MBU and a MCU is subtle, and lies in the underlying architecture of the memory. An MCU occurs when two physically adjacent bits are upset from the same particle strike. An MBU occurs when two logically adjacent bits are upset from the same particle strike. When the logical and physical mapping are not the same, then an MCU and a MBU can be different. Assuming that Figure 2.2 shows the

---

\(^1\)An SBU and an SCU can be used interchangeably. Through the remainder of this work, SBU will be used.
Figure 2.2: Example of an MBU Event

physical layout of a memory, then that event is an MCU. But the logical readout of the erroneous bits could be

- Word: 42, Bit: 30
- Word: 43, Bit: 31,

which would not be an MBU because the upsets were in different words. If, however, the logical readout of the erroneous bits was

- Word: 42, Bit: 30
- Word: 42, Bit: 31,

then this event would also be an MBU [28].

SEUs can occur from latched SETs, as discussed in the next section, or can occur directly in the memory cell. An SRAM memory cell is generally implemented as two cross-coupled inverters, as shown in Figure 2.3 which has a basic six transistor layout, as shown in Figure 2.4. The inverters drive each other, with one inverter driving the stored value and the other inverter driving the negated value. If a transistor that is used in the inverters is struck by an ionizing particle that causes it to erroneously turn on or off, it can cause the SRAM circuit to flip values. Since there are four transistors that pertain to the inverters in a six transistor SRAM cell, 2/3 of the transistors are sensitive and can cause SEUs [29].
2.1.2 Single Event Transients

SETs cause a voltage change on a node in the circuit, such as shown in Figure 2.5. SETs can either cause a positive voltage change, as in Figure 2.5a, or a negative voltage change, as in Figure 2.5b. SETs can either dissipate quickly or slowly as shown in the figures, and can occur as either a positive or negative voltage change.

SETs can propagate all the way to the output of the circuit, but generally in digital circuits the SET either dissipates without causing an error or becomes latched into a memory element causing an SEU. In order to cause an SEU, the SET must occur on a sensitive node and must change the logical value. Because an SET is inherently analog, to change the digital value, it must change the voltage such that it is perceived as a different logic value. If the digital value is already a logic ‘1’ and the SET causes a higher voltage, this will not change the logical value. Similarly, if the digital value is already a logic ‘0’, an SET that causes a lower voltage will not change the logical value.
Figure 2.5: Example of an SET from [1]
Figure 2.6: Standard D Flip-Flop

For example, there are several nodes that could cause an SEU in a standard D flip-flop (DFF) shown in Figure 2.6:

- **D (Data) Input** - Transients on the D input only become SEUs if the transient persists to the write period, that is, if the DFF is enabled and the correct clock edge has arrived.

- **En (Enable) Input** - Transients on the enable line can create SEUs if the transient introduces a glitch when a clock edge arrives and the newly latched value is different than the previously stored value.

- **S/R (Set/Reset) Input** - Transients on the S/R line can cause SEUs if the S/R changes the stored value in the DFF. If the S/R is synchronous, then there also has to be a clock edge to cause an SEU.

- **CLK (Clock) Input** - Transients on the clock line can cause false edges which can cause new values to latch into the DFF. These erroneous edges can also cause metastability if the value on the D input is still changing when the false clock edge occurs.

- **Q Output** - Transients on the Q output are fundamentally different than transients on the inputs. SETs on the Q output will only cause SEUs if there are memory cells downstream from the Q output, such as shown in Figure 2.7. In this example, the Q output feeds into another DFF D input after going through some combinational logic. If the transient propagates through the logic, it will appear as a transient on the D input. Similarly, the Q output could
pass through any amount of combinational logic, or none, into a different input of the DFF and cause an SEU.

SETs can also affect other memory cells in the circuit and cause SEUs [1], such as the block RAMs on an FPGA.

2.1.3 Single Event Functional Interrupt

SEFIs are events that cause more global/chip-wide effects. Where SEUs and SETs affect a few cells or nodes in a localized region, a SEFI is a more catastrophic event. For example, if an ionizing particle struck the power circuitry causing a power glitch which in turn causes a chip reset, this type of event would be a SEFI. Naturally, SEFIs are chip specific, depending on what circuitry is on the chip and how that circuitry would globally affect the chip [30].

2.2 Single Event Effects in FPGAs

Since FPGAs are electronic circuits, they are subject to all SEEs. How sensitive an FPGA is to any SEE is device specific and depends on characteristics such as the technology node, fabrication process, device generation/family, etc. While all effects from SEEs are important, SEUs are
particularly important for FPGAs because of their large configuration memory. This section will briefly detail the effects of SEUs on FPGAs and then outline the effects of other SEEs on FPGAs.

2.2.1 Single Event Upsets in FPGAs

SEUs are particularly important to SRAM FPGAs because of their large configuration memory. The newer FPGA devices contain nearly 1 Gb of configuration memory [31]. The more memory there is on the device, the more frequently SEUs will occur across devices in the same generation and family.

Since the contents of the configuration memory define the current function of the device, SEUs within the configuration memory can change the current circuit implemented on the FPGA. Changes in the LUT contents of the device will change the functionality of the device under certain input conditions. For example, the contents of a LUT could define the circuit shown in Figure 2.8, which defines a four input circuit of two AND gates followed by an XOR gate. If an SEU occurs in this LUT, it could change the XOR gate to an OR gate as shown in Figure 2.9. This means under certain input conditions, all ‘1’s in this case, the circuit generates an incorrect output.

SEUs within the configuration memory can also affect how the design is routed. While SEUs within the routing network do not alter the design, they can disrupt the electrical properties of the circuit. Fundamentally, an SEU in a routing bit can either connect two wires or disconnect them, depending on how it was set prior to the SEU. This can disrupt the circuit by causing opens, shorts, antennas and ghosts as described below and in Figure 2.10.

![Figure 2.8: LUT Contents Define Circuits](image_url)
• **Open** - The SEU occurs on a set routing bit, i.e., a configuration bit that is currently being used to connect to wires in the device. The SEU resets the configuration bit so that the two wires are no longer connected. This creates an undriven net in the design, which can lead to an error if the net changes, or fails to change, leading to an incorrect logical value being driven to a downstream cell.

• **Short** - The SEU occurs on an unset routing bit, or a configuration bit that is not currently being used. The SEU sets the configuration bit and two wires on the device become connected. In a shorting situation, both wires are used by two different nets in the design and this SEU creates a connection between both nets. This results in a short between the two nets in the design. A short can change the logical value on either net.

• **Antenna** - The SEU occurs on an unset routing bit, like a short. Instead of connecting two wires actively used by different nets, one wire is used by a net while the other wire is unused. This creates an antenna on the net. While this will not change the logical value of the net, it can negatively affect the timing, due to the extra capacitance of the additional wire.

• **Ghost** - Again the SEU occurs on an unset routing bit, like a short or an antenna, but both wires are unused. This does not directly affect the rest of the circuit, but can be the start of a “ghost” circuit. A ghost circuit can cause the device to consume above average power, but should not affect the correct logical operation.

Since routing bits constitute a significant portion of the configuration memory, most of the design failures occur in the routing network [32].

Figure 2.9: SEUs Change the Design Implemented on an FPGA
SEUs can also affect normal memory elements within the circuit, such as the RAM and flip-flops. This can cause circuit failure if the value in the memory element is used before it is corrected. While these types of failures are important, they occur much less frequently than SEUs within the configuration memory. This is because there is significantly more configuration memory than there is user memory on the device.

2.2.2 Other SEEs in FPGAs

Like all electronic circuits, FPGAs are subject to all SEEs. Characterizing the response of the device to these other SEEs is just as important as characterizing the SEU response. While the main objective of this work is to mitigate the effects of SEUs on FPGAs, previous and ongoing works are underway to characterize other SEE events on FPGAs [33–36].
2.3 Measuring the Reliability of FPGA Circuits

Before analyzing the improvement of TMR or any of the mitigation techniques proposed in this work, the reliability of a design with no mitigation needs to be measured. The reliability testing of the unmitigated design provides a baseline which is needed to show the improvement offered by any type of mitigation technique. There are a variety of testing techniques used to measure the reliability of devices used in high-upset, radioactive environments. These techniques can be used to measure the radiation response of the device and also to validate that a device/circuit has a high likelihood of surviving in an intended environment. Radiation testing is the most common technique to use. Another technique is fault-injection, which is less common, but easier to perform. These are the two most common techniques, and are used throughout this work to measure the effectiveness of the proposed mitigation techniques. This section details how these testing techniques are performed and their relative strengths and weaknesses.

2.3.1 Radiation Testing

One way to measure a circuit’s response to a radiation environment is to expose it to a radiation source, called radiation testing. For example, if a circuit’s radiation response in the terrestrial environment is desired, it can be exposed to a neutron source, as shown in Figure 2.11. While other types of radiation testing exist, neutron testing is used in this work [37].

Radiation testing has two main benefits: it simulates the actual effects seen in the deployed environment and it can be performed at an accelerated rate. The first benefit is the most important. Subjecting a device to the same environment it will be deployed in will inform the designers of the possible radiation effects and how robust the device is at handling them. However, the device needs to be tested for a sufficient amount of time or fluence to collect statistically significant data to be able to accurately estimate how the chip will behave once deployed. Using an accelerated environment helps with collecting this data.

For certain systems, testing at an accelerated rate can also be a drawback. This is especially true of systems that contain a repair element. In these systems, the overall system failure rate depends on the ratio of the repair rate to the radiation flux rate. In an accelerated environment, the radiation flux rate scales up many orders of magnitude, but the repair element may not be able
to. This can lead to a higher system failure rates than what would actually be observed in a real environment.

Another drawback to radiation testing is the relative inaccessibility to testing facilities. Depending on the type of particle used for testing, there is a high demand and only a few facilities available. Despite this, the main drawback is not having the facility in the lab to perform on-demand testing. Radiation tests must be scheduled and planned extensively before execution, making every minute of testing valuable. The need for more on-demand testing for quick feedback has driven the exploration of other testing techniques. Despite these drawbacks, radiation testing is still the best method for proving electronics in a radiation environment.

2.3.2 Fault Injection

Another way to measure the SEU radiation response of a circuit is to use a technique called fault injection. In fault injection, memory cells in the circuit are randomly selected, their values changed and then the outputs of the circuit are monitored for any failures, as shown in Figure 2.12. There are two types of fault injection: random fault injection and targeted fault injection. As
suggested by their names, in random fault injection a bit is randomly selected for fault injection, while in targeted fault injection, memory bits are selected by the user for fault injection [38–40].

The results of a random fault injection reveal the bit sensitivity of a circuit. The bit sensitivity shows the likelihood of a circuit failing from a random upset in the configuration memory. This simulates one of the effects observed during radiation testing. This makes fault injection useful in screening a circuit before it is taken to a radiation test, but does not replace the value offered by radiation testing.

2.3.3 Reliability Metrics

Reliability metrics are used to show how mitigation techniques affect a circuit. Throughout this work, multiple reliability metrics will be utilized. The goal of this section is to introduce those reliability metrics and analyze their relative strengths and weaknesses.
Reliability Function $r(t)$

The reliability of the system is defined to be the probability that the system is correctly functioning at a given time $t$. The output of the reliability function is only a probability; it does not guarantee that a system is correctly functioning at any time $t$. Any two systems can be compared at any time $t$ and the reliability function will reveal which system has a higher probability of functioning at that time. Similar to the reliability function is the failure function, which is the inverse of the reliability function, i.e.,

$$F(t) = 1 - r(t).$$

The failure function is the probability that the system is not correctly functioning at the input time $t$. The failure function is used in this work for plotting purposes, as the plots generated using the failure function are typically easier to decipher than when using the reliability function.

An example of a plotted failure function is shown in Figure 2.13, which plots the failure for two systems, the unmitigated (Simplex) and a TMR system. Time is plotted along the x-axis while the probability of failure is plotted along the y-axis. The y-axis is inverted to produce plots that mimic a reliability graph, i.e., northern values on the y-axis are low failure/high reliability while southern values on the y-axis are high failure/low reliability. The x-axis is ticked at values of $1/\lambda$, where $\lambda$ is the failure rate of the system. Plotting based on periods of $1/\lambda$ creates graphs that are general for all systems for any value of $\lambda$.

The graph in the figure shows the benefits of using behind the reliability/failure function metric. This graph shows that for lower values of $t$, the TMR system has a higher reliability than the Simplex circuit, but at higher values of $t$, the TMR system has a lower reliability than the Simplex system. If these are the two systems under consideration, than for short mission times, the TMR system would probably be better, but for longer mission times, the Simplex system would probably be better.

The reliability/failure function is useful as it provides the details for how the reliability changes over time and can be used to grade systems for certain periods of time that are of interest for particular missions. However, it is difficult to assess which system is generally better as one
may be better for some $t$, but worse at other $t$. This motivates the need for a single number that can be used to compare two systems.

**Mean Time to Failure (MTTF)**

The MTTF metric represents the average amount of time it takes for the system to fail. The MTTF is the integral of the reliability function,

$$MTTF = \int_{t=0}^{\infty} r(t)dt.$$  

It is a useful metric since it is a single number which can easily be used to compare against other systems. One of the downfalls of MTTF is that it does not convey as much information as the reliability function. Returning to the previous comparison of the TMR and Simplex systems, if

Figure 2.13: Reliability of Simplex and TMR Systems
reliability is more important early in the mission, it would be acceptable to permit a lower MTTF in exchange for a higher reliability earlier in the mission. This is what the TMR system accomplishes. Its MTTF is $\frac{5}{6\lambda}$, while the Simplex system’s MTTF is $\frac{1}{\lambda}$. By simply comparing the MTTF of two systems, it would be impossible to determine which system would be better suited for a particular mission.

**Cross-Section**

Another commonly used metric is the cross-section. For radiation testing, it is common to report results in terms of the cross-section, which is failures divided by fluence,

$$\sigma = \frac{\text{failures}}{\text{fluence}},$$

and has units of cm$^2$. This metric is an average of how many particles per cm$^2$ it takes to cause a failure. Thus, for a specified number of particles, $\sigma$ provides the expected number of failures for the system. This metric is analogous to the MTTF, in that it is an average, except it is specified in terms of particles instead of time. If the flux, or the rate of particles per unit time is known, then the cross-section can easily be transformed into the MTTF,

$$\text{MTTF} = \sigma \times \text{flux}.$$  

**FIT**

Another common metric is the failures in time (FIT) rate calculated for sea-level. The FIT is the number of failures for billion hours of operation. Sea-level is the radiation environment which specifies the flux, type of particles, and the energy distribution of those particles. This work uses the JESD89A standard [41], which specifies a neutron sea-level flux of 13 n/cm$^2$h. All neutron radiation measurements were made at the Los Alamos Neutron Science Center (LANSCE) for this work, where the neutron energy distribution is approximately the same as the natural terrestrial neutron distribution [42]. The calculation from cross-section to FIT is straightforward,

$$\text{FIT} = \sigma \times 13 \times 10^9,$$
where $\sigma$ is the calculated terrestrial cross-section. Like the cross-section, FIT is similar to the MTTF. It is the average number of failures over a billion hours of operation.

Confidence Intervals

Since all of these measurements are probabilistic in nature, standard confidence intervals are also presented. Typically 95% confidence intervals are used. These intervals represent a range based on the measured value. There is a 95% likelihood that the true value lies within this range. The tighter the intervals, the more confidence that the true value is relatively close to the experimentally measured value. Confidence intervals are calculated based on the methods presented in [43] and are based on the number of events observed. The more events observed, the tighter the intervals.

When more than 50 events are observed, the standard 2 sigma standard deviation can be used, that is,

$$2\sigma = \frac{2\sqrt{\text{events}}}{\text{experiments}}.$$ 

$\sigma$ in this equation refers to the standard deviation and should not be confused with the cross-section. The number of events would be the number of failures observed for either fault injection or radiation testing. The “experiments” would either be the total number of bits tested for random fault injection or the total fluence for radiation testing. The 2 sigma error can be used to calculate both the lower and upper confidence bounds, that is,

$$\text{bound} = \text{value} \pm 2\sigma.$$ 

In this case the bound would either be the upper or lower bound, and the value is the experimentally measured value, either the sensitivity for fault injection or the cross-section for radiation testing. If fewer than 50 events are observed, than the lower and upper bounds must be calculated separately. This can be done by using the tables in Figure 22 from [43].
2.3.4 Reliability of the Circuit with No Mitigation

Four different circuits are used in this dissertation to analyze the improvements offered by TMR and the mitigation techniques proposed later in the body of this text, the b13, md5, sha3 and aes128:

- **b13** - a simple finite state machine for a weather balloon. It is part of the ITC’99 benchmark suite [44]. In the instantiation used in this work, there are 256 copies that run in parallel and are reduced to a single output vector. The reduction network reduces the outputs two at a time until only a single output vector remains. The reduction network is designed in such a way that if any of the 256 copies fails, the reduction network will corrupt the output vector so that the failure is detected on the golden device;

- **md5** - a message-digest algorithm, used to calculate a checksum to verify data integrity. This circuit takes 512-bit blocks as input to produce a 128-bit hash value. This version instantiates three copies which are chained together in a series;

- **sha3** - secure hash algorithm 3. This version instantiates two copies chained together in a series. In this design the data is “absorbed” and the result is “squeezed” out, meaning that the input is xor’ed into a subset of the state. The output is then “squeezed” or read from the output blocks from the same subset of the state;

- **aes128** - advanced encryption standard. This version operates with a key size of 128 bits and instantiates three different copies, which are chained together in series. Since the key size is 128 bits, there are ten transformation rounds in the design.

The base implementation details of each circuit are shown in Table 2.1. All of these circuits were first tested using fault injection and the results from those tests are shown in Table 2.2.

Due to limited beam time, only the b13 circuit was tested using neutron radiation and the results of that experiment are shown in Table 2.3.

With only the unmitigated circuits tested, these results have little meaning other than showing how sensitive these circuits are to single bit upsets, or in the case of the b13 circuit, how sensitive it is to neutron radiation. The next chapter will introduce and explain TMR. Then these results will be analyzed with the results of the TMR versions of these circuits for both fault injection and
Table 2.1: Base Circuit Details

<table>
<thead>
<tr>
<th>Metric/Circuit</th>
<th># Cells</th>
<th># Routing Nodes</th>
<th># Sites</th>
<th># Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>25,388</td>
<td>226,748</td>
<td>4,269</td>
<td>256</td>
</tr>
<tr>
<td>md5</td>
<td>65,561</td>
<td>701,011</td>
<td>7,872</td>
<td>3</td>
</tr>
<tr>
<td>sha3</td>
<td>20,690</td>
<td>399,059</td>
<td>3,452</td>
<td>2</td>
</tr>
<tr>
<td>aes128</td>
<td>56,090</td>
<td>627,345</td>
<td>9,531</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2.2: Unmitigated Fault Injection Results

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Bit Sensitivity</th>
<th>+95% Confidence</th>
<th>-95% Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>45,542</td>
<td>617</td>
<td>$1.35 \times 10^{-2}$</td>
<td>$1.46 \times 10^{-2}$</td>
<td>$1.25 \times 10^{-2}$</td>
</tr>
<tr>
<td>md5</td>
<td>2,000,000</td>
<td>129,677</td>
<td>$6.48 \times 10^{-2}$</td>
<td>$6.52 \times 10^{-2}$</td>
<td>$6.45 \times 10^{-2}$</td>
</tr>
<tr>
<td>sha3</td>
<td>2,000,000</td>
<td>78,376</td>
<td>$3.92 \times 10^{-2}$</td>
<td>$3.95 \times 10^{-2}$</td>
<td>$3.89 \times 10^{-2}$</td>
</tr>
<tr>
<td>aes128</td>
<td>2,000,000</td>
<td>121,780</td>
<td>$6.09 \times 10^{-2}$</td>
<td>$6.12 \times 10^{-2}$</td>
<td>$6.05 \times 10^{-2}$</td>
</tr>
</tbody>
</table>

Table 2.3: Neutron Radiation Testing Results for No Mitigation of b13 Circuit

<table>
<thead>
<tr>
<th>TMR Type</th>
<th>Fluence (n/cm$^2$)</th>
<th>Number of Failures</th>
<th>Cross-Section (cm$^2$)</th>
<th>+95% Confidence</th>
<th>-95% Confidence</th>
<th>FIT Sea-Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>$3.19 \times 10^{11}$</td>
<td>555</td>
<td>$1.74 \times 10^{-9}$</td>
<td>$1.89 \times 10^{-9}$</td>
<td>$1.59 \times 10^{-9}$</td>
<td>$2.26 \times 10^1$</td>
</tr>
</tbody>
</table>

Radiation testing. Analyzing the unmitigated results in this context will provide more meaningful conclusions, particularly how much improvement the TMR mitigation technique provides.
CHAPTER 3. IMPLEMENTATION OF TMR ON FPGAS

Before applying any of the proposed mitigation techniques of this work, it is necessary to ensure that TMR has been properly applied to the circuit. When TMR is not properly applied, it can lead to failures that occur more frequently than the failures addressed by this work. This motivates the need to understand how to properly apply TMR to FPGA circuits, which is the goal of this chapter. This chapter will explore the techniques used by the state of the art TMR tools to achieve maximum reliability as well as exploring the currently available TMR tools.

3.1 Basic TMR Insertion

The first step in the TMR insertion process is the actual triplication of the circuit. The most common way to perform triplication is to take the original circuit and replicate it to produce three separate module circuits. The outputs of the three modules then drive the majority voter(s) which only propagates the majority value from the three modules, as shown in Figure 3.1. The application

![Diagram of basic TMR insertion](image)

Figure 3.1: Triple Modular Redundancy on an FPGA
of TMR will mask any single module failure. For example, if module 2 fails, then the outputs from modules 1 and 3 will outvote/mask any errors in the output stream from module 2. As soon as any two of the three modules fail, then the system has failed. Returning to the previous example, if module 2 has failed and then module 3 also fails, then the corrupted outputs from modules 2 and 3 can outvote the correct output from module 1. When the system has failed the output should not be trusted as it could be corrupted.

For an FPGA, basic TMR can be applied by triplicating the base circuit’s logic and memory resources. Once this is complete, the outputs of each of the triplicated modules can be routed through a voting network and the output of that voting network is routed to the FPGA’s output ports. TMR that is applied at the circuit level as just described is referred to as coarse grain TMR.

By applying TMR, the designer is increasing the circuit footprint by at least three times as each of the redundant circuit modules uses the same amount of logic and memory resources as the original circuit. The designer also has to account for the additional resources used by the voting network. In coarse grain TMR, the voting network is rather small, but in more sophisticated implementations of TMR, the voting network can be quite large. By increasing the circuit footprint, the designer has also increased the likelihood of something on the chip failing.

Each of the redundant circuits has the same failure rate as the original circuit. For example, if the original circuit has an MTTF of six days, than each of the redundant modules in the triplicated circuit would also have an MTTF of six days. The MTTF of the TMR circuit is not as simple as multiplying the MTTF of the original circuit by three. Recall that the TMR circuit only functions when at least two of the three redundant modules are functioning correctly. The TMR circuit fails as soon as two or more of the redundant modules fail. For example, if the first module fails on day 1 of operation, the second module fails on day 5 of operation and the third module fails on day 18 of operation, then the TMR circuit fails on day 5 of operation because that is when two of the modules have failed.

The notion that the TMR circuit is larger and that there are multiple module circuits that can fail can lead to some counter-intuitive conclusions. One of these counter-intuitive conclusions is that the MTTF of TMR is less than the MTTF of a circuit without TMR. The mathematical
theory behind TMR shows that

\[ MTTF_{\text{TMR}} = \frac{5}{6} MTTF_{\text{NO-TMR}}, \]

and that is assuming that the voting network cannot fail. If the goal of applying TMR was to increase the average circuit lifespan, then the designer failed. TMR alone does not increase a circuit’s MTTF.

Employing a repair element with TMR can increase the average circuit lifespan. Repairing a circuit refers to restoring any failed module within the TMR circuit to a correct, functioning state. As long as any failed module is repaired before another module fails, the TMR circuit will continue correct operation. Returning to the previous example, if the first module fails on day 1, but is repaired on day 2, then when the second module fails on day 5, the TMR circuit does not fail because only the second module is in a failed state on day 5.

The MTTF of the TMR circuit with repair has the potential to be much larger than the MTTF of the circuit without TMR. In fact, the MTTF of the TMR with repair circuit increases as the repair rate becomes larger with the respect to the original circuit’s failure rate. The MTTF approaches infinity as the repair rate approaches infinity, i.e., failed modules are repaired instantaneously. Figure 3.2 shows this trend. As the repair rate (\( \mu \)) increases, so does the reliability and the MTTF\(^1\).

For an FPGA circuit, adding repair has two steps, repairing the device configuration memory and repairing the dynamic memory elements within the circuit. The device configuration memory can be repaired through scrubbing the configuration memory of the device. This means correcting any errors that may be present in the SRAM memory that defines the device configuration. The dynamic memory elements can be repaired by implementing a strategy called feedback TMR, where TMR voters are placed on all of the feedback paths present in the circuit. Both of these repair elements are explored in detail later in this chapter.

\(^1\)More details on the theories and mathematical derivations of these and related TMR systems can be found in Appendix C
3.1.1 TMR Insertion Strategies

TMR can be inserted into a circuit by using either manual or automated methods. In practice, automated methods are generally applied to triplicate most of the circuit while manual methods are employed to ensure proper triplication of the more specialized components, such as the memories and synchronizers.

Manual TMR Insertion

Manual TMR insertion is typically performed by modifying the HDL to include triplication of each component and connecting the wires together. This triplication can be done at any granularity available to the designer in the code. For example, if there is a multiplier in the circuit, the
designer can choose to place three multipliers in the circuit, or can choose to go into the multiplier and triplicate it.

Manual TMR insertion is not usually preferred for most circuits. In large circuits with many hundreds of thousands of HDL code lines, it is easy to miss triplicating some components or to incorrectly wire components together which in turn increases the design time process. Additionally, each circuit is typically verified before adding TMR and should be verified again after inserting TMR. After all this work, the tool can easily optimize away the redundancy during the synthesis optimization step. To avoid this, the optimizations can be turned off, but this prevents the tool from performing many of the good optimizations which can lead to worse performance.

Automated Netlist TMR Insertion

TMR can be automatically added to the circuit post-synthesis using the graphical netlist\(^2\). After the synthesis step, the circuit is converted from HDL into the device specific primitives that can be placed on the device. The netlist also contains information about how these primitives are connected. The primitives represent nodes on the graph and the edges of the graph represent how the primitives are connected. TMR is inserted into the netlist by replicating the graph and adding voters.

There are several flavors of automated TMR; coarse-grain, before flip-flop and feedback are all popular varieties. These different schemes refer to where voters are inserted in the graph. In coarse-grain, voters are only inserted at the end of the graph, which represents the traditional TMR model. Before flip-flop inserts voters before every flip-flop. Feedback TMR, described in the repair section, inserts voters on just the feedback paths of the circuit, instead of before every flip-flop.

There are several tools available for performing netlist TMR. There are three commercial tools, Precision Hi-Rel by Mentor [45], Synopsys Synplify [46] and XTMR [47,48]. There is little information publicly available about these tools, but they all can perform some level of fine grain TMR. There is also little information available on how effective each of these tools is, but it is safe to assume that they provide some significant level of reliability improvement.

\(^2\)Appendix A contains more information on the general FPGA CAD flow.
An open source tool is also available, called the BL-TMR tool [49]. This tool was developed by BYU and Los Alamos National Laboratories. The tool takes the circuit netlist as input and outputs a netlist with TMR applied. By default, the tool applies feedback TMR, or places voters along every feedback path in the circuit as well as triplicating every component in the circuit. The user can also optionally input a list of cells for the tool to ignore and not triplicate. This is the tool used in this work.

Automated TMR Insertion Through HLS

A new branch of TMR research has begun investigating how to apply TMR during the high level synthesis (HLS) process [50]. In HLS, the user writes software-like code, which is then compiled down into an HDL language. With the HDL code, the design implementation flow continues as normal. When TMR is applied during the HLS process the output HDL code contains the triplication. This leads to similar problems encountered in manual TMR insertion, specifically, that the tool can optimize away the redundancy. This problem can be overcome by turning off optimizations, but more research is needed to compare how effective HLS TMR is to netlist-based TMR insertion techniques.

3.2 Techniques to Increase TMR Effectiveness

The discussions surrounding implementing TMR on FPGAs generally focus on the abstract details, such as just implementing the circuit three times, inserting a majority voter and performing configuration scrubbing. These details describe how TMR is implemented at the high level, but leave out many of the details necessary to properly apply TMR. These details range from properly implementing the voting network, the steps necessary for repair, BRAM triplication, etc. Properly applying TMR prepares the circuit for the additional mitigation strategies that will be presented in the later chapters of this dissertation.

3.2.1 Circuit Triplication and Partitioning

When the circuit is triplicated through simple replication as described at the beginning of this chapter it is referred to as coarse grain TMR. Instead of applying TMR at a coarse level, it can
be applied at a finer granularity, called partitioning or more frequent voting [51]. In partitioning, the system is first decomposed into sub-systems, where each sub-system performs some task of the system, as shown in Figure 3.3. The sub-systems are chained together to form the system. Partitioning occurs when TMR is applied to a decomposed system. Each sub-system is triplicated and voting occurs on the outputs of each sub-system before being propagated to the next sub-system, as shown in Figure 3.4. Each triplicated sub-system is referred to as a partition and the voting mechanism after each partition is referred to as a voting group. If the original system is decomposed into \( k \) sub-systems, then after TMR, the system will have \( k \) partitions and \( 3k \) sub-modules, with each partition containing three sub-modules.

With partitioning, the system can maintain correct operation even with multiple sub-module failures, under most conditions. In the normal TMR system, as soon as failures occur in multiple modules, the system will fail. With partitioning, the system will not fail as long as the failures occur in different partitions, as shown in Figure 3.5. In this hypothetical system there are three partitions. The system experiences three failures, one in each TMR domain. The system maintains correct operation because the failures have occurred in different partitions. However, as soon as multiple failures occur in multiple sub-modules in the same partition, the system will fail, as shown in Figure 3.6. In this example, the system has failed because there have been two sub-module failures in partition 2, one in sub-module 0 and the other in sub-module 1.

---

3For simplicity only one output is shown for each partition. In the full system, all of the sub-system outputs would be triplicated using the same method as shown in the figure.
Figure 3.4: Partitioning - Applying TMR to a Decomposed System

Figure 3.5: Multiple Failures in a Partitioned TMR System
Other variations of fine-grain TMR exist, called local TMR (LTMR), distributed TMR (DTMR), block TMR (BTMR) and global TMR (GTMR) [52]. In LTMR, every flip-flop in the design is triplicated and a voter is placed after every flip-flop. This technique is effective for anti-fuse and flash FPGAs, but is not effective for SRAM FPGAs. In DTMR, the entire design, except for global signals, is triplicated and voters are placed after every flip-flop. This is very similar to feedback TMR (described in the repair section), except in feedback TMR voters are only placed on feedback paths after every flip-flop. In BTMR, the design is decomposed into blocks and voters are placed on the outputs of the blocks. In GTMR, the global signals are triplicated.

3.2.2 Voter Triplication

One of the first decisions the TMR designer needs to make is whether or not to triplicate the TMR voters. By not triplicating the voter, the designer introduces a single point failure into the circuit, as shown in Figure 3.7. The outputs from all three of the TMR modules could be correct, but a failure in the voter itself could corrupt the output stream. To minimize the chance of a failure in the voter corrupting the system, the voter should be triplicated.

As the number of partitions grows, so too does the number of voters. The importance of triplicating the voters increases as the number of voters increases since there is a higher probability that a failure will occur in the voting network. On an FPGA, a voter will be implemented as a LUT3, that is a LUT with three inputs from the three different TMR domains. That corresponds to just eight bits. Those eight bits are small in comparison to the large configuration memory,
but another eight bits are added each time a voter is inserted into the design. In addition to the eight bit LUT, a single voter also introduces a single output routing net. This routing net also uses configuration bits that could cause failure. Most of the state of the art tools always triplicate the voter to avoid the single point failures introduced by a single voter.

3.2.3 Repair

Theoretical models show that the MTTF of TMR is actually worse than the unmitigated system. In order to realize the full benefits of TMR and increase the MTTF above that of the unmitigated system a repair element is needed. There are multiple steps necessary to implement repair on an FPGA. The first and most recognizable step is configuration scrubbing to correct errors in the configuration memory of the device. The second step is to add correction mechanisms for the dynamic memory elements in the circuit. This includes the flip-flops, BRAMs, LUTRAMs, SRLs, etc.

Configuration Scrubbing

The first step to implementing repair on an FPGA system is to implement configuration scrubbing. During configuration scrubbing all errors within the configuration memory are corrected. Configuration scrubbing will only correct the static components of the circuit, e.g., mainly the LUT contents and the routing. For example, configuration scrubbing will restore a LUT to its proper functionality, such as restoring an XOR gate when an SEU changed it to an OR gate,
but configuration scrubbing will not correct any errors in the circuit’s internal state that may have resulted from an SEU.

There are many ways to perform configuration scrubbing. One method is to store a golden copy of the configuration memory in a medium where it is much less susceptible to SEUs, such as a rad-hard flash memory. The golden copy can then be used to fix any errors within the configuration memory.

Another method to perform configuration scrubbing is to employ error correction codes (ECC). ECC bits are typically added to each word in the configuration memory. The ECC can be periodically checked for errors, and any discovered errors are then corrected. Similarly a cyclic redundancy check (CRC) could also be employed. The CRC computes a hash for the device memory. If the computed hash is different than the golden hash, a full scrub can be triggered [53–55].

**Feedback Repair for Flip-Flops**

The circuit state can become corrupted anytime an SEU occurs. For example, a TMR incrementer circuit could be composed of three separate incrementers, which then compare and propagate the majority value, as shown in Figure 3.8. During operation, one of the incrementers experiences a stall failure, where it does not increment for a few clock cycles. The other incrementers continue on and successfully propagate the correct value to the output. When the configuration scrubber corrects the circuit, the third incrementer resumes incrementing, but because its state is corrupted, it resumes incrementing from an incorrect value, and is still out of sync with the other two incrementers. Clearly, this TMR circuit is still not repaired, despite the configuration memory having been scrubbed.

To repair this circuit additional circuitry is needed to synchronize the state between the different TMR domains. One method to synchronize the redundant modules would be to implement a reset after detecting an error. A reset would bring all of the circuits into a known state, then computing would resume from this known state. While effective, this has the drawback of having to stall the system, implement a reset, lose all unsaved computations, in addition to the added computational cost of having to detect errors.

Another way to synchronize a circuit is to place voters on all of the feedback paths of the circuit. As shown in [56], voters in the feedback paths ensure that the state is synchronized in real-
time among the circuits. Returning to the incrementer example, when the stall error is repaired, the incrementer is passed in the majority value which is then propagated back into the register, as shown in Figure 3.9. In feedback TMR, voters are automatically placed along all of the feedback paths in the circuit. The memory element in the circuit is repaired at the rate at which it is written to. Feedback can be used for memory elements that are written to often, which is generally the case for the circuit’s flip-flops. Memory structures that are infrequently written to need more specialized repair schemes.

**BRAM Triplication and Repair**

The FPGA contains many on-chip memories called block RAMs (BRAM). BRAMs are similar to FFs, containing state information for the circuit. Just like the FFs, the memory in the BRAMs can become corrupted and need to be frequently repaired. There are several options for BRAM repair, but the repair circuitry must be explicitly inserted into the circuit in addition to the normal triplication.

One option for BRAM repair is to enable the ECC for the BRAM. Most BRAMs employ a single error correct, double error detect (SECDED) ECC protection for each BRAM word. This
means that ECC is able to correct any single errors and detect, but not correct, any double errors in each word in the BRAM. In order to apply correction to the word, the word would need to be read from memory, any errors identified by the ECC would be corrected and then the corrected word would be written back into the memory. If a double error is detected, the error cannot be corrected without additional mechanisms, such as fetching the identical word from one of the other TMR domains, and using that word, assuming it is correct, to repair the double error.

Another option for BRAM repair is to employ a BRAM scrubbing mechanism. Each time a word is read from the BRAM it would be read from all three triplicated BRAMs. The redundant words would be voted on and then the majority value would be propagated throughout the circuit as well as being written back into each BRAM. This BRAM scrubbing circuitry would need to be added for each BRAM in the original circuit.

A potential problem with the described ECC and scrubbing techniques is that it relies on the original circuit frequently accessing all of the BRAM contents. If some of the contents are rarely accessed, then errors can accumulate within the memory without being corrected. A better strategy would ensure that all of the memory contents are frequently repaired to minimize the chance of accumulation. With dual ported BRAMs, it is possible to employ the scrubbing mechanism or
ECC correction mechanism in the background without disruption to normal circuit operation. This mechanism would operate using the second port and would continually cycle through all memory addresses. However, if the original circuit utilizes both of the BRAM ports, this strategy would not be possible.

If background repair mechanism is not possible, then the designer needs to decide whether to repair the memory as it is used, or if it is possible to share the ports for the sake of repair. When the memories are not continuously repaired, it increases the probability of an undetected or uncorrectable error occurring [57]. The trade-offs for reliability and performance for each method are circuit dependent, so the circuit designer needs to decide the best course of action for their particular mission.

**Repair for Other Circuit Memory**

Along with the BRAMs, other memory structures in the device need to be properly triplicated and repaired. These structures include LUTRAMs, i.e., LUTs that are configured to act as small memories, and shift register lookups (SRL). These memories are significantly smaller than a BRAM which makes them easier to protect with ECC. The authors in [57] propose a duplication scheme to protect these memories. In this duplication scheme, the memory and control signals are duplicated. The ECC is checked whenever the memory is read. If an error is detected, then a state machine can read the same contents in the duplicated memory and use that to scrub the contents of the corrupted memory. Since the LUTRAM and SRL memories are small, all of their contents should be read/written to regularly, which will correct any errors regularly. If this is not the case for a particular circuit, than a scheme similar to the triplicated BRAMs can be implemented.

**3.2.4 Cross Clock Domain Timing Synchronization**

Another nuance in TMR application is how to apply TMR on signals that cross clock domains. Just like untriplicated data signals, triplicated data signals can cause metastability and other cross clock domain problems if not properly handled. The standard two flip-flop synchronizer will not work for TMR signals, even if this same synchronizer worked for the untriplicated data signal. This is because the triplicated signals behave like a bus rather than a single signal.
The problem with the standard synchronizer arises because of the differences in the data path delays of the triplicated signals coupled with the asynchronous nature of the receivers next clock edge. In a TMR system, this can cause the different domain signals to arrive at different clock cycles in the receiver’s clock domain. Consider the example in Figure 3.10. In case 1, all three domains are functioning correctly, but due to different data path delays, the signal in domain one arrives at the clock cycle prior to domains two and three. Due to the nature of TMR, the voter passes through the signal in this second clock cycle, since two of the three domains agree. Now consider case 2, where domain three has experienced a stuck at “0” failure. Domain one again arrives at the clock cycle before domain two, but since domain three has failed, the voter does not pass through any signal, and the pulse is missed entirely.

In [2], a modified synchronizer is proposed that will safely cross a clock domain. The idea behind the modified synchronizer is to pulse the signal for two clock cycles, as shown in Figure 3.11. Due to data path delays, domain one still arrives one clock cycle prior to the other two domains. All domains are functioning properly in case 1 and a single pulse is generated for two clock cycles in the receiver’s clock domain. In case 2 the third TMR domain has a stuck at “0” failure, but unlike the original synchronizer, the modified synchronizer still functions correctly as domains one and two overlap for at least one clock cycle. This can be done by adding additional circuitry to the standard two flip-flop synchronizer, as shown in Figure 3.12.
Making the proposed modification to the standard synchronizer in the user’s circuit cannot be done blindly. Due to the additional flip-flop in the modification, this will delay the pulse from being detected in the receiver’s clock domain by one clock cycle. This would also affect how rapidly pulses could be communicated between the clock domains. Another issue with this design is that it does not address other synchronization protocols that are used to handle bus communication. This is a known problem in TMR research and still an open research question.

3.3 Limitations of Current TMR Tools

The effectiveness of the BL-TMR tool in improving reliability has been previously studied. Of note are two particular works which study the LEON3 processor. In one work, the author showed a TMR design of the LEON3 achieved a $27,466 \times$ improvement over the unmitigated design in terms of bit sensitivity [58]. In the other study, the TMR LEON3 design only achieved
an improvement $51.3 \times$ over the unmitigated design [22]. The obvious question then becomes, why is there three orders of magnitude difference in improvement between these two studies?

Some of the difference between the two studies can be explained by differences in the testing methodology. One of the major differences is how the two studies classified failures. In [58] failures were classified by either the program not terminating, or terminating with an incorrect result. [22] had a much more strict classification of failures. A failure was classified as any discrepancy on the data bus during any clock cycle. Not all errors on the data bus will translate into failures for the currently running program. Each way of classifying failures is valid, but does create discrepancies when calculating bit sensitivity and improvement rates.

There were several other differences between the two studies that could factor into the different improvement rates: different programs running on the processor during testing; different FPGA generations, a Virtex-4 vs. a Kintex-7; one of the studies included software mitigation as well as hardware TMR; etc. All of these factors are important, but irrelevant when discussing the limitations of TMR. The real concern in both of these studies is that fault injecting a single bit caused TMR failure. In an ideal TMR system, no SBU would cause a circuit failure.

Discussion with the author of [58] revealed surprising details that were not published. The author hand placed and routed parts of the design in order to reduce the single bit sensitivity of the circuit. The authors of [22] let the vendor’s tool handle all placement and routing. This suggests that the sensitivity of a TMR circuit is directly dependent on that circuit’s implementation (placement and routing) on the device. The vendor’s tool, which has no knowledge of the redundancy, sometimes chooses an implementation that is less than optimal for reliability concerns. The effectiveness of current TMR techniques could be improved by providing mitigation for these single bit failures by altering the placement and routing.

A new tool could be built to mitigate single bit failures in TMR circuits, but it is important to first completely and properly apply TMR. Properly applying TMR protects many more bits than this new tool would protect. For example, using the bit sensitivities calculated in [22], the unmitigated design was calculated to have 240,539 total sensitive bits. The TMR design was calculated to have only 4,689 sensitive bits, a difference of 235,850 bits. This means that properly and completely applying TMR protected 235,850 bits. Meanwhile, this new tool would only protect up to
4,689 bits. While these 4,689 bits do limit the effectiveness of TMR, there is no use in protecting them if the 235,850 bits are not first protected.

3.4 Summary

Implementing TMR with repair is an effective technique to mitigate the effects of radiation on SRAM FPGAs, but there are several necessary steps to properly implement TMR with repair. The repair mechanism needs to fix both the configuration memory of the FPGA as well as all of the circuits state stored in dynamic memory. The user also needs to ensure that TMR is properly applied to specialized circuits, such as cross-clock domain synchronizers, to ensure proper circuit functionality. There are also other strategies, such as partitioning, that can allow TMR circuits to tolerate more upsets without causing the circuit to fail.

TMR is typically inserted into the circuit through a combination of automated and manual methods. After TMR is inserted into the circuit, the circuit’s netlist becomes triplicated, but the specific device implementation may not be optimal for circuit reliability. This is evident by fault injection tests which reveal the presence of single bits that can cause circuit failure. These single bit failures represent a limitation in the effectiveness of TMR. The circuit reliability can be improved by studying how the single bit failures occur and then providing mitigation for them.
CHAPTER 4. SINGLE BIT FAILURES IN TMR

As stated in the previous chapter, TMR with repair is an effective technique for increasing the circuit’s reliability, but is limited by the presence of single bit failures. Single bit failures (SBF) are failures in TMR circuits that are caused by a single configuration bit. When an SEU occurs in one of these bits, the circuit immediately fails without a chance for repair. Thus, the presence of SBF in a circuit places a limit on the maximum reliability that cannot be overcome by increasing the repair rate.

The SBF sensitivity for a circuit can be characterized through fault injection. Due to the nature of the algorithm, as explained in Chapter 2, there is only a single fault present in the system at any time. If a failure occurs, then the system failed because of a single bit. Table 4.1 shows the results from a fault injection campaign for several TMR circuits. The improvement over the unmitigated circuits ranges from just $665 \times$ to $730,680 \times$, with a geometric average improvement of $18,235 \times$. Again, the fact that fault injection finds any bits that cause failure, shows a significant limit to current TMR with repair implementations, which has also been shown in other works [3,22, 24]. In order to further improve the effectiveness of TMR, these SBF bits need to be mitigated, but before they can be mitigated they must first be understood. Understanding the architecture-level details on how TMR circuits fail from single configuration bits is the goal of this chapter. This will

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Fault Injections</th>
<th>Number of SBF</th>
<th>Bit Sensitivity</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>2,000,000</td>
<td>28</td>
<td>$1.40 \times 10^{-5}$</td>
<td>968×</td>
</tr>
<tr>
<td>md5</td>
<td>2,000,000</td>
<td>195</td>
<td>$9.75 \times 10^{-5}$</td>
<td>665×</td>
</tr>
<tr>
<td>sha3</td>
<td>12,000,000</td>
<td>2</td>
<td>$1.67 \times 10^{-7}$</td>
<td>235,128×</td>
</tr>
<tr>
<td>aes128</td>
<td>12,000,000</td>
<td>0</td>
<td>$8.33 \times 10^{-8}$</td>
<td>730,680×</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>28,000,000</strong></td>
<td><strong>225</strong></td>
<td><strong>$2.09 \times 10^{-6}$</strong></td>
<td><strong>18,235×</strong></td>
</tr>
</tbody>
</table>
be done by first differentiating between two types of SBF and then analyzing the device-specific architecture affected by these two types of SBF.

4.1 Types of SBF in TMR Systems

There are two different types of SBF in TMR circuits, single point failures (SPF) and common mode failures (CMF). The difference between them is subtle, but important. SPF occurs in systems due to a lack of triplication, while CMF affects two or more TMR modules simultaneously. The distinction is important as SPF can only ever be fully mitigated by completely triplicating everything in the circuit.

4.1.1 Single Point Failures

Single point failures represent any SBF that is caused by a failure in any untriplicated component/resource in the circuit. As an example of this, consider the TMR system presented in Figure 4.1a. In this system, the original module is triplicated completely, but there is only a single voter. If there is a failure in any module, it will still be masked by the other two modules. However, a failure in the voting circuitry would be propagated to the outputs causing a circuit failure.

A similar, but different single point failure is shown in Figure 4.1b. In this example the voters are triplicated, but the inputs to the TMR system are untriplicated. Any single module failure that occurs within the modules or within the voters will be masked by the system, but any errors that occur on the input to the branch point will be propagated to all of the modules and can cause a circuit failure.

Table 4.2 shows the results of a fault injection study on TMR circuits with untriplicated I/O (common-IO), where the I/O pins on the circuit are not triplicated. The range of improvement is much lower than the results presented at the beginning of the chapter for triplicated I/O (trip-IO) TMR circuits. The improvement for the common-IO circuits ranges from $10 \times$ to $1,160 \times$ for an geometric average improvement of $121 \times$. The average improvement is over two orders of magnitude less than that for trip-IO circuits, showing the impact SPF can have on the circuit. In these common-IO circuits, SPF must first be mitigated before mitigating for CMF can even be considered.
The TMR circuit failures from these examples occur because there are untriplicated resources in the system, such as a single I/O pin. The circuit failure does not occur within the redundancy, but outside of the redundancy. In all cases of SPF, the failure originates within the untriplicated circuitry in the system. The only way to completely remove SPF from a system is to
completely triplicate all the circuitry, including the I/O ports and voters. However, as the table at the beginning of the chapter showed, even after complete triplication, there are still other failures.

4.1.2 Common Mode Failures

Common mode failures represent any SBF where the failure originates in the redundancy. Because the failure occurs in the redundancy, this means that a single bit can simultaneously affect multiple TMR domains. CMF does not occur due to a lack of circuit triplication, as was the case for SPF. In CMF, the circuitry that is failing has been completely triplicating, but something in the underlying architecture of the device has mapped multiple instances of TMR circuitry to the same bit.

To envision the difference between SPF and CMF, consider a circuit that employs a single voter. Suppose that a single bit upset occurs in the voter and causes the voter to fail, which in turn causes the circuit to fail. This is an example of SPF, because the failure occurred because the voter was not triplicated. Now consider the same circuit, but it employs a triplicated voter. Suppose that a single bit upset occurs and causes two voters to fail, which in turn causes the circuit to fail. This is an example of CMF. The problem was not that the voter was not triplicated, but in how the voting circuitry was implemented on the device.

4.2 How Single Point Failures Occur in FPGA Circuits

As previously stated, single point failures occur in TMR circuits that are not fully triplicated. This mainly occurs when all of the I/O is not triplicated, as the voting mechanism is almost always triplicated when implemented on FPGAs. One common reason the I/O is not triplicated is because there are not enough physical I/O pins to use on the device. If the original (untriplicated) design utilizes more than 1/3 of the I/O pins, then by definition, there will not be enough I/O pins to completely triplicate a design.

Figure 4.2 shows an example of a TMR circuit with untriplicated I/O on an FPGA. There are five locations in this diagram where an SPF can occur, labeled 1-5. Two of the five locations are the I/O themselves. Two other locations are on the routes leading to/from the I/O. When coming from the input I/O, there is a single route before it eventually splits to feed each individual TMR
domain. Then when going to the output I/O, there is a single route from the reduction voter to the output. The final location is the single reduction voter. Because there is only a single output, the output of the three TMR domains needs to be reduced to a single net for the device output. This is done through the use of a single voter, not the typical triplicated voter used throughout the rest of the circuit.

SPF can also occur when other resources in the circuit are not triplicated, such as when a circuit uses a single resource. Such resources could be the MGTs, DSPs or BSCAN. Like I/O, there are only relatively few of these resources, compared to the relatively large number of LUTs and FFs. Again, like the I/O, if the untriplicated design uses more than 1/3 of these resources, then they cannot all be triplicated.

An example of such a circuit is shown in Figure 4.3. SPF occurs in the middle of these circuits, instead of occurring at the beginning or end of a circuit, as was the case with I/O SPF. There are four locations in the limited resource circuit where SPF can occur: in the single voter, in the route leading to the resource, in the resource and in the route leading from the resource.

Understanding where failures can occur from the netlist perspective helps in assessing failures when they are observed through fault injection or in radiation testing. The next few subsections analyze where failures occur from a variety of different designs. First, the use of single voters is analyzed. Then SPF arising from the design implementation will be analyzed, such as routing SPF and CMF.
4.2.1 Single Voter Failures

Single voter failures occur when the voters are not triplicated. When the voter is not triplicated, failures can occur in each voter. Failures can also occur in the routing network as each voter outputs a single net which then branches out to feed each TMR domain. As the number of partitions grow, so does the voting network, and consequently, the sensitivity of the voting network.

Fault injection can be used to test how using only a single voter for each partition affects the circuit sensitivity. For these tests, only feedback TMR is used, i.e., partitions are only formed on the feedback paths in the circuits. Then only a single voter is used, as shown in Figure 4.4.

The results of this test are shown in Table 4.3\(^1\), which shows the number of SBF in common-IO

---

\(^1\)The md5 was only tested to 37,318 bits since a large number of failures were observed. This was done so that more testing time could be dedicated to circuits where failures were harder to observe.
Table 4.3: Single Voter Failures in TMR Circuits

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Common-IO</th>
<th></th>
<th></th>
<th>Trip-IO</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of</td>
<td>Number of</td>
<td>Number of</td>
<td>Number of</td>
<td>Number of</td>
</tr>
<tr>
<td></td>
<td>Partitions</td>
<td>Fault Injections</td>
<td>SBF</td>
<td>Fault Injections</td>
<td>SBF</td>
</tr>
<tr>
<td>b13</td>
<td>8,448</td>
<td>2,000,000</td>
<td>12,027</td>
<td>2,000,000</td>
<td>14,872</td>
</tr>
<tr>
<td>md5</td>
<td>6,030</td>
<td>37,318</td>
<td>326</td>
<td>2,000,000</td>
<td>12,356</td>
</tr>
<tr>
<td>sha3</td>
<td>3,374</td>
<td>2,000,000</td>
<td>4,513</td>
<td>10,000,000</td>
<td>23,608</td>
</tr>
<tr>
<td>aes128</td>
<td>8</td>
<td>2,000,000</td>
<td>101</td>
<td>10,000,000</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>17,860</td>
<td>6,037,318</td>
<td>16,967</td>
<td>24,000,000</td>
<td>50,840</td>
</tr>
</tbody>
</table>

and trip-IO TMR circuits. As shown in the table, the single voter designs are significantly more sensitive than the triplicated voter designs shown in the previous tables, 4.2 and 4.1. As the number of voters grows, so does the sensitivity. Comparing the data in this table with the fault injection results for the triplicated voter circuits at the beginning of this chapter shows that triplicating the voters automatically yields an average sensitivity improvement of $5 \times$ for common-IO circuits and $231 \times$ for trip-IO circuits.

### 4.2.2 SPF From the Circuit Implementation

The failures in the triplicated voter designs from Tables 4.2 and 4.1 are comprised of both SPF and CMF. Each bit that caused a failure can be analyzed to determine what low-level architecture was affected and hence, how the circuit failed. Using the previous figures (4.2-4.4) as a guideline, there are several types of failures:

- **Routing Failure** - A failure within the routing network. This can occur either from the input pin or going to the output pin from the reduction voter. A routing failure can further be split into a non-clock routing failure or a clock routing failure. It is useful to make this distinction on an FPGA as the routing for non-clock and clock signals are distinct and use separate resources;

- **Voting Failure** - A failure that occurs within the reduction voter that is used to drive a single signal out of the chip. This occurs in the LUT that is used for the voting logic;
Table 4.4: SPF in TMR Circuits

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Routing Failures</th>
<th></th>
<th></th>
<th></th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non-Clock Failures</td>
<td>Clock Failures</td>
<td>Voter Failures</td>
<td>CMF Failures</td>
<td>Other Failures</td>
</tr>
<tr>
<td>b13</td>
<td>1,941</td>
<td>491</td>
<td>0</td>
<td>103</td>
<td>256</td>
</tr>
<tr>
<td>md5</td>
<td>65</td>
<td>197</td>
<td>0</td>
<td>30</td>
<td>83</td>
</tr>
<tr>
<td>sha3</td>
<td>52</td>
<td>13</td>
<td>0</td>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td>aes128</td>
<td>51</td>
<td>13</td>
<td>0</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>Total</td>
<td>2,109</td>
<td>714</td>
<td>0</td>
<td>139</td>
<td>405</td>
</tr>
</tbody>
</table>

- **Other Failure** - Any other form of failure. This would mostly be CMF or other localized configuration failures.

Using these failure types, the data for the common-IO, triplicated voter TMR version from Table 4.2 can be classified, shown in Table 4.4.

These results show that routing failures, both logic and clock, are by far the largest cause of SPF. This is because input nets can have a large fanout making them more susceptible. It is possible to map the location of failures onto the physical design implementation to visualize what part of the design is being affected by the failure. Figure 4.5, for example, shows an input pin that has a large fanout net and red dots that show the location of several failures observed during fault injection. As the figure shows, this particular route is quite sensitive. There are many more nets similar to this one in the circuit, which make the overall design quite sensitive. Similarly, output nets are also sensitive, as shown in the figure.

The reason that high fanout nets are so sensitive, is that any disruption on the net will be propagated downstream. If this disruption occurs on a portion of the net that feeds more than one TMR domain, than the disruption will affect multiple TMR domains, potentially causing a failure. This effect is magnified when the net also feeds multiple cells, increasing the likelihood that a failure will be observed for the given set of input vectors. Figure 4.6 shows an example of a net with multiple sink pins. A disruption in any of the labeled nodes could cause a failure, as it would be propagated to multiple TMR domains. Disruptions higher in the tree are more catastrophic, as these disruptions would be propagated not only to multiple TMR domains, but also to multiple
cells. The more sink pins there are, the more levels there are in the tree, leading to more potential catastrophic failures. These failures can occur on both logic and clock routing nets.

Voter failures occur much less frequently, because there are fewer bits that control the voting logic. Since voters take three input signals (from the three domains), it would be implemented
as a LUT3, or a LUT with three inputs. Only 8 configuration bits are needed to implement a LUT3, meaning there are 8 potential sensitive bits, but since there is only one fault present in the system at a time, only two of the bits matter, the bit for all 0’s and the bit for all 1’s, as these are the only input conditions the LUT will see, if it itself has failed. So the total number of potential voter failures that can occur is 2 times the number of output pins in use. In modern FPGAs that are approaching 1Gb of configuration memory, the configuration bits that could cause voter failures would represent only a tiny fraction of the total configuration memory.

4.3 Common Mode Failures

When the design is completely triplicated, there is no potential for failure from the viewpoint of the circuit netlist. All of the voters would be triplicated, as well as all of the I/O ports and all of the specialized resources. But, the fault injection tests from the beginning of the chapter showed that SBF can still occur, called CMF. These failures do not occur due to incomplete triplication in the circuit, as was the case for SPF, but rather occur in how the design is implemented. In order to properly apply TMR, the TMR designer needs to be aware of the underlying device architecture. Traditional TMR approaches that only alter a design’s netlist are insufficient to resolve CMF.

In order to understand how CMF occurs on the device, it is necessary to understand the low-level architectural details of the FPGA. This next subsection starts at the high-level of how designs are routed, then delves into the specific details on how the configuration memory programs the routing network. This information is then used to show how an upset in a single bit of the FPGA configuration memory can simultaneously affect multiple TMR domains and cause circuit failure.

4.3.1 FPGA Routing Network

The programmable routing network on the FPGA contains many possible connections to accommodate the many different circuits that the device can implement. At each stage in the routing network, a source wire can connect to any number of sink wires, as shown in Figure 4.7, where the source wire A can be routed to B, C, D and/or E. Each possible connection is called a programmable interconnect point (PIP), e.g., from A to B, A to C, etc. A PIP can be turned on or
off. When a PIP is turned off there is no connection from the source to sink wire, but when it is
turned on, there is a connection from the source to sink wire.

There are many source/sink wire connections on the device, similar to the example shown
in Figure 4.7. Groups of similar source/sink wire connections are organized into tiles called routing
switchboxes, as shown in Figure 4.8a. In the figure, on the left side are input/source wires to the
routing switchbox and on the right side are output/sink wires of the switchbox. For example,
source wire A can connect to the sink wires P, S, V and Y. For any given circuit implementation,
the source wire A can connect to any, all or none of these sink wires.

Due to the programmable nature of the FPGA, this example switchbox can have many
different, but valid configurations. An example of a valid switchbox configuration is shown in
Figure 4.8b. In this particular configuration, the source wire A is connected to the sink wire S,
which means that the PIP that connects wire A to wire S is turned on, and the PIPs that connect
wire A to wires P, V and Z are off.

Instead of looking at forward PIP connections from source wire to sink wire, it is possible
to examine reverse PIP connections from sink wire to source wire. Examining PIP connections
from this viewpoint shows which source wire can connect to a sink wire, as shown in Figure 4.9a.
This example shows that source wires A, D, H and K can connect to the sink wire S.
Figure 4.8: Example of a Routing Switchbox with a Valid Configuration

Figure 4.9: Example of How Sink Wires are Driven with an Invalid Configuration
In a valid configuration, only one source wire can connect to a sink wire. The example from Figure 4.8a is a valid configuration because at most one source wire is connected to a sink wire for all of the sink wires in the switchbox. The example in Figure 4.9 is an invalid configuration because there are two source wires, A and K, which are connected to the sink wire S.

PIPs are a useful construct when discussing the routing network as they show all of the possible connections available on an FPGA device. However, PIPs do not necessarily represent how the configuration memory actually programs the device. It would be naïve to assume that there is a one-to-one correspondence between bits in the configuration memory and PIPs in the routing network. In reality, how the configuration memory programs the device is much more complex and device specific. The details that are about to be presented are specific to the Xilinx 7-Series FPGA devices. The applicability of this information and the failure mechanisms to other FPGA devices will be discussed in the conclusion of this dissertation.

A Xilinx patent reveals that the configuration memory interacts with the routing network through a mux structure, called a routing mux or a PIP junction, shown in Figure 4.10a [59]. A routing mux is a special structure which controls which source wire can drive a sink wire. There is one routing mux for each sink wire in a switchbox.

The figure from the patent can be redrawn into a more typical mux format to better visualize what is occurring, shown in Figure 4.10b. The routing mux is programmed by setting exactly two
bits in the mux, one column bit and one row bit. In the example from the figure, the column bits are
the bits labeled 0-4 while the row bits are labeled 5-8. The column bits control which transistors in
the mux drive each row wire and the row bit controls which row wire drives the output (sink) wire.

Figure 4.11 shows an example configuration for a routing mux. As shown at the top of the
figure, there are many possible source wires that connect to the output wire. For the configuration
in the figure, the net marked in red is currently connected to the output wire of the routing mux.
The wires marked in bolded black and green are other nets in the design, but are not currently
connected to the output wire. All other wires are unused in this design. The bottom of the figure
shows how this would be configured in a routing mux. The nets that can connect to the output wire
appear as labels in the routing mux. There are also a number of unused wires that are not being
driven by nets in the circuit which are blank in the routing mux. To route a net, the column and
row bit associated with that net’s location in the routing mux need to be set. In this example, the
net “clk_TMR_1” is located in the first column on the third row, which corresponds to column bit
0 and row bit 7.

4.3.2 Routing CMF

Understanding how the routing mux operates is essential to understanding how routing
CMF occurs in circuits. This section analyzes what could happen if an SEU occurred in a routing
bit for different routing mux configurations. Examples will be used to discuss what happens when
an SEU occurs in a routing mux. For these examples, the base configuration of the routing mux
is the same as the previous sections and shown in Figure 4.12. In the routing mux, an SEU could
occur in one of two places, a column bit or a row bit.

SEUs in an Unused Routing Mux

In an unused routing mux, neither a column bit or a row bit is set. This means that there is
nothing driving the row wires or the output wire of the routing mux. There are, however, still nets
that could drive the mux output wire. This means, that if the right column and row bits were set,
one of the mux source wires would drive the sink wire. For example, in the mux from Figure 4.12,
if column bit 0 and row bit 5 were set, then the net “data_TMR_2” would drive the sink wire.
Figure 4.11: Example Configuration of a Routing Mux
If an SEU were to upset a row bit in an unused routing mux, then one of the row wires would start driving the output wire, as shown in Figure 4.13. But, there is no column bit that is set,
Figure 4.14: Column Bit SEU in an Unused Routing Mux

so there is no active signal driving the row wire. This means that the sink wire is still undriven and there are no shorts in the design.

If the SEU were to instead occur in a column bit, then each row wire would start to be driven, as shown in Figure 4.14. Because there is no set row bit, only the row wires are being driven, and the sink wire of the mux is not being driven. This could potentially disrupt a net in the design by increasing its timing, due to the increased capacitance on the wire. However, such changes in the net timing would be minimal. Since neither an SEU in a row or column bit in an unused mux will affect multiple TMR domains, an unused routing mux cannot cause CMF.

4.3.3 SEUs in a Used Routing Mux

In a used routing mux there is a set column and row bit. Because there is a set column and row bit the mux sink wire is actively driven. Even though there is only a single net that is driving the sink wire, there are many potential nets that could be driving the sink wire. An SEU in a row bit would allow another row wire to drive the sink wire, as shown in Figure 4.15. Since another row wire is already driving the sink wire, this would create a short, if the newly connected row
wire is actively being driven. In the example from the figure, the mux sink wire is already being
driven by the net “clk_TMR_1”, but is shorted to the net “data_TMR_2”.

This could possibly cause a circuit failure since the short involves two different TMR do-
mains. However, electrically, the node will be driven to a single voltage value, which will be interpreted as either a logic ‘1’ or a logic ‘0’. Both signals will either be read as a ‘1’ or a ‘0’. To be driven to either value, there has to be a least one signal that is already driving that value. For example, the node will only be driven to a logic ‘0’ if either clk_TMR_1 or data_TMR_2 is trying to drive a ‘0’ and vice-versa for a logic ‘1’. This means that at most one net will be driven to an incorrect value at any given time instant. Since this only affects one net, this only affects one TMR domain, thus, a row short in a used routing mux will not cause CMF.

An SEU in a column bit would activate another column of transistors, which would create shorts along the row wires, as shown in Figure 4.16. This creates a short at the sink wire, but also creates shorts along all the row wires. Since the mux in the figure has four rows, there is a possibility for up to four different shorts. In the example from the figure there are two different shorts, one between the nets “data_TMR_2” and “clk_TMR_0” and the nets “clk_TMR_1” and
“data_TMR_1”. If both “clk_TMR_0” and “clk_TMR_1” are driven to incorrect values due to the short, then multiple TMR domains are affected which would cause CMF. It is SEUs in column bits in used routing muxes that cause routing CMF.

4.3.4 Clock CMF

The SBF reported for the trip-IO designs at the beginning of this chapter in Table 4.1 will be classified later in this chapter after completing a discussion on all types of CMF. But, looking for patterns within all of the routing CMF detected reveals a surprising trend, in all cases, two different domain clocks are always involved. No routing CMF has been observed involving only data nets or just one of the clocks. Theoretically, there is nothing preventing routing CMF from occurring with other nets. These routing CMF that only involve clocks have been dubbed “clock CMF”.

Figure 4.16: Column Short in a Routing Mux
Clock CMF Locations

Since clock CMF must involve two different domain clocks a natural question that arises is where clock CMF can occur on the FPGA. The Xilinx 7-Series FPGA architecture employs a global clock network separate from the general routing network. This type of architecture limits the locations on the device where two clocks could simultaneously short. These locations occur in the configurable logic block (CLB) routing tile switchbox\(^2\). The CLB routing switchbox then routes into the CLB logic tiles which contains the user defined logic, as shown in Figure 4.17.

The CLB logic tile contains two slices and each one of the slices can only support one clock. Each cell in the slice must operate off of that single clock, so each clocked cell in the slice must pertain to a single domain since the clocks have been triplicated. This means that each CLB logic tile contains two sink wires to support up to two clocks, one for each slice. Two routing muxes in each CLB routing switchbox exist to support these clocks, shown in Figure 4.17.

\(^2\) Additional details about the Xilinx 7-Series architecture can be found in Appendix B.
4.3.5 Site CMF

Another type of CMF, unrelated to clock CMF, has been detected during testing. In the Xilinx architecture, it is possible to map two LUTs of input size five or less onto the same LUT6. When this occurs, there can be at most six unique inputs into the LUT. In a TMR system, this would allow two LUTs from different TMR domains to be placed at the same physical location, i.e., they are placed on the same LUT.

In a SLICEM site, each LUT can be configured as a LUT or as another special memory element. Some of the other configurations include a RAM mode (LUTRAM) and a shift register mode, among others. To accommodate these different modes, these LUTs can also take as input a clock signal, a write enable and a reset line. When configured in a typical LUT mode, these extra inputs are not used.

Some configuration bits are used to define the current mode for a LUT in a SLICEM. During fault injection, it has been observed that when these mode configuration bits change, it also changes the contents of the LUT. Performing a readback after injecting one of these bits reveals that there are many upset bits, all associated with a single LUT. For example, in the MD5 design running on the Artix7-200T FPGA, after injecting the bit in frame 0x000139E, word 55 and bit 17, the following additional bits reported being upset, where the formatting is <frame>:<word>:<bit>:

<table>
<thead>
<tr>
<th>Frame</th>
<th>Word</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00013A0</td>
<td>55</td>
<td>17</td>
</tr>
<tr>
<td>0x00013A0</td>
<td>55</td>
<td>18</td>
</tr>
<tr>
<td>0x00013A0</td>
<td>55</td>
<td>21</td>
</tr>
<tr>
<td>0x00013A0</td>
<td>55</td>
<td>22</td>
</tr>
<tr>
<td>0x00013A0</td>
<td>55</td>
<td>24</td>
</tr>
<tr>
<td>0x00013A0</td>
<td>55</td>
<td>25</td>
</tr>
<tr>
<td>0x00013A0</td>
<td>55</td>
<td>26</td>
</tr>
<tr>
<td>0x00013A0</td>
<td>55</td>
<td>27</td>
</tr>
<tr>
<td>0x00013A1</td>
<td>55</td>
<td>17</td>
</tr>
<tr>
<td>0x00013A1</td>
<td>55</td>
<td>18</td>
</tr>
<tr>
<td>0x00013A1</td>
<td>55</td>
<td>21</td>
</tr>
<tr>
<td>0x00013A1</td>
<td>55</td>
<td>22</td>
</tr>
<tr>
<td>0x00013A1</td>
<td>55</td>
<td>28</td>
</tr>
<tr>
<td>0x00013A1</td>
<td>55</td>
<td>29</td>
</tr>
<tr>
<td>0x00013A1</td>
<td>55</td>
<td>30</td>
</tr>
<tr>
<td>0x00013A1</td>
<td>55</td>
<td>31</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>16</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>19</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>20</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>23</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>24</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>25</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>26</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>27</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>28</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>29</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>30</td>
</tr>
<tr>
<td>0x00013A2</td>
<td>55</td>
<td>31</td>
</tr>
</tbody>
</table>

This event affects all of the LUT’s contents, regardless of whether multiple LUTs have been placed there or not. For a TMR system, this means that if two LUTs from different domains are placed at the same physical LUT in a SLICEM, then an SEU in one of these bits could cause
<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Fault Injections</th>
<th>Clock CMF</th>
<th>Site CMF</th>
<th>Other CMF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>2,000,000</td>
<td>24</td>
<td>2</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>md5</td>
<td>2,000,000</td>
<td>176</td>
<td>18</td>
<td>1</td>
<td>195</td>
</tr>
<tr>
<td>sha3</td>
<td>12,000,000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>aes128</td>
<td>12,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>28,000,000</td>
<td>201</td>
<td>21</td>
<td>1</td>
<td>223</td>
</tr>
</tbody>
</table>

As will be shown in the next subsection, this occurs much less frequently than clock CMF. This type of CMF has been dubbed “site CMF”.

### 4.3.6 CMF Distribution Analysis

The failures from Table 4.1 at the beginning of the chapter can now be classified into different types of CMF, shown in Table 4.5. The table shows clock CMF is the most common type of CMF that occurs in these triplicated circuits. Only about 10% of CMF does not occur from clock CMF. One bit that is not attributable to clock CMF or site CMF has been detected during fault injection, its cause is unknown.

As the table shows, some circuits are more susceptible to clock CMF than other circuits. This is to be expected as different circuits utilize the device differently and have different implementation constraints. What is important is that clock CMF and the other CMFs can occur on any circuit. Using the analysis techniques presented in the next chapter, it is possible to determine the susceptibility of a particular circuit to clock CMF, presented in Table 4.6. As expected, a higher percentage of the utilized tiles in the b13 and md5 designs could contain clock CMF. Combined with a high utilization of the total number of CLB tiles, 17,420 for the Artix-7 200t part, explains why these designs showed a higher sensitivity to clock CMF during fault injection.

---

3 As part of this work, a database was used to determine how the bits controlled the underlying device architecture. This database contains the information for a large portion of the configuration memory, but is not complete. The unknown bit may in fact be a clock CMF or a site CMF or another unknown CMF.
Table 4.6: Clock CMF Susceptibility of Circuits

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Tiles with CMF</th>
<th>Used Tiles</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>2,471</td>
<td>9,728</td>
<td>25.4%</td>
</tr>
<tr>
<td>md5</td>
<td>6,742</td>
<td>16,167</td>
<td>41.7%</td>
</tr>
<tr>
<td>sha3</td>
<td>89</td>
<td>8,209</td>
<td>1.08%</td>
</tr>
<tr>
<td>aes128</td>
<td>99</td>
<td>13,814</td>
<td>0.72%</td>
</tr>
</tbody>
</table>

Figure 4.18: How One SBU Can Affect Multiple TMR Modules from [3]

4.4 Previous Studies on Single Bit Failures

Other works have also investigated how TMR circuits fail on FPGAs. In [60], the authors introduce and study domain-crossing errors (DCE). A DCE is any event that causes a failure in two or more TMR modules and causes TMR circuit failure. DCEs of all sizes are studied, including 1-bit, 2-bit and 3-bit. A 1-bit DCE is the same as an SBF. The study notes that single-bit DCEs exist, but were not observed in Virtex-I and were rare in Virtex-II testing. Some of the single-bit DCEs observed in the Virtex-II study were attributed to a shared global logic “zero” tie off for unused inputs, but many of the other single-bit DCEs were unrelated to that event and no further cause was specified. One of the main conclusions was that many of the DCEs occur in the routing network. 99% of all observed DCEs occurred in the configuration logic. 75% of these CLB DCEs, occurred solely in the routing network with an additional 22% spanning the routing network and LUT contents.

In [3, 61], the authors study what causes CMF and how to resolve it. Their work presents three cases of how an SBU could affect multiple TMR modules, presented in Figure 4.18.
• **Case A** - Normal operation. Both net A and net B are routed correctly.

• **Case B** - Short. The SEU causes the source of net of net A ($A_S$) to also route to the destination node of net B ($B_D$). Both net A and net B are still routed to their respective destination nodes as well. This creates a short between nets A and B, which if they correspond to different TMR modules, would affect two different TMR domains. This can happen when two nets route through the same switchbox.

• **Case C** - Double open. Both net A and net B become unrouted, disconnecting them from their destination nodes.

• **Case D** - Open combined with a short. Net B unroutes from its destination node and reroutes to $A_D$. This creates a short on net A and an open on net B.

The authors propose mitigation strategies to address these three cases which will be discussed in the next chapter of this dissertation. These results were shown on a Xilinx Spartan-II XC2S30PQ144 device.

The goal of this work is to identify the cause of SBF on FPGAs and then to provide mitigation for SBF. This work differs from previous work in that it studies both SPF and CMF, where other works only study CMF. This work also differs from previous work in the identified cause for CMF. This work agrees that the majority of CMF occurs in the routing network, but occurs from upsets in bits that simultaneously cause multiple shorts. Other works have studied CMF on older FPGA devices (Xilinx Virtex-I, Virtex-II and Spartan-II) while this work studies CMF on a newer FPGA device (Xilinx 7-Series).

### 4.5 Summary

Single bit failures (SBF) can occur in TMR circuits on FPGAs as either single point failures (SPF) or common mode failures (CMF). SPF occurs when a system is not entirely triplicated, such as when the I/O pins are not triplicated (common-IO). CMF occurs even when the system is entirely triplicated.

The majority of SPF occurs in the routing network. The routing network is susceptible on nets that come from the input pins and feed the internal circuitry. SPF can also occur in the
reduction voter network and on nets from the reduction voters to the output pins. In common-IO circuits, the number of SPF bits greatly outnumbers the number of CMF bits.

Most CMF also occurs in the routing network. Routing CMF happens when an upset occurs in a column bit, which can simultaneously cause multiple shorts along the row wires in a routing mux. Empirical results show that CMF occurs when two different clock nets from different domains are involved in these multiple row wire shorts, called clock CMF. Empirical results also show that clock CMF is the most common type of CMF. Another type of CMF is called site CMF and occurs when two different domain LUTs are physically placed on the same SLICEM LUT. With the architectural-level details on how SBF occurs, the next chapter will present several techniques to mitigate SPF and CMF.
CHAPTER 5. MITIGATION FOR SINGLE BIT FAILURES IN TMR

By understanding the low-level architectural cause behind SPF and CMF, it is possible to develop mitigation techniques to address them. This chapter will examine several techniques designed to mitigate both SPF and CMF. Several techniques for addressing SPF will first be discussed. Then the techniques to address CMF will be discussed. After discussing the techniques for SPF and CMF, relevant related work will be discussed. The next chapter then analyzes the impact and effectiveness of these techniques on the reliability of real FPGA designs.

5.1 Single Point Failure Mitigation Techniques

As explained in the previous chapter, most SPF occurs in the routing network from the untriplicated I/O pins to the internal circuitry on the FPGA. Each I/O pin can either be a clock net or a non-clock net. On the device, the clock nets and the non-clock nets are routed using different resources. Thus, different mitigation techniques must be developed for these different nets. Two mitigation techniques were developed and implemented for SPF: split-clock and split-IO. While they will only be presented in the context of untriplicated I/O, these techniques could also be used for untriplicated resources as discussed in the previous chapter. These mitigation techniques are all implemented in Vivado by executing TCL scripts which perform the low-level manipulations, found in Appendix H.

The implementation impact of the proposed techniques can be measured by comparing key metrics against the common-IO circuits. The key metrics, shown in Table 5.1 for the common-IO circuit, are

- **fmax** - The maximum frequency that the clock can be run at in the circuit;
- **Number of nodes** - The number of routing nodes used by the circuit;
- **Number of cells** - The number of cells in a circuit;
Table 5.1: Common-IO (3-Voter) Implementation Metrics

<table>
<thead>
<tr>
<th>Circuit</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>54.16</td>
<td>1,050,721</td>
<td>104,014</td>
<td>17,902</td>
<td>9,107</td>
</tr>
<tr>
<td>md5</td>
<td>90.29</td>
<td>2,674,886</td>
<td>214,479</td>
<td>30,778</td>
<td>15,765</td>
</tr>
<tr>
<td>sha3</td>
<td>52.48</td>
<td>1,341,883</td>
<td>71,886</td>
<td>14,567</td>
<td>7,554</td>
</tr>
<tr>
<td>aes128</td>
<td>87.23</td>
<td>1,884,785</td>
<td>168,013</td>
<td>26,602</td>
<td>13,779</td>
</tr>
</tbody>
</table>

Table 5.2: Routing Nodes Utilized by Untriplicated Nets in Common-IO (3-Voter)

<table>
<thead>
<tr>
<th></th>
<th>b13</th>
<th>md5</th>
<th>sha3</th>
<th>aes128</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes</td>
<td>45,407</td>
<td>69,623</td>
<td>18,604</td>
<td>62,467</td>
</tr>
</tbody>
</table>

- **Number of sites** - The number of sites on the device used by the circuit;

- **Number of tiles** - The number of tiles on the device used by the circuit.

In addition to these key metrics another metric will be used to measure the effectiveness of the SPF mitigation techniques. This metric is the number of routing nodes used by untriplicated nets, shown in Table 5.2 for the common-IO circuit’s metric. The SPF mitigation techniques will be compared against the common-IO metrics as they are discussed.

5.1.1 Split-Clock SPF Mitigation Technique

Clock buffers are used to drive the clock signal onto the clock specific routing network and to minimize the skew across the network. An input I/O pin for a clock is first routed through a clock buffer. There are different clock buffers available on the device, which are used to drive different regions on the device. The most common type of buffer is a global buffer (BUFG). In a typical design, the clock would be routed through a BUFG, or similar buffer, and then routed to each cell in the circuit.

In a common-IO TMR circuit, there would only be the same number of clocks in the triplicated design as there were in the original design since the clock input pins are not triplicated. The SPF impact of the untriplicated clocks can be reduced by internally triplicating the clocks. This can be done by triplicating the buffer each clock pin is routed through. The output of each buffer
is then routed to the cells associated with that particular clock and TMR domain. This mitigation technique is shown in Figure 5.1.

As shown in the figure, there are a few locations where SPF can occur in the common-IO design, denoted by the red “x”s. The first location is from the input pin to the buffer and the second location is from the buffer to the cells of each domain. While the figure only shows one red “x”, it is likely that each of these nets uses multiple wire segments, which would correspond to multiple configuration bits. One “x” is used for simplicity to mark the different locations SPF can occur. After buffer tripling SPF can only occur on the net from the input pin to each of the buffers. The possibility for SPF has been removed from the design implementation after the triplicated buffers.

Triplicating the clocks also has another positive side effect; it adds spatial separation between each TMR domain. Each slice in the device can only be driven by a single clock net. This forces each FF placed on each slice to be driven by the same clock net. This means that before the clock is triplicated, FFs from each domain can be placed on the same slice, but after clock triplication, only FFs from the same domain can be placed on the same slice. This also makes it possible to perform CMF techniques which will be explained in the next section.
By applying the split-clock technique, the geomean average number of routing nodes used by untriplicated nets is reduced by $0.03 \times$ from the common-IO circuit, as shown in Table 5.3. Most of the circuits see a dramatic reduction in routing nodes, but the one exception is the b13 circuit where the number of routing nodes used by untriplicated nets actually increased. This is likely due to the increase in spatial separation between the TMR domains coupled with the high fanout input nets in the b13 circuit.

A potential drawback to the split-clock mitigation technique is the increased utilization of the clock buffers and the increased utilization in the clock routing network on the device. Most complex designs utilize multiple clocks, so there may not be enough buffers to triplicate each clock. Furthermore, each region of the device can only be driven by a few clocks. This can increase the routing congestion for non-clock nets as cells may need to be placed in separate clock regions and then be routed across those regions.

Another potential drawback of this technique is that it may introduce clock skew into the circuit between each of the TMR domain clocks. The additional skew may cause the circuit to not function or the designer may need to slow the clock down. This work does not address any of these scenarios, but investigates the effectiveness of the triplicated buffers on the reliability on the benchmark circuits used for this work.

To address some of these concerns, Table 5.4 shows the implementation metrics across the five metrics as well as the geomean average change over the common-IO circuit. The geomean change for the maximum frequency shows that there is minimal impact on the maximum frequency ($0.99 \times$). The most negatively impacted circuit was the md5 where the maximum frequency changed by $0.97 \times$, but on the other hand the b13 circuit showed a positive change with an increased maximum frequency ($1.01 \times$ change).

### 5.1.2 Split I/O SPF Mitigation Technique

Similar to the split-clock mitigation technique, non-clock inputs on general I/O pins can also be internally triplicated, as shown in Figure 5.2. Similar to clock nets, the non-clock nets route from the input pin through any number of wire segments (denoted by the circles in the figure) before it splits to feed each TMR domain. As the figure shows, SPF can occur before the
Table 5.4: Split-Clock Implementation Metrics

<table>
<thead>
<tr>
<th>Circuit</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>54.46</td>
<td>1,072,431</td>
<td>104,016</td>
<td>17,954</td>
<td>9,149</td>
</tr>
<tr>
<td></td>
<td>1.01×</td>
<td>1.02×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>1.00×</td>
</tr>
<tr>
<td>md5</td>
<td>87.52</td>
<td>2,680,478</td>
<td>214,481</td>
<td>31,260</td>
<td>15,968</td>
</tr>
<tr>
<td></td>
<td>0.97×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>1.02×</td>
<td>1.01×</td>
</tr>
<tr>
<td>sha3</td>
<td>52.38</td>
<td>1,382,390</td>
<td>71,888</td>
<td>15,841</td>
<td>8,376</td>
</tr>
<tr>
<td></td>
<td>1.00×</td>
<td>1.03×</td>
<td>1.00×</td>
<td>1.09×</td>
<td>1.11×</td>
</tr>
<tr>
<td>aes128</td>
<td>87.10</td>
<td>1,880,774</td>
<td>168,015</td>
<td>26,668</td>
<td>13,867</td>
</tr>
<tr>
<td></td>
<td>1.00×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>1.01×</td>
</tr>
<tr>
<td>Geomean</td>
<td>0.99×</td>
<td>1.01×</td>
<td>1.00×</td>
<td>1.03×</td>
<td>1.03×</td>
</tr>
</tbody>
</table>

Figure 5.2: Split I/O Technique

net splits denoted with the red “x”s. Also similar to clock pins, SPF can be mitigated by splitting the net early so that SPF is limited to occurring near the input pin.

Splitting the non-clock nets must be performed differently than splitting the clock nets. Unlike clock nets which pass through a buffer before being routed onto the clock tree, non-clock nets are routed directly from the input pin to each corresponding cell. To mitigate SPF, the net should be split as early as possible. Just how early the net is able to split is dependent on the device architecture as there may be local routing from the I/O pin to the configurable routing network that needs to be performed before the net can be routed onto the general routing network.

There are two steps to implementing the split-IO technique within the vendor tool. The first step is to introduce a pseudo-buffer into the netlist. Like a clock buffer, this pseudo-buffer can be inserted for each separate domain to force the net to split. The pseudo-buffer is implemented as a pass-through LUT, which is a single input LUT, whose function is to copy the logic value of
the input wire onto the output wire. Since these LUTs are inserted after TMR insertion (which occurs after synthesis and optimizations), the pass-through LUTs will not be optimized away from the netlist. The split-IO technique using pass-through LUTs is shown in Figure 5.3. The second step to the split-IO technique is to constrain these pseudo-buffer LUTs to be placed in the CLB tile closest to the input pin. The exact TCL commands that were used to do this can be found in Appendix H.

Implementing the split-IO technique can impact device utilization in two ways. First, there is a need for three additional LUTs to serve as the pseudo-buffers for each input net. These LUTs would be inserted into the netlist and would increase the number of LUTs by three times the number of non-clock input pins for a given circuit. Second, this would increase the routing utilization and congestion. Without the pseudo-buffers the tool is free to split the net as close to the sink cells as it would like. With the pseudo-buffers in place, the tool must route three copies of the net to the sink cells. The increase in routing congestion would be dependent on the number of sink cells for the net, and how close the sink cells are placed to the pseudo-buffers on the device.

Tables 5.5 shows the impact the split-IO technique has on the circuit implementation metrics. As shown, there is a positive impact on the timing for all circuits, except the md5 (which was minimally impacted). Overall, fmax had a geomean improvement of $1.09 \times$. The other metrics show there is a minimal impact on the number of routing nodes ($1.01 \times$ average change) and the number of cells, sites and tiles.
Table 5.5: Split-IO Implementation Metrics

<table>
<thead>
<tr>
<th>Circuit</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>57.60</td>
<td>1,053,805</td>
<td>104,044</td>
<td>18,524</td>
<td>9,605</td>
</tr>
<tr>
<td></td>
<td>1.06×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>1.03×</td>
<td>1.05×</td>
</tr>
<tr>
<td>md5</td>
<td>89.29</td>
<td>2,686,487</td>
<td>214,509</td>
<td>30,779</td>
<td>15,765</td>
</tr>
<tr>
<td></td>
<td>0.99×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>1.00×</td>
</tr>
<tr>
<td>sha3</td>
<td>57.24</td>
<td>1,381,222</td>
<td>71,916</td>
<td>13,961</td>
<td>7,141</td>
</tr>
<tr>
<td></td>
<td>1.09×</td>
<td>1.03×</td>
<td>1.00×</td>
<td>0.96×</td>
<td>0.95×</td>
</tr>
<tr>
<td>aes128</td>
<td>108.27</td>
<td>1,890,578</td>
<td>168,043</td>
<td>25,175</td>
<td>13,625</td>
</tr>
<tr>
<td></td>
<td>1.24×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>0.95×</td>
<td>0.99×</td>
</tr>
<tr>
<td>Geomean</td>
<td>1.09×</td>
<td>1.01×</td>
<td>1.00×</td>
<td>0.98×</td>
<td>1.00×</td>
</tr>
</tbody>
</table>

Table 5.6: Split-IO Routing Nodes Utilized by Untriplicated Nets

<table>
<thead>
<tr>
<th></th>
<th>b13</th>
<th>md5</th>
<th>sha3</th>
<th>aes128</th>
<th>Geomean</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes</td>
<td>16,181</td>
<td>68,624</td>
<td>17,181</td>
<td>52,560</td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>0.36×</td>
<td>0.99×</td>
<td>0.92×</td>
<td>0.84×</td>
<td>0.72×</td>
</tr>
</tbody>
</table>

Tables 5.6 shows how the split-IO technique affects the number of routing nodes used by untriplicated nets. The split-IO technique has a much lower impact on the utilization by untriplicated nets than the split-clock technique, a $0.72\times$ change instead of a $0.03\times$ change for split-clock. This suggests that an untriplicated clock network utilizes more of the routing network than the non-clock nets. However, split-IO had a much larger impact on the b13 than the split-clock technique.

5.1.3 Output Placement

Untriplicated nets that route to an output pin can also cause SPF. It is harder to provide mitigation for nets that route to an output pin. This is because the net is coming from the redundant circuitry and needs to be reduced to a single output wire. The triplication from TMR ceases at the reduction voter, which is used to propagate the majority value from the TMR domains to the output pin, as shown in Figure 5.4. The wire segments from the reduction voter to the output pin are susceptible to SPF. Because only a single net can drive the output pin, no buffers can be inserted to triplicate the output net. Instead the only strategy is to ensure that the path from the reduction voter to the output pin is as small as possible. This is done by placing the reduction voter in the CLB.
tile closest to the output pin, just like the pseudo-buffers were placed in the CLB tile closest to the input pin. No additional cells are needed to implement this technique. For the results (including the implementation metrics just presented), output placement will be combined with split-IO.

5.1.4 Early Split SPF Mitigation Technique

All of these techniques, split-clock, split-IO and output placement, can be used together. When used together, they are referred to as early split (ES). For maximum SPF mitigation, they would be most effective when used together, but certain missions may dictate otherwise. For example, there may be environments where parts of the FPGA may be more susceptible to failure than other parts. Or, there may be enough resources to implement split-IO, but there may not be enough resources to implement split-clock.

Table 5.7 shows the impact the ES technique has on the implementation metrics. As the geomean shows, there is almost no impact on any of the metrics over the common-IO circuit. However, the specific fmax results for each circuit show opposing trends. The md5 circuit shows a negative impact on the maximum frequency (0.67× change) while all of the other circuits show a positive impact on the maximum frequency. These balance each other out in the geomean which shows no (1.00×) change over the common-IO circuit.

Table 5.8 shows the impact the ES technique has on the nodes used by untriplicated nets in the circuit. The number of untriplicated nodes is greatly reduced by implementing split-clock, split-IO and output placement together. From a strict SPF perspective, this should have a positive impact on the circuit reliability, which will be shown in the next chapter.
Table 5.7: Early-Split Implementation Metrics

<table>
<thead>
<tr>
<th>Circuit</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>56.84</td>
<td>1,074,569</td>
<td>104,046</td>
<td>18,164</td>
<td>9,303</td>
</tr>
<tr>
<td></td>
<td>1.05×</td>
<td>1.02×</td>
<td>1.00×</td>
<td>1.01×</td>
<td>1.02×</td>
</tr>
<tr>
<td>md5</td>
<td>60.57</td>
<td>2,695,277</td>
<td>214,511</td>
<td>30,497</td>
<td>15,668</td>
</tr>
<tr>
<td></td>
<td>0.67×</td>
<td>1.01×</td>
<td>1.00×</td>
<td>0.99×</td>
<td>0.99×</td>
</tr>
<tr>
<td>sha3</td>
<td>57.13</td>
<td>1,331,938</td>
<td>71,918</td>
<td>14,447</td>
<td>7,414</td>
</tr>
<tr>
<td></td>
<td>1.09×</td>
<td>0.99×</td>
<td>1.00×</td>
<td>0.99×</td>
<td>0.98×</td>
</tr>
<tr>
<td>aes128</td>
<td>112.42</td>
<td>1,894,084</td>
<td>168,045</td>
<td>25,165</td>
<td>13,598</td>
</tr>
<tr>
<td></td>
<td>1.29×</td>
<td>1.00×</td>
<td>1.00×</td>
<td>0.95×</td>
<td>0.99×</td>
</tr>
<tr>
<td>Geomean</td>
<td>1.00×</td>
<td>1.01×</td>
<td>1.00×</td>
<td>0.99×</td>
<td>1.00×</td>
</tr>
</tbody>
</table>

Table 5.8: Early-Split Routing Nodes Utilized by Untriplicated Nets

<table>
<thead>
<tr>
<th></th>
<th>b13</th>
<th>md5</th>
<th>sha3</th>
<th>aes128</th>
<th>Geomean</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes</td>
<td>237</td>
<td>172</td>
<td>169</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>0.01×</td>
<td>0.00×</td>
<td>0.01×</td>
<td>0.00×</td>
<td>0.00×</td>
</tr>
</tbody>
</table>

5.2 Common Mode Failure Mitigation Techniques

To address CMF, three mitigation techniques were developed: incremental routing (RCMF), incremental placement (PCMF) and striping. RCMF did not prove to be as promising a technique and the details for technique are only discussed in Appendix E. The two incremental techniques, RCMF and PCMF, address clock CMF and are implemented as separate steps to the design implementation flow, which are implemented in the open source CAD tools RapidSmith2 and Rapid-Wright [7,62]. The striping technique addresses both clock CMF and site CMF and is a set of tool command language (TCL) commands that are inputs into the typical flow, as shown in Figure 5.5.

As the figure shows, the typical flow through the tool chain after inserting netlist TMR is to first place the design, then to route the design and finally to generate the bitstream which can be used to program the device. The incremental techniques take place after specific tool chain steps and then are inserted back into the tool chain either using Tincr or a DCP file [63].

Table 5.9 shows the key implementation metrics for the trip-IO circuit. The baseline results for trip-IO circuit can be used as a baseline to compare against the mitigation techniques that

---

1 A discussion on the entire design implementation flow and open source CAD tools can be found in Appendix A
will be presented throughout this section and to measure the impact these techniques have on the implementation.

Before describing PCMF and striping, there are several new concepts that need to be introduced. These concepts are pertinent to the PCMF algorithm. The first concept is the notion of a fail set, or a set of cells that can cause circuit failure. The second concept is downstream cells. These concepts are first described, followed by the PCMF algorithm and then the striping algorithm.

### 5.2.1 Fail Sets

The PCMF algorithm that will be described relies on an algorithm that can identify a fail set. A fail set is any set of cells that would cause TMR failure if all of the cells in the set simultaneously failed. For example, in a fully triplicated design the following sets of cells would form a fail set:

- A set of all the cells in a design;

<table>
<thead>
<tr>
<th>Circuit</th>
<th>( f_{\text{max}} ) (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{b13}</td>
<td>58.24</td>
<td>1,259,660</td>
<td>104,258</td>
<td>18,751</td>
<td>9,754</td>
</tr>
<tr>
<td>\textbf{md5}</td>
<td>75.47</td>
<td>2,723,574</td>
<td>214,723</td>
<td>31,729</td>
<td>16,244</td>
</tr>
<tr>
<td>\textbf{sha3}</td>
<td>55.46</td>
<td>1,412,958</td>
<td>72,130</td>
<td>15,573</td>
<td>8,286</td>
</tr>
<tr>
<td>\textbf{aes128}</td>
<td>112.03</td>
<td>1,884,604</td>
<td>168,257</td>
<td>26,716</td>
<td>13,891</td>
</tr>
</tbody>
</table>
Algorithm 1 Identify Fail Set

procedure ISFAILSET(cells)
    Initialize set<Domain> possFailSets
    for each cell in cells do
        possFailSets.add(cell.domain)
    end for
    if possFailSets.size() ≥ 2 then
        return True
    end if
    return False
end procedure

• A set of all the cells in two of the three domains (3 total sets).

Just as there are obvious fail set groups, there are obvious non-fail set groups:

• The null (empty) set;

• A set containing only one cell;

• A set containing any number of cells limited to a single domain.

For any non-trivial set of cells (such as the ones just described), determining whether the set is a fail set or not is more challenging. Does a set containing two cells from different domains constitute a fail set? Some cells only affect the output vectors under certain input conditions. Identifying fail sets is difficult using only a static analysis of the circuit.

When identifying fail sets, it is much more important to positively identify fail sets as opposed to correctly identifying non-fail sets. Under this stipulation, it would permtitable to allow false positives as long as there are no false negatives. This can easily be done using the algorithm shown in Algorithm 1. This algorithm identifies fail sets based on whether there are cells from multiple domains or not. If the input set of cells pertain to more than one TMR domain, then this algorithm classifies the set as a fail set, otherwise the set is not a fail set. This algorithm would not allow any false negatives since a failure must involve more than one TMR domain. This is sufficient to explain how to perform the PCMF technique, but a better classifier is presented in Appendix E.
5.2.2 Downstream Cells

When a failure occurs on a net, it will not cause a failure in all of the cells on that net since each routing mux contains a buffer on its output. This means that a failure will only be propagated forward on the net, and only the downstream cells on the net will fail. No upstream cells on the net will be affected. For example, in Figure 5.6, node “4” experiences a short and fails as a result. Only the cells immediately downstream of node “4” fail, while the other cells on the net do not fail. When considering fail sets in the design, only downstream cells from a potential short need to be considered.

5.2.3 Incremental Placement (PCMF)

The first technique used to remove CMF from a design is incremental placement or placement CMF (PCMF). PCMF is a mitigation technique aimed at removing clock CMF from a circuit. PCMF uses incremental placement, and in incremental placement the placement of a design is gradually altered until some goal is obtained. For PCMF, this goal is that there are no tiles where clock CMF can occur. PCMF removes clock CMF from the implementation by swapping slices in tiles with CMF with slices in other tiles, one at a time. This goal is achieved by constraining both slices in the tile to use the same TMR domain clock or by ensuring that all of the clocked cells in the tile do not form a fail set.
The PCMF algorithm follows these general steps:

1. Start with placement from Vivado;

2. Identify tiles with clock CMF;

3. Swap slices among tiles to remove clock CMF;

4. Repeat step 3 until all clock CMF is removed from the design.

Placement occurs in Vivado and is exported into the PCMF algorithm either using a RapidSmith2 Checkpoint (RSCP) file or a DCP file, depending on whether RapidSmith2 or RapidWright is being used, respectively. Steps 2-4 take place in RapidWright/RapidSmith2, and the altered design is imported back into Vivado either using a Tincr checkpoint (TCP) file or a DCP file, again dependent on whether RapidSmith2 or RapidWright is used to perform PCMF. This flow is shown in Figure 5.7.

After placement the PCMF algorithm needs to identify tiles with clock CMF. The steps to identify clock CMF are:

1. Identify tiles with slices driven by clocks from two different TMR domains.

2. Determine if the clock-driven cells in these slices form a fail set.

Identifying tiles with clocks from two different TMR domains can be done by iterating over all of the used tiles in a design within the RapidWright/RapidSmith2 API. The RapidWright/RapidSmith2 API can then be used to query if a tile contains clocks from multiple TMR domains.

If a query returns that a tile contains clocks from multiple TMR domains, then the PCMF algorithm needs to determine whether the clock-driven cells from the slices in that tile form a fail set or not. These clock-driven cells represent the downstream cells from the mux where the multiple clock shorts would occur, as explained in the previous chapter. If the fail set algorithm
The next step of the PCMF algorithm is to swap sites in the tiles with clock CMF into different tiles. This step is repeated until no tiles with clock CMF remain in the design implementation. A swap between any two tiles can be performed by selecting one slice in each tile and swapping their locations, as shown in Figure 5.8. In this figure, the tile with CMF is shown in green and a potential swap could be performed with the tile directly above it. In the example, the left slice in both tiles is selected for the swap. A potential swap could be performed with any selection of two slices from both tiles.

Performing a swap may not resolve clock CMF. After performing a swap, clock CMF may exist in both, one of, or none of the tiles involved in the swap. Clock CMF can be detected after the swap by using the same algorithm that was used to originally detect it (the “Identify CMF” step of the PCMF algorithm). The only difference is that only the tiles involved in the swap need to be checked for clock CMF. If CMF is detected after the swap, there are a few options available to the algorithm designer:

Algorithm 2 CMF Identification

```plaintext
procedure IDENTIFYCMF(design)
    Initialize collection<Tile> usedTiles
    Initialize collection<Slice> tileSlices
    Initialize collection<Cell> cellsWithClks
    Initialize collection<Tile> tilesWithCMF
    usedTiles ← design.getUsedTiles()
    for each tile in usedTiles do
        tileSlices ← tile.getUsedSlices()
        cellsWithClks.clear()
        for each slice in tileSlices do
            cellsWithClks.addAll(slice.getCellsWithClocks())
        end for
        if ISFAILSET(cellsWithClks) then
            tilesWithCMF.add(tile)
        end if
    end for
    return tilesWithCMF
end procedure
```

determines that the clock-driven cells do form a fail set, then this tile is saved for processing later in the algorithm. The pseudo-code for the “Identify CMF” step of the PCMF algorithm is shown in Algorithm 2.
Figure 5.8: Example of a Swap During the PCMF Algorithm

1. Undo the swap and search for a different swap;

2. Accept the swap if there are fewer tiles with CMF than there were before the swap. The tile without CMF would be removed from the list of tiles with CMF, while the tile with CMF would remain on the list;

3. Accept the swap if the number of tiles with CMF is equal to the number of tiles with CMF before the swap occurred. Ensure that the tiles with CMF list is updated properly. In some cases, the list will remove the original tile with CMF and add the other tile that was swapped;

4. Accept the swap and ensure that both tiles are on the tiles with CMF list. This may increase the number of tiles with CMF if one of the tiles did not have clock CMF before the swap occurred.

Each of these options has its advantages and disadvantages. Option 1 is a greedy algorithm that aggressively searches for an implementation without CMF. It requires less book keeping than the other options. One of its downsides is that it may never find a solution that completely eliminates CMF. Option 2 is also a greedy algorithm, but requires more bookkeeping. The algorithm may have an easier time finding solutions that completely eliminate CMF than option 1. Option 3 does not reduce the CMF impact, but it does not increase it either. This option would allow more design space exploration of the circuit which should help the algorithm find a CMF-free solution, but will
also increase the algorithm’s run time. Option 4 allows for even more design space exploration, but runs the risk of never terminating. This work uses option 2, due to its simplicity in implementing and space exploration it provides.

To limit how far slices are moved from their original placement, the algorithm searches for potential swap tiles in iterations. During the first iteration, only tiles that are immediate neighbors to the tile with CMF are explored, as shown in Figure 5.8. The eight tiles shown are the immediate neighbors to the tile in the center. Each combination of slice swaps within each of the tiles is explored to perform the swap. All potential swaps are analyzed and the swap that reduces the number of CMF tiles the most is accepted. If a swap is explored that eliminates CMF from all tiles it is immediately accepted. The algorithm proceeds by exploring swaps for the next tile with CMF in the list. If no acceptable swaps are found during this iteration, the algorithm proceeds to the next tile in the list.

After the first iteration is completed, the algorithm searches for potential swaps for all of the remaining tiles among all tiles that are neighbors to its immediate neighbors. For example, in the second iteration, a total of 24 tiles are explored, 8 among the immediate neighbors of the tile and 16 among the neighbors of the immediate neighbors. The original neighbor tiles are explored as their slice contents may have changed from other swaps. The algorithm then proceeds the same as the first iteration (where potential swaps were only explored among the immediate neighbors). After the completion of the second iteration, if there are still tiles with CMF the algorithm proceeds to search the neighbors of the tiles explored during the second iteration. The algorithm continues to iterate until there are no tiles left with CMF or there are no more tiles to search on the device. In the latter’s case, the algorithm would leave CMF in the design implementation. The complete algorithm for swapping is shown in Algorithm 3.

Just like the SPF mitigation techniques, the PCMF mitigation technique could have negative or positive impact on the design implementation metrics. Table 5.10 shows the impact PCMF has when applied on the trip-IO circuit. The geomean change in fmax was only $0.99 \times$, meaning that the maximum frequency was only minimally impacted. Only the md5 circuit showed a negative impact on the maximum frequency, a $0.94 \times$ change from the trip-IO version of the circuit.

Another metric that was slightly negatively impacted was the number of routing nodes, which showed a geomean change of $1.01 \times$ from the trip-IO circuit. Again the md5 circuit was the
most impacted with a 1.04× change. The number of cells did not change, as no cells were inserted into the netlist and the number of sites and number of tiles used slightly decreased over the trip-IO circuit.

5.2.4 PCMF Algorithm Results

Another important metric is the time it takes the PCMF algorithm to run, which is reported in Table 5.11. The number of CMF tiles before PCMF and after PCMF along with the number of
iterations it took the PCMF algorithm to complete are also reported in the table. In order to reduce routing congestion, the number of swap iterations for PCMF was limited to 20. This limits how far cells are allowed to move from their starting location. For most of the circuits, the algorithm took less than two seconds to complete. The exception is the md5 circuit, which took several hours to complete. This is likely due to the high amount of carry chains in the circuit, which need to be preserved during swapping. Swapping a single carry chain to a new location could displace another carry chain and this effect could cascade to affect many carry chains. Thus many swaps may need to be considered to analyze a single swap.
Carry Chains

The PCMF algorithm described above does not account for carry chains in the design. Carry chains are special structures within the FPGA device to implement quick accumulation operations. When there is a carry chain dependency between two slices in a circuit, then those slices must be placed in the same slice column, as shown in Figure 5.9\textsuperscript{2}. An illegal placement would occur if one slice from a carry chain is swapped instead of the entire chain. More swaps may need to occur to accommodate the entire carry chain.

The PCMF algorithm can be altered so the entire carry chain is swapped with the respective slice when a swap occurs, as shown in Figure 5.10. The swap of the entire carry chain may create a swap that involves another carry chain (in the other swap tile), and then the other carry chain would also need to be swapped in its entirety. When all of the swapping is completed, many carry chains across many different tiles and slices may have been swapped. In the example from the figure, the tile with CMF contains a slice with a carry chain. Both of these carry chains are swapped, but this swap affects a carry chain in the swap tile. This carry chain dependency must also be swapped which then resolves all of the carry chain dependencies. All tiles involved in the swap(s) need to be checked for CMF after the swap.

\textsuperscript{2}More information on carry chains can be found in Appendix B
5.2.5 Striping

Another technique for addressing CMF is to implement striping. Striping removes CMF differently than PCMF as there is no need for a separate tool flow or any open-source CAD tools, as shown in Figure 5.5. Instead, striping resolves CMF from a different angle: it forces the vendor tool to implement the design without introducing CMF. The steps to implement the striping flow are the same as any other flow, that is the TMR design is placed and then routed followed by bitstream generation.

Since the vendor’s tool has no inherent knowledge of CMF, the tool must be constrained in a way that prevents CMF during implementation. These constraints are similar to other placement constraints that tell the tool where I/O pins should be placed as well as general floor planning constraints. The tool must abide by these constraints or report that an implementation is not possible with the given constraints and circuit netlist. The striping technique can be implemented as a set of TCL commands that are input into the flow, using an XDC file which place constraints on the implementation.

The previous chapter shows that CMF can occur when there are either two domain clocks in the same tile (clock CMF) or cells from multiple domains are placed on the same LUT (site CMF). A single LUT belongs to a single slice which belongs to a single tile. Therefore, if a tile is constrained so that only the cells from a single domain could be placed on it, then it would prevent

Figure 5.10: Example of a Carry Chain Dependency
both clock CMF and site CMF. This occurs because no tile would need two domain clocks, as there are only cells from a single domain. Similarly, two domains could not share the same LUT, as only one domain’s cells could be placed on any LUT in a given tile.

Each tile in a design could be constrained to a single domain to prevent CMF. However, it is unclear which domains should be assigned to each tile. The amount of routing needed for a circuit grows as the domains are spread further away on the device. For example, if domains are located on opposite sides of the chip, significant routing would need to be performed to route between the inputs and outputs of the voters at each partition. This could increase routing congestion significantly and even create an unroutable design. This would also impact the maximum frequency the circuit can achieve.

How each tile is constrained could also impact the successful completion of the placer. If there are not enough resources dedicated to a single domain then the tool may not be able to place all of the cells for a domain. This can be easily solved by assuring an equal distribution of tiles for all of the domains. There are also circuit specific constraints that must be met, such as carry chain dependencies. These dependencies require that a domains cells must be placed on consecutive row tiles. An alternating technique, where a domain is constrained to every third tile would not work for a circuit with carry chain dependencies.

To accommodate carry chain dependencies and to limit routing congestion between domains, the striping mitigation technique constrains each domain to every third column of the device, as shown in Figure 5.11. This ensures that every tile in device is limited to a single domain and carry chains have an entire column for their dependencies. Alternating columns helps to reduce the routing needed between the domains and the voters.

Striping is implemented through a set of TCL commands that create pblocks throughout the device and then assigns specific domains to those pblocks [64]. The steps to performing striping are:

1. Create a physical block (pblock) for each column of the device;
2. Combine pblocks such that every third column is included within the same TMR domain;
3. Assign all the cells within a TMR domain to be placed within the associated pblocks.

The specific TCL commands to implement striping are shown in Appendix H.
Table 5.12 shows how striping affects the implementation metrics of the trip-IO circuit. When striping was applied to the md5 and sha3 circuits, the router was unable to route the design. Thus, striping produced an unimplementable circuit for the md5 and sha3 circuits and no implementation metrics are available for those designs. The geomean for fmax also shows that striping has a significant impact on the maximum clock frequency, with a geomean change of $0.89 \times$ over the base trip-IO circuit.

5.2.6 Combining CMF Mitigation with SPF Mitigation Techniques

Both PCMF and striping can be used in tandem with the SPF techniques to increase the reliability of common-IO TMR. While striping can be used without any SPF techniques, doing so has little practical sense, as the failure cross-section of a common-IO design will usually be dominated by SPF. Before applying any CMF technique, the most effective SPF techniques should first
be applied. To use the PCMF technique, the clock must be triplicated since the failure mechanism involves multiple clocks.

Tables 5.13 and 5.14 show how applying PCMF to the split-clock and early-split SPF mitigation techniques affects the implementation metrics over the common-IO circuit. The impact of PCMF on the split-clock and early-split techniques shows similar trends as were observed for the implementation metrics of those techniques, as discussed in the SPF section of this chapter. PCMF does not seem to significantly impact these circuits more than the split-clock and early-split techniques had already impacted the common-IO circuit.

5.3 Previous Work

This dissertation is not the first work to propose low-level placement and routing changes to improve TMR circuits on FPGAs. The most relatable work to the mitigation techniques proposed
Table 5.14: Early-Split-PCMF Implementation Metrics

<table>
<thead>
<tr>
<th>Circuit</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>57.08</td>
<td>1,085,475</td>
<td>104,046</td>
<td>18,164</td>
<td>9,360</td>
</tr>
<tr>
<td></td>
<td>1.05×</td>
<td>1.03×</td>
<td>1.00×</td>
<td>1.01×</td>
<td>1.03×</td>
</tr>
<tr>
<td>md5</td>
<td>59.58</td>
<td>2,795,225</td>
<td>214,511</td>
<td>30,497</td>
<td>15,889</td>
</tr>
<tr>
<td></td>
<td>0.66×</td>
<td>1.04×</td>
<td>1.00×</td>
<td>0.99×</td>
<td>1.01×</td>
</tr>
<tr>
<td>sha3</td>
<td>57.44</td>
<td>1,411,757</td>
<td>71,918</td>
<td>15,842</td>
<td>8,384</td>
</tr>
<tr>
<td></td>
<td>1.09×</td>
<td>1.05×</td>
<td>1.00×</td>
<td>1.09×</td>
<td>1.11×</td>
</tr>
<tr>
<td>aes128</td>
<td>113.13</td>
<td>1,894,716</td>
<td>168,045</td>
<td>25,165</td>
<td>13,612</td>
</tr>
<tr>
<td></td>
<td>1.30×</td>
<td>1.01×</td>
<td>1.00×</td>
<td>0.95×</td>
<td>0.99×</td>
</tr>
<tr>
<td>Geomean</td>
<td>1.00×</td>
<td>1.03×</td>
<td>1.00×</td>
<td>1.01×</td>
<td>1.03×</td>
</tr>
</tbody>
</table>

In this chapter was done in [3, 61]. In this work, the authors propose an algorithm to reduce the impact of CMF called the reliability-oriented place and route algorithm (RoRA). This algorithm consists of both a place and route function. The placement algorithm aims to optimize two constraints: minimizing the congestion in a routing window and maximizing the reliability. This is done by calculating the number of potential reliability constraints that might be generated from a given placement.

The routing algorithm is composed of two steps, the global router followed by a local router. The global router aims to find general paths to from the sink to source node, optimizing for routability, while considering nodes that would cause reliability concerns to be forbidden. The local router then assigns actual wires to the net based on the global route. In the end, their results showed a 22% slower frequency on average than the Xilinx routed solutions and there were still single bits that caused TMR failure in the circuit, but RoRA did show a significant reduction in the number of bits causing TMR failure. Additional studies of RoRA were performed in [65], where the authors presented an ADD8 circuit that did not show any failures with about 5,000 fault injections during a fault injection campaign.

In addition to RoRA, the authors also proposed a static analysis tool in [66] to determine the number of critical bits in a circuit, i.e., the number of bits that would cause circuit failure. This tool was validated on a Virtex-2 chip using fault injection. This tool was later incorporated into a larger tool and called the static analyzer tool (STAR-LX), and validated in radiation [67]. By understanding the underlying device architecture, how the circuit is implemented and how bits can cause circuit failure, STAR-LX can estimate the number of critical bits in a circuit. The authors
show that it is faster to use STAR-LX to estimate the circuit sensitivity than it would be to perform fault injection. The authors do not comment on the potential impact of MCUs.

The STAR-LX tool was then incorporated into another tool which also provided mitigation for the estimated critical bits, called VERI-Place [68]. In one study of the VERI-Place tool on the b13 design, the VERI-Place tool improved the circuit FIT rate $16 \times$ over an unmitigated design and $13 \times$ over the X-TMR tool [44]. In another study, the VERI-Place tool was used to mitigate SBF a Cortex-M0 processor implemented on a Virtex-5 FPGA [69]. Radiation testing in a proton beam showed that VERI-Place improved the cross-section over the unmitigated design by $46 \times$ and improved the cross-section over the XTMR version by $89 \times$. The XTMR version performed worse than the unmitigated version. The papers seem to suggest that this tool is just applying placement and routing constraints.

Other works have also shown that the placement and routing can be altered to achieve greater reliability [70, 71]. These algorithms propose changes to the cost algorithms in the placer and router that make the CAD steps aware of potential reliability concerns. They show a reduction in their internally calculated error propagation probability (EPP) metric, but stop short of showing any fault injection results. The main drawback of these works is that they are implemented on theoretical device architectures that are used with the VTR tool [72]. The architecture and electrical properties of real devices differ from the simulated architectures used in these previous works. The designer must make assumptions about how configuration memory interacts with the device and how the theoretical device should electrically respond when a short occurs. These assumptions make it unclear how these algorithms would translate to a real device.

This work is similar to previous works in that it seeks to mitigate SPF and CMF through changes in the placement and routing. The specific placement and routing changes are specific to this work because the low-level failure mechanisms presented in the last chapter differ from previous work. Ultimately, the metrics that matter are the bit sensitivity and cross-section improvements. As will be shown in the next chapter, this work shows a high bit sensitivity improvement in fault injection with millions of injections (instead of thousands) and the cross-section improvements in neutron radiation are greater in this work.
5.4 Summary

This chapter first explored a couple of techniques to mitigate for SPF that occurs in common-
IO systems, called split-clock and split-IO. Split-clock forces the clock nets to split into three do-
mains by inserting additional clock buffers. Similarly, split-IO forces all other nets to split into
three domains by inserting pseudo-buffers and placing them near the I/O. Combined these tech-
niques are called early-split (ES).

This chapter then explored a couple of techniques to address CMF. Clock CMF can be
resolved with a technique called placement CMF (PCMF). In PCMF slices are swapped until clock
CMF does not exist in any tile. Clock CMF can also be resolved with a technique called striping,
which also address site CMF. In striping, each domain can only be placed in every third column of
the device. Both of these techniques can also be applied to common-IO designs, but should only
be employed after SPF mitigation techniques have also been employed.

Both the SPF and CMF mitigation techniques modify the design at the low-level by altering
how the design is placed and/or routed. While these techniques look promising, they have not been
proven using fault injection or radiation testing. The next chapter will show how these mitigation
techniques perform during these tests and how they improve a circuit’s reliability.
CHAPTER 6. RELIABILITY IMPROVEMENT AND TESTING RESULTS

The last chapter explored several mitigation techniques to reduce the impact of SPF and CMF. The chapter showed how each SPF mitigation technique reduced the number of untriplicated routing nodes in the circuit or, in the case of CMF, showed how the CMF mitigation techniques reduced the number of tiles with clock CMF. What the last chapter did not show was how each of the mitigation techniques impacted the circuit reliability metrics, such as the bit sensitivity and radiation cross-section.

This chapter presents and analyzes the impact the SPF and CMF mitigation techniques have on the circuit reliability metrics. This chapter first presents the test structure and methodology used for both fault injection and radiation testing. Then the results of the fault injection test and radiation tests are presented. The chapter then concludes with an analysis of how the mitigation techniques addressed SBF and the limitations of the mitigation techniques.

6.1 Test Structure and Methodology

A proper test structure and methodology is needed to properly measure the reliability metrics of the SPF and CMF mitigation techniques. This section explains how the fault injection and radiation testing data was measured and collected. The same setup is used for both types of testing environments, fault injection and radiation testing, so that different trends between the two types of tests can be explored and analyzed.

For a dynamic test, detecting failures is a complicated task as failures can occur at any time. To obtain accurate logs, all detection and logging logic in the circuit should operate reliably. This work uses a golden copy with the device under test (DUT) structure for testing, shown in Figure 6.1. In this setup there are two different chips, the golden copy and the DUT, both running the same copy of the circuit. The only difference between the two chips is that the DUT is exposed to faults, whether that be through fault injection or radiation testing, while the golden copy is not.
subject to any faults. The boards are run in lockstep so that failures in the DUT can be detected in real time. All detection logic is implemented on the golden copy chip so that the detection logic is not subject to faults.

Since the mitigation techniques were developed for Xilinx 7-Series FPGAs, the Nexys Video Artix-7 boards available from Digilent were chosen for testing. The Nexys Video Artix-7 is equipped with an Artix-7 200T FPGA chip which contains 215,360 logic cells, 33,650 slices, 269,200 flip-flops and has a total of 59,145,600 bits of configuration memory. This board was chosen because of its relatively cheap cost (about $200 academic pricing) and ability to access the signals needed for the experimental setup.

The Nexys Video Artix-7 boards were configured into a golden copy with DUT structure as shown in Figure 6.2. This setup is called the TURTLE (Testing ultra-reliability techniques using low-cost equipment). One of the boards is the “golden copy”, referred to as the master, while the other board is the “DUT”, referred to as the slave. The boards are connected together via an FMC coupler card which handles all of the communication between the boards. There are 22 signals that communicate over the FMC coupler card:

- 1 signal for the clock;
- 1 signal for the reset;
- 10 signals for the inputs;
- 10 signals for the outputs.

When a common-IO circuit is being tested, there are only 22 signals communicating across the FMC coupler card. When a trip-IO circuit is being tested, all of these signals are triplicated, so a
total of 66 signals are communicating across the FMC coupler card. In addition to the communication signals across the FMC coupler card, the JTAG chain is extended across the card so that the golden copy and DUT are on the same JTAG chain.

The master device contains multiple circuits to support the test structure. First, the master contains the golden copy of the circuit under test. This golden copy circuit is run in lockstep with the DUT and fed with the same input vectors, clock and reset lines. In addition to the golden copy of the circuit, the master device also contains a comparator, a BSCAN circuit, a synchronization state machine and the input vector generation circuit. The comparator compares the output vectors of the golden copy and DUT (from the slave device) every clock cycle. When a mismatch is observed on any bit in the 10-bit output vector, the comparator notifies the BSCAN, which is responsible for transmitting the data that gets logged during the test. The synchronization state machine also talks to the comparator and toggles the reset lines on both the golden copy and the DUT until the circuits run in lockstep. The input generation circuit generates input vectors used by the golden copy and DUT circuits.

The BSCAN circuit interfaces with registers that can be read through the JTAG chain. For the experimental setup, the BSCAN register is configured as follows:

- 24 bits of a constant value - 0x43434C;
- 1 bit signifying a system failure;
- 1 bit of a constant value - 0;
• 3 comparison bits signifying a TMR domain failure on the slave (1 bit for each TMR domain);

• 3 synchronization bits signifying if a TMR domain agreed with the majority value propagated from the slave.

The last six bits of the BSCAN word are only applicable to trip-IO designs when the outputs of all three domains are communicated across the FMC coupler card. In a common-IO circuit only the majority value is propagated across the FMC coupler card. An example of a properly working BSCAN word would be 0x43434C15, which would represent that the slave has not failed, but that TMR domain 2 has failed and does not agree with TMR domains 3 and 1. In contrast, if the word 0x43434CAD was read from the device, it would represent that the slave has failed, and that TMR domains 3 and 1 have failed, but TMR domain 2 has not failed.

Failures are observed through a device called the JTAG configuration manager (JCM), which was developed at Brigham Young University (BYU) [73–75]. The JCM was built to handle configuration of FPGA devices using either the JTAG or SelectMap protocols. The JCM can configure the device, read the device status registers, inject faults, scrub, etc. and was specifically built to handle common tasks for fault injection and radiation testing. For this experiment the JCM is connected to the master device via the JTAG port and has access to the slave device on the same chain, since the JTAG chain is extended through the FMC coupler card. When communicating with the TURTLE, the JTAG clock speed is programmed to 50 MHz.

The JCM runs a Linux operating system (OS) which allows it to run software which can be tailored to a specific test. For these experiments, the JCM was configured to run a continuous loop:

1. Read the BSCAN and detect failures;

2. Issue the recovery mechanisms (if a failure was detected);

3. Log the test status.

When the comparator circuit detects a failure, the BSCAN circuit is programmed to save the failure in the BSCAN register until the JCM issues a BSCAN reset that signals that the failure has been logged. When a failure is observed by the JCM, it will attempt four different recovery mechanisms:
1. Issue BSCAN reset (internal reset to the BSCAN circuitry on the master);

2. Full device scrub followed by a BSCAN reset;

3. Device reset (toggle the reset lines);

4. Full reconfiguration.

The JCM will attempt each recovery mechanism in order, and will allot a 1ms period after the completion of a recovery mechanism before attempting the next recovery mechanism. Occasionally, the synchronization state machine on the master is unable to synchronize the master and the slave devices. In this case, the JCM will attempt another full reconfiguration.

It can be difficult to observe a statistically significant number of failures during an experiment, since the SPF and CMF mitigation techniques can increase a circuit’s reliability. When few failures are observed, there is a low confidence in a measured bit sensitivity or cross-section. As the circuit’s reliability increases, so do the confidence intervals, which creates a problem.

To counteract this problem of increasing confidence intervals, multiple experiments can be run in parallel to increase the data collection rate. For example, if five experiments are run in parallel, then data can be collected $5 \times$ faster. The TURTLE was designed so that multiple layers can be stacked on top of each other, as shown in Figure 6.3. Using the stack of five layers, as shown in the figure, is especially effective for neutron radiation testing since beam time is limited.
6.1.1 Benchmark Circuits

Benchmark circuits are needed in order to prove the effectiveness of the SPF and CMF mitigation techniques. However, there are no standard benchmark circuits across the FPGA testing community. Some of the challenges with benchmark circuits are that hardware is difficult to implement, circuits quickly become outdated as hardware scales with Moore’s law and the user needs to know how to correctly stimulate the circuit with input vectors. This has led others to utilize synthetic circuits to test the effectiveness of mitigation techniques on FPGAs.

For example, one previous work implemented an ADD8 and a simple filtering circuit to test the effectiveness of their mitigation techniques [65]. Others have opted to implement a LEON3 softcore processor and then run software benchmarks on the core [22, 58]. The main goal behind any benchmark circuit is to utilize some significant portion of the device.

To prove the effectiveness of the SPF and CMF mitigation techniques, four different benchmark circuits were created. The first of these circuits, the b13, was chosen because it has been tested in previous work [44]. The b13 design comes from the ITC’99 benchmark suite and is a simple finite state machine that interfaces with a weather station. Because the b13 is a small circuit, it has been replicated 256 times to utilize more of the device resources. Because the b13 has been tested in previous work, it was selected as the only circuit tested in radiation due to the limited availability of beam time.

Three other circuits were also used for this work, the md5, sha3 and aes128. The md5 and sha3 circuits are hashing algorithms while the aes128 is an encryption algorithm. They have also been replicated to increase the circuit utilization. Details on these four benchmark circuits, including their implementation metrics, can be found in Appendix F. All of the circuits and all of the TMR variations of the circuits were always clocked at 50 MHz.

6.2 Fault Injection

All of the SPF and CMF mitigation techniques\(^1\) were applied to all of the benchmark circuits and then tested with fault injection. Recall from Chapter 2.3.2 that fault injection is a testing method where upsets are introduced into the configuration memory. After an upset is introduced,

\(^1\)A summary of all of the SPF and CMF mitigation techniques can be found in Appendix G.1.
the circuit behavior is monitored for any failures. The upset is then corrected and a new upset is introduced. This process is repeated until some number of faults have been injected into the circuit or a certain number of failures have been observed.

There are a number of variables in the fault injection algorithm that can be configured differently for the experiment. For this work, the following configuration was used:

- 1 random bit was selected for each injection from the type 0 frames of the configuration memory;
- The same bit can be chosen multiple times during the test;
- The fault was allowed to propagate for 1 ms after the injection and was then corrected;
- If a failure occurred, the TURTLE recovery mechanisms were employed;
- Each circuit was tested with at least 2,000,000 injections or to at least 100 failures;
- When no failures were observed, the circuit was tested to 12,000,000 injections (there are 59,145,600 bits in the configuration memory).

Using this configuration, the TURTLE can be injected at a rate of about 95 faults per second.

The results of each fault injection technique for each circuit and each mitigation technique are plotted in Figure 6.4. The numerical results for these tests can be found in Appendix F. To understand the relative value of each technique, the total number of injections and detected failures are reported in Table 6.1. This table also reports the geometric mean for the mitigation technique’s sensitivities and their improvement. The goal of the geometric means is to provide an idea for which techniques are usually more effective for any general circuit.

The table presents several values/metrics from the fault injection test:

- **Number of Injections** - The number of faults injected into the circuit. For this table, this is the total number of injections across all of the benchmark circuits for the mitigation technique;

- **Number of Failures** - The number of failures that resulted from the injected faults. There can be at most one failure for each injected fault. For this table, this is the total number of failures across all of the benchmark circuits for the mitigation technique;
• **Bit Sensitivity** - The probability that a random bit flip will cause a failure. This is the number of failures divided by the number of injections. Because this table shows the geomean bit sensitivity, the bit sensitivity for each mitigation technique is the geomean average of the bit sensitivities for each benchmark circuit;

• **Improvement** - The bit sensitivity improvement in relation to the unmitigated design. This is calculated by dividing the unmitigated bit sensitivity by the mitigation technique’s bit sensitivity.

By convention, when no failures are detected, it is assumed that the next injection would have caused a failure. This is why in striping, where no failures were detected, the bit sensitivity is a non-zero number.

The main takeaway from this table and plot is that the SPF and CMF mitigation techniques do decrease the sensitivity to SBF. If the circuit fails during any iteration of the fault injection
Table 6.1: Fault Injection Geomean Results

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Bit Sensitivity</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>6,045,542</td>
<td>330,450</td>
<td>$3.80 \times 10^{-2}$</td>
<td>1×</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>6,037,318</td>
<td>16,960</td>
<td>$1.54 \times 10^{-3}$</td>
<td>24.8×</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>6,132,724</td>
<td>3,367</td>
<td>$3.16 \times 10^{-4}$</td>
<td>121×</td>
</tr>
<tr>
<td>Split-IO</td>
<td>8,000,000</td>
<td>5,446</td>
<td>$8.39 \times 10^{-5}$</td>
<td>453×</td>
</tr>
<tr>
<td>Split-clock</td>
<td>6,538,320</td>
<td>778</td>
<td>$1.13 \times 10^{-4}$</td>
<td>337×</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>6,336,939</td>
<td>486</td>
<td>$8.45 \times 10^{-5}$</td>
<td>451×</td>
</tr>
<tr>
<td>ES</td>
<td>9,255,262</td>
<td>313</td>
<td>$2.27 \times 10^{-5}$</td>
<td>1,675×</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>18,000,000</td>
<td>377</td>
<td>$1.63 \times 10^{-5}$</td>
<td>2,340×</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>24,000,000</td>
<td>50,840</td>
<td>$4.56 \times 10^{-4}$</td>
<td>83.4×</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>28,000,000</td>
<td>223</td>
<td>$2.05 \times 10^{-6}$</td>
<td>18,576×</td>
</tr>
<tr>
<td>PCMF</td>
<td>28,000,000</td>
<td>49</td>
<td>$8.18 \times 10^{-7}$</td>
<td>46,511×</td>
</tr>
<tr>
<td>Striping</td>
<td>24,000,000</td>
<td>0</td>
<td>$8.33 \times 10^{-8}$</td>
<td>344,660×</td>
</tr>
</tbody>
</table>

algorithm, it is due to the effects of a single bit because of how the fault injection test is configured for this work. Therefore, the bit sensitivity rate is synonymous with the SBF sensitivity rate.

In addition to the main takeaway, there are a few general trends from the plot and table:

- The ES-PCMF mitigation technique offered the most improvement for common-IO TMR circuits;
- The split-IO and split-clock-PCMF mitigation techniques offer about the same improvement for common-IO TMR circuits;
- No failures were observed during fault injection using the striping mitigation technique.

Since the fault injection results for the mitigation techniques are promising, the next step is to validate these mitigation techniques through radiation testing.

### 6.3 Neutron Radiation Testing

Radiation testing can be used to validate the effectiveness of a mitigation technique in radiation. Unlike fault injection where only SBU effects are tested, radiation testing measures a
circuit’s response to all SEEs. Radiation testing provides context to how effective these mitigation techniques are in the context of other system failures.

Radiation testing can be performed by setting a device on a desk and letting the natural atmospheric radiation irradiate the device. Xilinx has been performing studies like this at various altitudes and locations in a research study called the Rosetta Experiment [76]. This provides incredibly accurate data for the terrestrial environment, but because of the low flux rate, it can take years to collect enough data.

Devices and circuits can instead be tested at an accelerated rate so that data is collected much more quickly. Neutron radiation testing can be done at the Los Alamos Neutron Science Center (LANSCE) beam [42]. The LANSCE beam operates by using a linear accelerator (LINAC) to accelerate protons to 800 MeV which then strike a tungsten spallation target [77]. Several neutron flight paths are generated when the protons strike the tungsten target, shown in Figure 6.5.
Each of the beam paths creates a different neutron energy spectrum, shown in Figure 6.6. This work tested at ICE House I and II, which are on the 30° flight path (4FP30L and 4FP30R). The ICE House I flight path offers a flux rate roughly six orders of magnitude greater than the sea-level flux rate. The energy spectrum of neutrons of the ICE House I flight path is also roughly the same as the terrestrial neutron spectrum, as shown in Figure 6.7 [5]. Ice House II has the same neutron energy spectrum as Ice House I, but it has about a 2× flux because it is closer to the tungsten target.

The mitigation techniques were tested at LANSCE in Ice House I and II on three separate occasions. The DUT was placed within the flight path and the beam was collimated to two inches so that the DUT FPGA chip was exposed to radiation, but the surrounding area (outside of the two inches) was exposed to much less radiation. The same TURTLE setup that was used for fault injection was used for radiation testing, with the same five layer stack.

The experiment is organized into scrub cycles. A scrub cycle is similar to a fault injection iteration, except faults are caused by the neutron beam. Each scrub cycle consists of the following:

- Perform a device readback (about 2.37 seconds);
- Read BSCAN to detect any failures;
- Perform recovery mechanisms if there was a failure;
Figure 6.7: LANSCE Ice House I Neutron Spectrum Compared to Cosmic-Ray Neutron Spectrum [5]

- Scrub any faults in the configuration memory;

- Log current beam fluence.

Logging the beam fluence at each scrub iteration helps prevent data loss when the system unexpectedly crashes while it is unattended.

TMR was applied to the master circuit since the master device is close to the beam. While the upset rate on the master is many orders of magnitude lower than the upset rate of the DUT, it is still higher than normal conditions because of its proximity to the beam. In addition, when a failure occurs, the recent outputs from the slave and master are recorded so that the failure can be attributed to the master or slave device. The TMR and master/slave output logs helps to ensure measured failures are actually due to the slave circuit failing.

For the radiation test, the TURTLE was set up incident to the beam and run at room temperature. The results from the LANSCE neutron tests are shown in Table 6.2 and are graphically plotted in Figure 6.8. There are a number of measurements and calculations that resulted from the radiation test that are presented in the table:
Table 6.2: Neutron Radiation Testing Results

<table>
<thead>
<tr>
<th>TMR Type</th>
<th>Fluence (n/cm²)</th>
<th>Number of Failures</th>
<th>Cross-Section (cm²)</th>
<th>+95% Confidence</th>
<th>-95% Confidence</th>
<th>FIT Sea-Level</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>3.19 × 10¹¹</td>
<td>555</td>
<td>1.74 × 10⁻⁹</td>
<td>1.89 × 10⁻⁹</td>
<td>1.59 × 10⁻⁹</td>
<td>2.26 × 10¹</td>
<td>1×</td>
</tr>
<tr>
<td>Common-IO</td>
<td>8.61 × 10¹⁰</td>
<td>51</td>
<td>5.92 × 10⁻¹⁰</td>
<td>7.58 × 10⁻¹⁰</td>
<td>4.26 × 10⁻¹⁰</td>
<td>7.70 × 10⁰</td>
<td>2.9×</td>
</tr>
<tr>
<td>Split-clock</td>
<td>1.48 × 10¹¹</td>
<td>39</td>
<td>2.64 × 10⁻¹⁰</td>
<td>3.60 × 10⁻¹⁰</td>
<td>1.87 × 10⁻¹⁰</td>
<td>3.43 × 10⁰</td>
<td>6.6×</td>
</tr>
<tr>
<td>ES</td>
<td>2.69 × 10¹¹</td>
<td>19</td>
<td>7.06 × 10⁻¹¹</td>
<td>1.10 × 10⁻¹⁰</td>
<td>4.28 × 10⁻¹¹</td>
<td>9.18 × 10⁻¹</td>
<td>25×</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>4.91 × 10¹¹</td>
<td>33</td>
<td>6.72 × 10⁻¹¹</td>
<td>9.43 × 10⁻¹¹</td>
<td>4.62 × 10⁻¹¹</td>
<td>8.74 × 10⁻¹</td>
<td>26×</td>
</tr>
<tr>
<td>Trip-IO</td>
<td>1.69 × 10¹²</td>
<td>34</td>
<td>2.01 × 10⁻¹¹</td>
<td>2.81 × 10⁻¹¹</td>
<td>1.39 × 10⁻¹¹</td>
<td>2.62 × 10⁻¹</td>
<td>86×</td>
</tr>
<tr>
<td>PCMF</td>
<td>3.98 × 10¹¹</td>
<td>2</td>
<td>5.03 × 10⁻¹²</td>
<td>1.81 × 10⁻¹¹</td>
<td>5.03 × 10⁻¹³</td>
<td>6.53 × 10⁻²</td>
<td>350×</td>
</tr>
<tr>
<td>Striping</td>
<td>1.90 × 10¹²</td>
<td>28</td>
<td>1.47 × 10⁻¹¹</td>
<td>2.13 × 10⁻¹¹</td>
<td>9.79 × 10⁻¹²</td>
<td>1.92 × 10⁻¹</td>
<td>120×</td>
</tr>
</tbody>
</table>

- **Fluence** - the number of high energy neutrons (>10 MeV) the circuit was exposed to per cm²;
- **Number of failures** - the number of failures that occurred while the circuit was exposed to radiation;
- **Cross-section** - the number of failures divided by the fluence;
- **95% confidence** - the 95% confidence intervals for the cross-section (see Chapter 2.3.3);
- **FIT sea-level** - the FIT rate at sea-level (13×cross-section×10⁹);
- **Improvement** - the reduction of the cross-section over the unmitigated design (the unmitigated design cross-section divided by the cross-section of the TMR type).

Since beam time is limited, only the b13 circuit and only some of the mitigation techniques were tested.

The major takeaway from beam testing is that the mitigation techniques improve the cross-section of the circuit. This means that the circuit with mitigation will, on average, survive longer
than those circuits without the mitigation techniques. The improvements are not as high as they were in fault injection, but this is because SBUs are not the only cause of circuit failure, as will be explored in the next section.

In addition, there are several other takeaways from the data:

- The common-IO circuit only showed a $3 \times$ improvement over no mitigation;

- The ES-PCMF was the best mitigation technique for common-IO circuits with an improvement of $26 \times$;

- The trip-IO circuit performed better than all of the mitigation techniques for common-IO circuits;

- PCMF performed better than striping in radiation testing, even though striping performed better than PCMF using fault injection.

The next section will analyze why PCMF performed better in radiation testing than striping.


6.4 Replay of Radiation Test

An FPGA circuit’s radiation resiliency can be improved by altering the circuit’s placement and routing. However, the last section raised an interesting question. Why was PCMF better than striping in radiation testing when striping performed better in fault injection?

To further understand the failures observed during radiation testing, the radiation test logs can be used to perform a “replay” using fault injection. The following is a sample of a scrub cycle taken during a radiation test.

```
1 [STATUS] Failures=>0 Upsets=>7617 Raw=>0x43434C3F RECOVERY MODE=>None Time =>2017-1-20-20:52:7
2 [SCRUB] 0x00000E02:64=>0x00020000 Count=>1
3 [SCRUB] 0x00000E03:64=>0x00010000 Count=>1
4 [STATUS] Failures=>0 Upsets=>7619 Raw=>0x43434C3F RECOVERY MODE=>None Time =>2017-1-20-20:52:9
```

During each iteration, the JCM logs any faults in the configuration memory and whether or not the DUT failed. This process repeats until the end of the radiation test.

Using the information provided by the logs, each scrub cycle where a failure occurred can be “replayed”. A radiation replay is a form of fault injection, but instead of a single bit being injected during each iteration, the upset bits logged during a scrub cycle are all injected during each iteration. If a failure occurs during the replay, then the failure in that scrub cycle was caused by SEUs in the configuration memory. When a failure does not occur during the replay, then the failure was caused by other radiation effects.

If the replay determines that the cause of failure was due to SEUs in the configuration memory, then fault injection can be done at a finer granularity. When fault injection is performed at a finer granularity, every combination of upset bits can be injected to determine which subset caused the failure. For example, suppose that the bits in the scrub cycle shown above caused a failure. The failure could be attributable to both bits, or either one of the bits.

If multiple bits are determined to be the cause of a failure, then further analysis can be used to determine if the multiple bits are the result of an MCU or multiple SEUs. If an MCU is the cause of the failure, then the design failed from a single event. However, if multiple SEUs (SEU accumulation) are the cause of a failure, then the design failed from multiple, independent events.

Determining the difference between SEU accumulation and MCUs is difficult as logically adjacent bits are not necessarily physically adjacent on the device. Statistical methods can be
Table 6.3: Classification of Failures from Radiation Test

<table>
<thead>
<tr>
<th>Design/Failure Type</th>
<th>SBU</th>
<th>MCU</th>
<th>Accumulation</th>
<th>Not Repeatable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>202 (36.4%)</td>
<td>4 (.7%)</td>
<td>4 (.7%)</td>
<td>345 (62.2%)</td>
</tr>
<tr>
<td>Common-IO</td>
<td>20 (39.2%)</td>
<td>7 (13.7%)</td>
<td>3 (5.9%)</td>
<td>21 (41.2%)</td>
</tr>
<tr>
<td>Split-Clock</td>
<td>9 (23.2%)</td>
<td>7 (17.9%)</td>
<td>2 (5.1%)</td>
<td>21 (53.8%)</td>
</tr>
<tr>
<td>ES</td>
<td>1 (5.3%)</td>
<td>6 (31.6%)</td>
<td>3 (15.8%)</td>
<td>9 (47.4%)</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>3 (9.1%)</td>
<td>7 (21.2%)</td>
<td>6 (18.2%)</td>
<td>17 (51.5%)</td>
</tr>
<tr>
<td>Trip-IO</td>
<td>2 (5.9%)</td>
<td>15 (44.1%)</td>
<td>4 (11.8%)</td>
<td>13 (38.2%)</td>
</tr>
<tr>
<td>Striping</td>
<td>0 (0.0%)</td>
<td>18 (64.3%)</td>
<td>6 (21.4%)</td>
<td>4 (14.3%)</td>
</tr>
<tr>
<td>PCMF</td>
<td>0 (0.0%)</td>
<td>1 (50.0%)</td>
<td>1 (50.0%)</td>
<td>0 (0.0%)</td>
</tr>
</tbody>
</table>

Employed to determine which upsets have a high probability of being an MCU. This is done by using the aggregate data across the entire test to observe common upset patterns. Due to the random nature of the test, most upset patterns should only occur a few times, thus patterns that occur many times are highly unlikely to be random. These patterns can be used to identify MCUs for every scrub cycle. This is still an active area of research, but a tool built on these principles has been employed for this work [78].

The results of the radiation replay are presented in Table 6.3. This table shows how many failures can be attributed to SBUs, MCUs and SEU accumulation or other for each TMR design. Other upsets were likely caused by errors in unmonitored areas of the device, such as the user memory. As expected, applying the SPF and CMF mitigation techniques reduces the impact of SBUs with the striping and PCMF designs showing no failures due to SBUs.

The cross-sections from the radiation test can be updated to filter out failures that were caused by SEU accumulation. SEU accumulation is much more likely to happen in the accelerated testing environment because upsets are happening at an accelerated pace. This increases the likelihood of observing multiple, independent events in the same scrub cycle. In a deployed environment where the repair rate easily outpaces the bit upset rate, SEU accumulation is much less likely.

Table 6.4 shows the results of the radiation test, with the failures caused by SEU accumulation filtered out. These results should more accurately reflect what would happen in a real environment where multiple, independent are much less likely. With these results, the PCMF version still shows the greatest cross-section improvement of $688 \times$.
Table 6.4: Neutron Radiation Testing Results Without Accumulation

<table>
<thead>
<tr>
<th>TMR Type</th>
<th>Fluence (n/cm²)</th>
<th>Number of Failures</th>
<th>Cross-Section (cm²)</th>
<th>95% Confidence</th>
<th>95% Confidence</th>
<th>FIT Sea-Level</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>3.19 × 10¹¹</td>
<td>551</td>
<td>1.73 × 10⁻⁹</td>
<td>1.88 × 10⁻⁹</td>
<td>1.58 × 10⁻⁹</td>
<td>2.25 × 10¹</td>
<td>1 ×</td>
</tr>
<tr>
<td>Common-IO</td>
<td>8.61 × 10¹⁰</td>
<td>48</td>
<td>5.57 × 10⁻¹⁰</td>
<td>7.39 × 10⁻¹⁰</td>
<td>4.10 × 10⁻¹⁰</td>
<td>7.25 × 10⁰</td>
<td>3.1 ×</td>
</tr>
<tr>
<td>Split-clock</td>
<td>1.48 × 10¹¹</td>
<td>37</td>
<td>2.50 × 10⁻¹⁰</td>
<td>3.45 × 10⁻¹⁰</td>
<td>1.76 × 10⁻¹⁰</td>
<td>3.25 × 10⁰</td>
<td>6.9 ×</td>
</tr>
<tr>
<td>ES</td>
<td>2.69 × 10¹¹</td>
<td>16</td>
<td>5.95 × 10⁻¹¹</td>
<td>9.67 × 10⁻¹⁰</td>
<td>3.49 × 10⁻¹¹</td>
<td>7.73 × 10⁻¹</td>
<td>29 ×</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>4.91 × 10¹¹</td>
<td>27</td>
<td>5.50 × 10⁻¹¹</td>
<td>7.98 × 10⁻¹¹</td>
<td>3.63 × 10⁻¹¹</td>
<td>7.15 × 10⁻¹</td>
<td>31 ×</td>
</tr>
<tr>
<td>Trip-IO</td>
<td>1.69 × 10¹²</td>
<td>30</td>
<td>1.78 × 10⁻¹¹</td>
<td>2.54 × 10⁻¹¹</td>
<td>1.20 × 10⁻¹¹</td>
<td>2.31 × 10⁻¹</td>
<td>97 ×</td>
</tr>
<tr>
<td>PCMF</td>
<td>3.98 × 10¹¹</td>
<td>1</td>
<td>2.51 × 10⁻¹²</td>
<td>1.41 × 10⁻¹¹</td>
<td>2.51 × 10⁻¹³</td>
<td>3.27 × 10⁻²</td>
<td>688 ×</td>
</tr>
<tr>
<td>Striping</td>
<td>1.90 × 10¹²</td>
<td>22</td>
<td>1.16 × 10⁻¹¹</td>
<td>1.75 × 10⁻¹¹</td>
<td>7.26 × 10⁻¹²</td>
<td>1.51 × 10⁻¹</td>
<td>149 ×</td>
</tr>
</tbody>
</table>

Table 6.5: Resources Affected by MCUs

<table>
<thead>
<tr>
<th>Resources</th>
<th>Striping</th>
<th>PCMF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of Bits</td>
<td>Percentage</td>
</tr>
<tr>
<td>Routing</td>
<td>52</td>
<td>96.3%</td>
</tr>
<tr>
<td>LUT Contents</td>
<td>2</td>
<td>3.7%</td>
</tr>
</tbody>
</table>

One surprising result that requires further analysis is the difference between striping and PCMF. In fault injection, striping shows more improvement than PCMF, but in radiation testing, PCMF showed more improvement than striping. The MCU results from Table 6.3 showed that striping was much more sensitive to MCUs than PCMF was. The configuration bits associated with the MCU failures for both of these designs can be analyzed to observe what resources were most affected, which are reported in Table 6.5. This MCU analysis shows that the routing network is by far the most susceptible resource to MCUs. The striping design could be more sensitive to MCUs because it utilizes more routing nodes.

To ensure that the striping is more sensitive to MCUs, an MCU distribution for each experiment can be calculated. An MCU distribution shows the likelihood of a n-sized MBU occurring
Table 6.6: Number of Bit Upsets per SEU

<table>
<thead>
<tr>
<th>MCU/TMR Type</th>
<th>1-bit (SBU)</th>
<th>2-bit</th>
<th>3-bit</th>
<th>4+-bit</th>
<th>Mean Upsets Per Scrub</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>68.8%</td>
<td>25.2%</td>
<td>2.1%</td>
<td>3.8%</td>
<td>2.05</td>
</tr>
<tr>
<td>Common-IO</td>
<td>73.8%</td>
<td>21.6%</td>
<td>1.8%</td>
<td>1.9%</td>
<td>1.74</td>
</tr>
<tr>
<td>Split-clock</td>
<td>73.7%</td>
<td>21.8%</td>
<td>1.8%</td>
<td>2.7%</td>
<td>1.75</td>
</tr>
<tr>
<td>ES</td>
<td>73.8%</td>
<td>21.4%</td>
<td>1.9%</td>
<td>2.9%</td>
<td>1.76</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>74.2%</td>
<td>21.1%</td>
<td>1.8%</td>
<td>2.8%</td>
<td>1.75</td>
</tr>
<tr>
<td>Trip-IO</td>
<td>72.3%</td>
<td>21.9%</td>
<td>2.1%</td>
<td>3.1%</td>
<td>1.79</td>
</tr>
<tr>
<td>PCMF</td>
<td>69.9%</td>
<td>23.9%</td>
<td>2.5%</td>
<td>3.8%</td>
<td>1.85</td>
</tr>
<tr>
<td>Striping</td>
<td>71.2%</td>
<td>23.1%</td>
<td>2.4%</td>
<td>3.3%</td>
<td>2.19</td>
</tr>
<tr>
<td>Total</td>
<td>72.2%</td>
<td>22.5%</td>
<td>2.2%</td>
<td>3.2%</td>
<td>1.92</td>
</tr>
</tbody>
</table>

when an SEU occurs and can be calculated using the MCU patterns previously discussed. The MCU distribution is broken down to show how many of the events caused single bit upsets, two bit upsets, three bit upsets, etc., and are shown in Table 6.6.

The distribution shows that during the radiation test, the striping circuit on average had fewer MCUs than the PCMF circuit did during testing. This means that the striping circuit had fewer MCU events on average than the PCMF test, but the striping circuit still showed a higher susceptibility to MCUs. It would appear that the striping mitigation technique is more susceptible to MCUs because MCUs affect the routing network, as shown in Table 6.5 and the striping technique using more routing nodes, as was shown in Tables 5.10 and 5.12 back in Sections 5.2.3 and 5.2.5.

The goal of this dissertation was to reduce SPF and CMF, not MCUs. Confirming the hypothesis that striping is more sensitive to MCUs because it utilizes more routing nodes than PCMF is outside the scope of this work. What this analysis has shown is that MCUs were the limiting factor for why striping and PCMF did not improve the radiation cross-section more than what was measured.

6.5 Summary

The SPF and CMF mitigation techniques described in chapter 5 were tested using both fault injection and radiation testing to show whether or not they improved certain reliability metrics. The
fault injection tests showed that the mitigation techniques did improve the single bit sensitivity. Striping was the best technique in fault injection, since no failures were observed during fault injection testing.

While fault injection testing is useful, the mitigation techniques also needed to be tested in radiation, where the circuit is exposed to all SEEs. The radiation test also showed that the mitigation techniques were able to improve the circuit cross-section. However, unlike fault injection, the PCMF mitigation technique was able to reduce the cross-section more than the striping technique. Further analysis showed that the striping technique was more sensitive to MCUs than the PCMF technique, which should be further studied in future work.

The goal of this dissertation was to lower the SBU failure rate of TMR circuits on FPGAs. Using a radiation replay, the failures observed during radiation testing were classified by cause of failure. When the mitigation techniques were applied, all circuits showed less susceptibility to SBUs. Thus, the mitigation techniques achieved the goal of this dissertation.
CHAPTER 7. CONCLUSION

While TMR with repair does improve the radiation resiliency of the FPGA circuit, it is not as effective as it could be. There are single bits within the configuration memory that when upset can cause instantaneous TMR circuit failure. In essence, these bits represent a failure mechanism that is not repairable. These failures cannot be avoided by increasing the repair rate, but can only be avoided by providing specific mitigation for them.

This dissertation showed that the radiation cross-section could be improved for circuits when non-complete TMR is applied, such as common-IO. By applying the split-clock, split-IO and PCMF techniques together (ES-PCMF), the cross-section improved an order of magnitude over common-IO TMR. While SPF will always exist in these circuits, this work showed that most SPF can be successfully mitigated.

The results of applying mitigation for CMF showed an overall cross-section improvement of nearly $700 \times$ over no mitigation. The improvement was only limited by the effects of MCUs, an SEE effect outside the scope of this dissertation. The goal of mitigating failures caused by single bits was achieved by this work.

By accomplishing the goal of reducing the SBU sensitivity of TMR circuits on FPGAs, this work contributed four novel contributions to the state of the art:

- Low-level theory into how SBUs cause failure in the FPGA architecture;
- Multiple mitigation techniques to address both SPF and CMF;
- Extensive fault injection and radiation testing results to prove the efficacy of the mitigation techniques;
- The development of Markov chains to show the theoretical impact of single bit failures on TMR with repair systems (shown in Appendix D).
7.1 Future Work

There are a number of areas where this worked can be expanded upon. These areas of future work range from further tool development for SPF and CMF to additional radiation testing on 7-Series and other FPGA device generations.

7.1.1 Early Split Custom Routing

The SPF techniques presented in this work focus on constraining the vendor’s tool such that SPF is reduced. It should be possible to achieve similar reliability performance by doing custom routing. Recall from Chapter 5 that buffers and pseudo-buffers were inserted into the design netlist to achieve an early split on the input nets. Instead of inserting buffers, the input nets could be split early on by a custom routing tool and then fed into each domain. This would allow for much more design space exploration, but might suffer from increased timing delays and routing congestion if the custom router is not as effective as the vendor’s tool.

7.1.2 Domain Placement

When a common clock is shared between the TMR domains, it is likely that clocked cells from multiple domains are placed onto the same site. This would make it impossible to perform early split custom routing on the clock net, as the clock is shared among cells placed on the same site. In order to be able to perform an early split custom routing on the clock nets, the clocked cells from each domain would first need to be separated and placed onto different sites. This would allow an early split technique custom routing to be performed on a clock net without the need for multiple clock buffers. This would also add spatial separation between the TMR domains that should also improve its reliability.

7.1.3 SPF In Specialized Resources

This work exclusively analyzed SPF in the context of common-IO systems, but noted that SPF can occur in other locations on the device when specialized resources are not triplicated. An extension of this work could study the impacts of SPF when untriplicated resources are utilized,
such as untriplicated MGTs, DSPs, BRAM, etc., and could study if any of the proposed techniques in this work could also be used in those circuits.

### 7.1.4 Techniques for Site CMF

As detailed in Chapter 5, the PCMF technique only addressed clock CMF while striping addresses both clock CMF and site CMF. The PCMF tool could become more effective by addressing site CMF. This would be similar to the domain placement technique just described, but would be done for LUTs in SLICEM tiles. Fractured LUTs that contain the contents of two TMR domains could be identified and then re-placed to address site CMF.

### 7.1.5 MCU Mitigation Techniques

The analysis from the previous chapter showed that most of the failures in the CMF mitigated TMR schemes are caused by MCUs. The next logical step to improve TMR’s radiation reliability would be to address the MCU failures. To do this would likely require studies into the most common types of MCUs that occur for a given FPGA generation/family followed by a study of which resources these MCUs would affect. Some form of placement and routing changes could then be performed to avoid bits that are likely to be affected by MCUs.

### 7.1.6 Heavy Ion Testing

In this work only neutron testing was used to measure the effectiveness of the proposed techniques. These are sufficient to show that the techniques are effective in a terrestrial environment, but do not adequately show how these techniques would perform in a space environment. Heavy ion testing would be needed to demonstrate this. The proposed techniques should offer some reliability improvement in heavy ions, but the exact amount of improvement would need to be measured.
7.1.7 Applicability to Other Devices

Throughout this work, the impact of SBF and the mitigation techniques were targeted specifically for Xilinx 7-Series FPGAs. While the low-level details on how SBF occurs on FPGAs are specific to the 7-Series family of devices, the cause of CMF should translate to other device generations. The routing mux details shown for clock CMF are general from a Xilinx patent, and are not particular to a single device, generation or family. The Virtex-4,5,6 and UltraScale families can all have multiple clocks in the same tile, which likely implies that these devices are also susceptible to clock CMF.

The CMF mitigation techniques should also be applicable to other device generations. The exact details on how to implement the PCMF algorithm may vary from generation to generation, but the general principles of the algorithm should carry over. The next steps for this work would be to apply PCMF and the other techniques on a Xilinx UltraScale device and then study the technique’s effectiveness on other devices and generations.

7.1.8 Algorithm Improvement

For a majority of the benchmark circuits, the proposed PCMF algorithm was effective in both resolving CMF and in the amount of time it took the algorithm to complete. A notable exception was the md5 circuit. The difference in the md5 circuit from the rest of the benchmark circuits was the presence of many carry chains. A swap involving a carry chain would invoke other swaps to preserve the carry chain dependency, but these swaps could displace another carry chain continuing the cycle. It is not the presence of carry chains that defeat the algorithm, as the aes128 and sha3 both had carry chains, but rather the presence of many carry chains that defeat the PCMF algorithm. This should be further explored to examine if potential algorithms changes are warranted.

7.2 Concluding Remarks

This work has contributed novel contributions to the understanding of TMR, FPGA-based systems. This work provided low-level theory into how single bits can cause TMR circuit failure in FPGAs, offered several mitigation techniques to address these failures and provided extensive
fault injection and neutron radiation testing data to validate the effectiveness of these techniques. In addition to these novel contributions, this work has explored the shortcomings of these techniques, particularly how MCUs cause these circuits to fail. While not all circuits will need the level of reliability achieved in this work, this work has provided the designer with new strategies to utilize to implement high reliability FPGA-based systems.

The mitigation techniques of this work can be applied to future systems intended for the space environment. By improving the reliability of FPGAs, the engineer is more enabled to choose an FPGA for the FPGA’s high-performance computing capabilities and its radiation tolerance. FPGAs can be used to usher in the next age of space computing.

The results of this work can impact other high-reliability systems. The low sea-level FIT rate is ideal for widely deployed systems, such as self-driving cars or commercial aircraft. FPGAs can even be deployed in high energy physics experiments, such as in the large hadron collider (LHC) at CERN. The future is bright with possibilities for the use SRAM FPGAs in society.
APPENDIX A. GENERAL FPGA CAD

The FPGA CAD flow generally starts from source HDL code, e.g., VHDL or Verilog, which is then interpreted and eventually compiled into the FPGA bitstream that programs the device, as shown in Figure A.1. There is an optional step at the beginning of the flow for high level synthesis (HLS). In HLS, the designer first starts with some higher level code, usually C or C-like code, which is then compiled into HDL code. The CAD process would then follow the same flow as if the user had started with HDL code [79].

There are several distinct steps within the flow as shown in the figure. The HDL code is first synthesized into a design netlist, which can be represented with an electronic design interchange format (EDIF) encoding. This textually represents the circuit schematic. The EDIF file along with the FPGA device information is then input into the pack and place step which assigns locations for all of the cells in the netlist. After pack and place, each net in the circuit is routed. The output of both the pack/place and route steps is a implemented design, contained in a design checkpoint file (DCP). The DCP file contains information about the netlist and the physical implementation for the placement and routing of the cell/snets. After routing, the flow ends by creating a bitstream which can be used to program the device.

The goal of this appendix is to provide a brief overview of each of the steps in the CAD flow and discuss how TMR is implemented within this process. Many of these steps utilize heuristics to find good, but not necessarily optimal solutions.

Figure A.1: FPGA CAD Flow
A.1 Synthesis

FPGA synthesis is the process of converting HDL code into device specific components which can be physically implemented on an FPGA device. This consists of two parts, HDL synthesis followed by technology mapping. During the synthesis step, the HDL is converted from code into a gate-level netlist. This netlist is then run through several optimizations to eliminate redundancy, dead circuitry, etc. Afterwards, this netlist is technology mapped onto the FPGA. This gate-level netlist is converted into a primitive-level netlist, comprised of components that exist, and can be placed on the targeted FPGA device. The goal of this step is to derive a solution that uses the fewest number of cells and nets [80].

A.2 Pack/Place

The placement process takes two steps, packing and then placement.

A.2.1 Packing

The cells from the synthesis process can be directly placed on the device, but for large devices and large circuits it may be difficult to find a good solution in a reasonable amount of time. Packing is first performed to group cells together which can be placed on sites. For example, in the 7-Series devices, each CLB slice consists of 4 LUT6’s, a carry chain and 8 flip-flops [8]. Up to that many cells may be packed into a single CLB slice. One goal of packing is to group similar cells together to reduce the amount of routing that must route through the general interconnect network.

A.2.2 Placement

After packing, the placer takes the packed sites and places them on the device. The placer is working at the site level, and not at the cell level. This means that when a site is moved, all of the cells at that site are also moved. The first goal of the placer is to produce a routeable design given the limited resources on the device. This is done two-fold, by reducing the potential congestion between routing switchboxes and reducing the overall design half perimeter wire length (HPWL). The HPWL of the design is the sum of half of the perimeter of every bounding box for every net in
the design. The second goal of the placer is to produce a placement that is likely to meet the user defined timing constraints.

Placement has traditionally be implemented using simulated annealing [81]. Simulated annealing randomly samples the solution space to find a local optima that are comparable to the global optima. Finding a good solution in simulated annealing can take large amounts of time. To combat this, analytical placers have been used to find a good initial solution, which could then followed by a little simulated annealing to further optimize the design [82].

A.3 Routing

After each cell has a physical location on the device, each net can be routed. With limited resources, the router must both decide how each net can be routed and decide which net gets to use each resource. In a highly utilized design, it is highly likely that many nets will compete for the same routing resources, but only one net can use any resource in the final implementation. To further complicate the problem, some nets are more critical to the overall circuit timing than other nets and should have priority when routing. The router must balance all of these constraints. Most recent routers are based on the PathFinder negotiated cost algorithm [83].

A.4 TMR

Netlist-level TMR is implemented after the synthesis step, but before the placement step. This work uses the BL-TMR tool [49] which implements feedback TMR [56]. Netlist TMR takes the EDIF netlist as an input and outputs a different EDIF netlist with TMR inserted. This flow is shown in Figure A.2. After TMR has been applied, the tool flow continues from the placement step all the way to bitstream. The variations of TMR proposed in this work operate at different locations throughout the flow.
A.4.1 Constrained Variations

In the constrained variations of TMR, namely, split-io, split-clock, early-split and striping, the additional TMR modifications are enforced during their implementation. For example, in striping, the placer is constrained to place each TMR domain in specific columns of the device. The tool is allowed to run as it sees fit to implement the design within these constraints.

A.4.2 Separate CAD Flow Variations

PCMF and RCMF operate as separate CAD flow steps. These require additional steps to implement. In PCMF, placement is allowed to occur and then after placement has finished, PCMF slightly alters upon that placement. Similarly, RCMF, as explained in Appendix E, operates after the normal routing step and slightly alters upon the routing. Open source research CAD tools are needed to implement PCMF and RCMF.

A.5 Open Source CAD Tools

Open source FPGA CAD tools allow for the manipulation of designs outside of the typical design flow. These tools have the ability to completely implement a design through the routing step outside of the vendor tools. In order to generate the bitstream, however, the implementation must be loaded into the vendor tool. Two different tools were used for this work, RapidSmith2 [62, 84] and RapidWright [7].

RapidSmith was created first as part of the work for a dissertation [85] and was later updated to work with new generation devices and branded as RapidSmith2. The creator of RapidSmith started working for Xilinx and created RapidWright, using some of the same code originally used in RapidSmith. Both of these tools are quite similar, with a few implementation differences. The algorithm for PCMF is the same across both.

The major differences between the two are how they interface with Vivado. RapidSmith2 interfaces with Vivado through TCL, using the open-source Tincr library [63], as shown in Figure A.3 [6]. Tincr has the ability to save a design in a Tincr checkpoint (TCP), which uses XDC files to save the implementation state. These can be read in by RapidSmith2 which then exports
a RapidSmith2 checkpoint (RSCP). Tincr can import the RSCP into Vivado. RapidWright interfaces with Vivado using DCP files which can be read and written by RapidWright, as shown in Figure A.4 [7].
This appendix is intended to provide a basic overview of the Xilinx 7-Series architecture that is pertinent to this work. The CLB logic tile contains two logic slices, as shown in Figure B.1. There is a left and right slice. Top and bottom slices do not exist, they are just drawn that way for the figure. Tiles are laid out in a grid structure as shown in the figure. Routing tiles are not shown in the figure.

Each slice has a carry chain dependency to a slice in the tile above and below it. The carry chain is used to perform fast arithmetic increment operations. The carry chain introduces a placement constraint for a circuit. If there is a carry chain between two slices in a circuit, the slices in that circuit must be placed in the vertical alignment dictated by the circuit’s netlist.

There are two types of logic slices, a SLICEL and a SLICEM. An example of a SLICEL is shown in Figure B.2 and an example of a SLICEM is shown in Figure B.3. Both types of logic slices...
slices contain four LUT6’s and eight flip-flops, along with some incrementing logic for the carry chain. A LUT6 is a 6-input LUT, which can implement any 6-input combinational logic function. The LUT6 can be fractured to form two LUT5’s.

The difference between a SLICEL and a SLICEM has to do with the possible LUT configurations. In a SLICEL, a LUT can only be a LUT. In a SLICEM, a LUT can instead be configured as small memory. This memory could be a 64-bit RAM or it could be a shift register lookup (SRL). There would be extra configuration bits associated with a SLICEM that would control whether a LUT is currently configured as a LUT or a memory.

There are some signals within a slice that are shared by all of the cells in that slice. These signals are the write enable (WE/WEN), the clock (CK/CLK), the chip enable (CE) and the set/reset (SR). This creates a packing constraint as all the cells that are packed into a slice must share these signals.

There is a CLB switchbox (routing tile) adjacent to each CLB logic tile. This switchbox contains the PIPs that drive the CLB logic tile as well as PIPs that connect to wires in other CLB switchboxes. All nets entering and leaving the CLB logic tile must pass through this switchbox, with the exception of any carry chain signal.

There is special routing for the global clock network. The clock network is designed in a way to minimize clock skew across the device. Clocks can be routed on the general routing network, but this is generally not advised as it can significantly increase clock skew. Generally, when a clock routes to a logic slice, it is routed off the clock tree directly into the input wire for the slice, as shown in Figure B.4. In this example, two clocks are routed from the global clock tree to the two logic slices in the tile. A local routing wire (from the general routing network) could also be routed to the clock pin of the slice. More information about the Xilinx 7-Series FPGA architecture can be found in [8, 86].
Figure B.2: Diagram of a SLICEL from [8]
Figure B.3: Diagram of a SLICEM from [8]
Figure B.4: Routing a Clock to a Logic Slice
APPENDIX C. RELIABILITY METRIC DERIVATIONS AND TMR MODELING

Graphical models are often employed to theoretically determine the reliability and MTTF of proposed mitigation strategies. These graphical models provide a visual representation on what is happening in the system and are backed with mathematics which can symbolically represent these graphs. While there are usually simplifying assumptions made for theoretical models, they are extremely useful for determining potential benefits and limitations of a possible mitigation technique.

These mitigation technique models build off of the metrics of a single module (single circuit). The first metric of interest is the hazard rate of the module. The hazard rate represents how the failure rate of a module changes over time. For radiation environments with a constant flux, the hazard rate would be constant, meaning that the module has an equal chance of failure throughout its lifetime. By convention, the constant hazard rate is represented by the greek letter $\lambda$, which can be estimated by observing the number of failures $n$ over a time period $t$ and then dividing $n/t$. The hazard rate $z(t)$ itself does not provide insight into how likely a module is to still be operating at any time $t$, that would the reliability function $r(t)$. The reliability of a module can be calculated from the hazard with the following equation

$$r(t) = e^{-\int_0^t z(\xi) d\xi}.$$  \hspace{1cm} (C.1)

For a constant hazard rate $r(t) = e^{-\lambda t}$, an exponential failure rate. The MTTF of the system can then be calculated by taking the expected value of $r(t)$, that is,

$$\text{MTTF} = \int_0^\infty r(t) dt.$$  \hspace{1cm} (C.2)

For the constant hazard rate MTTF=$1/\lambda$ [87].
For many systems, an MTTF of $1/\lambda$ is not sufficient. Mitigation strategies connect many modules together to attempt to increase the overall reliability of the system. Mathematical models are used to theoretically determine what the reliability characteristics of these systems would be to evaluate and compare them. These models can be powerful to determine potential flaws in a fault-tolerant technique, but usually stop short at completely modeling the system as many simplifying assumptions are made. The goal of this appendix is to provide background on how to derive the reliability function $r(t)$ and the MTTF from a mathematical model. There are two models that are typically used, block diagrams and Markov chains.

### C.1 Block Diagrams and Fault Trees

Block diagrams and fault trees can be used to show how a system succeeds, or fails, using a graph. Using either a block diagram or fault tree is acceptable and are equivalent mathematically. Consider the Simplex system in Figure C.1. The success state is represented by the final circle. To get to it, module $x_1$ must be operating correctly$^1$. Using this model, the reliability of the Simplex system would be the same as the module failure rate, i.e., $r(t) = e^{-\lambda t}$.

For an example of a more complex system, consider the TMR system in Figure C.2. There

---

$^1$The fault tree model would use the same diagram. The fail state can only be reached after the single module has failed.
are many ways to “succeed” in this system. As long as at least two of the three modules function correctly then the system also functions correctly. There are four paths possible for a correctly operating system:

- Modules $x_1$, $x_2$ and $x_3$ all operate correctly (i.e. no module failures).
- Modules $x_1$ and $x_2$ operate correctly and module $x_3$ has failed.
- Modules $x_1$ and $x_3$ operate correctly and module $x_2$ has failed.
- Modules $x_2$ and $x_3$ operate correctly and module $x_1$ has failed.

The reliability of the system can be calculated by calculating the reliability for each path using each modules reliability and failure rates, which are

$$r(t)_{\text{Module}} = e^{-\lambda t},$$
$$F(t)_{\text{Module}} = 1 - e^{-\lambda t}.$$  

The reliability of the system would then be

$$r(t)_{\text{TMR}} = (e^{-\lambda t})(e^{-\lambda t})(e^{-\lambda t})$$
$$+ (e^{-\lambda t})(e^{-\lambda t})(1 - e^{-\lambda t})$$
$$+ (e^{-\lambda t})(e^{-\lambda t})(1 - e^{-\lambda t})$$
$$+ (e^{-\lambda t})(e^{-\lambda t})(1 - e^{-\lambda t})$$
$$= 3e^{-2\lambda t} - 2e^{-3\lambda t}. \tag{C.3}$$

The fault tree model leads to the same reliability equation, but it is calculated differently. The TMR fault tree diagram is the same as the TMR block diagram as shown in Figure C.2 (but the diagram could be different for other fault-tolerant techniques). This is because the failure condition is the same as the success condition, if at least two of the three modules have failed, then the system has failed. This means there are four possible paths for the failed system:

- Modules $x_1$, $x_2$ and $x_3$ have all failed.
- Modules $x_1$ and $x_2$ have failed and module $x_3$ is operating correctly.
• Modules $x_1$ and $x_3$ have failed and module $x_2$ is operating correctly.

• Modules $x_2$ and $x_3$ have failed and module $x_1$ is operating correctly.

The probability of failure for the system can be calculating using the reliability and failure rates of a single module.

$$F(t)_{TMR} = (1 - e^{-\lambda t})(1 - e^{-\lambda t})(1 - e^{-\lambda t})$$

$$+ (1 - e^{-\lambda t})(1 - e^{-\lambda t})(e^{-\lambda t})$$

$$+ (1 - e^{-\lambda t})(1 - e^{-\lambda t})(e^{-\lambda t})$$

$$+ (1 - e^{-\lambda t})(1 - e^{-\lambda t})(e^{-\lambda t})$$

$$= 1 - 3e^{-2\lambda t} + 2e^{-\lambda t}$$  \hspace{1cm} (C.4)

The reliability of the system can be calculated by taking one minus the probability of failure for the system

$$r(t)_{TMR} = 1 - F(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t}.$$ \hspace{1cm} (C.5)

Depending on the system, it may be easier to calculate the success paths or the failure paths, but either the block diagram or the fault tree can be used to calculate the system reliability.

Calculating system reliability through graphical methods is powerful as it allows the module blocks to each have different reliabilities/hazard rates. These methods are also powerful because of their simplicity. The reliability of the system is the sum of the reliability of its different paths. This can be very effective for systems where the success/failure paths are well known.

One of the main drawbacks to graphical methods, however, is the inability to model all of the fault-tolerant techniques available to the reliability engineer. One example of this is repair. A repair mechanism would represent a feedback mechanism in the graph, which distorts the notion of success and failure paths. This motivates the need for a different graphical model which can accurately characterize these other mitigation techniques.

### C.2 Markov Chain

Continuous time Markov chains can be used to model more complex systems, such as those systems with repair mechanisms. A Markov chain is composed of states and transitions between
those states. At each time instance $\Delta t$ the system can transition into a new state at a specified rate. Possible transitions from any state $S_i$ to another state $S_j$ are specified in the chain as an arc between those two states. An example of the Simplex system as a Markov chain is shown in Figure C.3. In this example, the transition rate between states $S_0$ and $S_1$ is $\lambda$. If at time $t$ the system is in state $S_0$, then the probability that the system will transition into $S_1$ at time $t + \Delta t$ is $\lambda \Delta t$. Conversely, the probability that the system remains in state $S_0$ is $1 - \lambda \Delta t$. For simplicity, usually only the transitions to other states are shown in the graph and it is implied that the system transitions back into the same state if it did not transition during the time period $\Delta t$. Also for simplicity, the variable $\Delta t$ is usually omitted, but implied.

Figure C.4 shows the same system as Figure C.3, but with the simplifications made. Using this Markov chain, the transition rate matrix for the system can be derived.

$$T = \begin{bmatrix} -\lambda & \lambda \\ 0 & 0 \end{bmatrix}$$

The transition rate matrix can be altered into the frequency domain using the following equation:

$$A = [sI - T] = \begin{bmatrix} \lambda + s & -\lambda \\ 0 & s \end{bmatrix}.$$
The probability of being in any state can be derived using $A$ by taking $A$’s inverse. The frequency domain solution for each probability state can be found on the top row of the matrix

$$ P_i(s) = A^{-1}(1, i). $$

The time-domain probability can then be calculated for each state by taking the inverse Laplace transform

$$ p_i(t) = \mathcal{L}^{-1}\{P_i(s)\}. $$

The reliability $r(t)$ of the system can be calculated from these probabilities. For any system with an absorbing or fail state, the reliability is the sum of all non-absorbing states. That is, if state $j$ is the absorbing state of a system, then

$$ r(t) = \sum_{i \neq j} p_i(t). \quad (C.6) $$

Note that the probability of being in state in the system at time $t$ is 1, that is

$$ \sum_i p_i(t) = 1. \quad (C.7) $$

Using Equation C.7 and rearranging Equation C.6 means that

$$ r(t) = 1 - p_j(t), \quad (C.8) $$

where $j$ is the absorbing/failure state of the system. Systems modeled by complex Markov chains often use Equation C.8 because only one state probability has to be calculated.

There are two commonly used methods to calculate the mean time to failure (MTTF) of the system. The first is to integrate the reliability for all time, i.e.,

$$ \text{MTTF} = E[r(t)] = \int_{t=0}^{\infty} r(t) dt. $$
The second method is to calculate the MTTF directly from the Laplace domain. The Laplace transform of the previous equation is

\[ \mathcal{L}\{\text{MTTF}\} = \mathcal{L}\left\{ \int_0^t r(\tau)d\tau \right\} = \frac{R^*(s)}{s}, \quad (C.9) \]

where \( R^*(s) \) is \( \mathcal{L}\{r(\tau)\} \). The final value theorem states

\[ \mathcal{L}\{\lim_{t \to \infty}\} = \lim_{s \to 0} s F(s). \quad (C.10) \]

The time domain MTTF equation can be rewritten to transform it into the Laplace domain using the final value theorem

\[ \text{MTTF} = \int_{t=0}^{\infty} r(t) dt = \lim_{t \to \infty} \int_{t=0}^{t} r(t) dt = \lim_{s \to 0} R^*(s). \quad (C.11) \]

Because the Laplace transform is a linear function, Equation C.6 can be used to calculate \( R^*(s) \)

\[ R^*(s) = \mathcal{L}\left\{ \sum_{i \neq j} p_i(t) \right\} = \sum_{i \neq j} P_i(s), \quad (C.12) \]

where \( j \) is the absorbing state of the system. This means the MTTF can be calculated directly from the Laplace domain using the following equation:

\[ \text{MTTF} = \lim_{s \to 0} \sum_{i \neq j} P_i(s). \quad (C.13) \]

For the Simplex System:

\[ A^{-1} = \begin{bmatrix} \frac{1}{\lambda+s} & \frac{\lambda}{s(\lambda+s)} \\ 0 & \frac{1}{s} \end{bmatrix} \]

\[ P_0(s) = \frac{1}{\lambda+s}, \quad P_1(s) = \frac{\lambda}{s(\lambda+s)} \]

\[ p_0(t) = e^{-\lambda t}, \quad p_1(t) = 1 - e^{-\lambda t} \]
C.2.1 Traditional TMR Model

A popular mitigation strategy for protecting any system against failure is triple modular redundancy (TMR). In a TMR system, the Simplex system is replicated three times and a voter is employed to select the majority output. Using this scheme, the TMR system will continue operating correctly as long as two of the three redundant modules are operating correctly. The TMR system will only fail when two or more of the redundant modules have failed.

The TMR system can be modeled using 3 states: normal \( S_0 \), impaired \( S_1 \) and failed operation \( S_2 \), shown in Figure C.5. In normal operation \( S_0 \) all three of the sub-systems are operating correctly and there is no failure in any of the modules. In impaired operation \( S_1 \), one of the modules has failed and is operating incorrectly, but the other two modules are still operating correctly. In the failure state \( S_2 \), the system has failed as at least two of the modules have failed and are operating incorrectly.

The transitions between each of these states occurs at different rates. Since any of the three modules could fail in the first state, the transition from \( S_0 \) to \( S_1 \) happens at three times the failure rate of one module, making the transition rate \( 3\lambda^2 \). From \( S_1 \) to \( S_2 \), only two of the modules can fail, hence a transition rate of \( 2\lambda^2 \).

\[ r(t) = p_0(t) = e^{-\lambda t} \]

\[ MTTF = \frac{1}{\lambda}. \]

2\( \lambda \) is used to denote the failure of a single system. Adding redundancy, such as triplication, would change \( \lambda \) for that system, i.e. \( \lambda_{SIM} \neq \lambda_{TMR} \). However, it is generally assumed that \( \lambda_{SIM} \approx \lambda_{TMR} \). For the remainder of this work, no distinction will be made between the slightly different \( \lambda \) for each system so that attention is given to the equations themselves.
Using this Markov model, the state transition matrix can be derived as

\[
T = \begin{bmatrix}
-3\lambda & 3\lambda & 0 \\
0 & -2\lambda & 2\lambda \\
0 & 0 & 0
\end{bmatrix},
\]

which can be used to derive the reliability and MTTF equations:

\[
r(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t},
\]

\[
MTTF = \frac{5}{6} \times \frac{1}{\lambda} = \frac{5}{6} MTTF_{SIM}.
\]

The MTTF improvement of TMR is only 5/6. This implies that TMR actually decreases the MTTF over a Simplex system.

C.2.2 N Modular Redundancy

TMR only replicates a system three times, but any number of replications is permissible. Generally, any sort of modular redundancy is referred to as \( n \) modular redundancy (NMR), where \( n \) refers to the amount of redundancy. In TMR, \( n = 3 \), but other popular NMR techniques are 5 (QMR) and 7 (SMR). \( n \) is usually an odd number, so that there is always a simple majority that can be used for voting. If an even number is used, additional logic must be used to distinguish between error masking and error detection. When an equal number of modules disagree on the system output, the system can only detect that a failure has occurred, but cannot mask it. \( \gamma \) represents the maximum number of modules that can fail before NMR will fail. For an odd modular redundancy \( n \)

\[
\gamma = \left\lfloor \frac{n}{2} \right\rfloor + 1,
\]

or the closest integer number less than half of \( n \). For an even modular redundancy \( n \)

\[
\gamma = \frac{n}{2},
\]

139
or exactly half of \( n \). This is because the system itself has not failed when half of the modules have failed, because the failure can still be detected.

The Markov model for a general NMR system is presented in Figure C.6. As in TMR, \( S_0 \) represents the normal operation state where all of the redundant copies are operating correctly. In states \( S_1 \) to \( S_i \), \( i \) of the redundant modules have failed and are operating incorrectly. The maximum number of impaired states is \( n - \gamma \). In state \( S_{n-\gamma+1} \), at least half of the redundant modules have failed and the system itself has failed. The failure rate between any of the states is the number of redundant modules, \( n \), minus the number of already failed modules, \( i \), times the failure rate of each module, which is \( (n-i)\lambda \).

The transition matrix for NMR is variable, depending on the amount of modular redundancy \( n \),

\[
T = \begin{bmatrix}
-n\lambda & n\lambda & 0 & \ldots & 0 & 0 \\
0 & -(n-1)\lambda & (n-1)\lambda & \ldots & 0 & 0 \\
0 & 0 & 0 & \ldots & -\gamma\lambda & \gamma\lambda \\
\end{bmatrix}
\]

A general equation for reliability and MTTF for any NMR system cannot be directly derived from the Markov model, but they can be for specific values of \( n \). This is done by choosing \( n \) and then computing the transition matrix for the system.
For example, \( n=5 \) has the following transition matrix:

\[
T = \begin{bmatrix}
-5\lambda & 5\lambda & 0 & 0 \\
0 & -4\lambda & 4\lambda & 0 \\
0 & 0 & -3\lambda & 3\lambda \\
0 & 0 & 0 & 1
\end{bmatrix}.
\]

After three module failures, the 5MR system has failed. While the TMR system can only tolerate one module failure, the 5MR system can tolerate up to, and including, two module failures. The reliability and MTTF equations for the 5MR system would then be,

\[
r(t) = 10e^{-3\lambda t} - 15e^{-4\lambda t} + 6e^{-5\lambda t},
\]

\[
\text{MTTF} = \frac{47}{60} \times \frac{1}{\lambda}.
\]

As with TMR, NMR does not actually increase the MTTF over the Simplex system, the MTTF is always worse for any \( n \). This happens because as more redundancy is added, the rate for a failure in any module increases. As \( n \to \infty \) then \( n\lambda \to \infty \). Using the same method as was shown for 5MR, the MTTF for \( N=7, N=9 \) and \( N=11 \) can also be calculated,

\[
\text{MTTF}_{7\text{MR}} = \frac{319}{420} \times \frac{1}{\lambda},
\]

\[
\text{MTTF}_{9\text{MR}} = \frac{1879}{2520} \times \frac{1}{\lambda},
\]

\[
\text{MTTF}_{11\text{MR}} = \frac{20417}{27720} \times \frac{1}{\lambda}.
\]

Comparing the values of these NMR systems produces the following inequality:

\[
\text{MTTF}_{\text{SIM}} > \text{MTTF}_{\text{TMR}} > \text{MTTF}_{5\text{MR}} > \text{MTTF}_{7\text{MR}} > \text{MTTF}_{9\text{MR}} > \text{MTTF}_{11\text{MR}} > \ldots
\]

This trend shows that the MTTF decreases as more redundancy is added.
However, MTTF is not necessarily the best metric to judge the quality of TMR or NMR. The reliability equations for the NMR systems show their benefit:

\[ r(t)_{7\text{MR}} = 35e^{-4\lambda t} - 84e^{-5\lambda t} + 70e^{-6\lambda t} - 20e^{-7\lambda t}, \]

\[ r(t)_{9\text{MR}} = 126e^{-5\lambda t} - 420e^{-6\lambda t} + 540e^{-7\lambda t} - 315e^{-8\lambda t} + 70e^{-9\lambda t}, \]

\[ r(t)_{11\text{MR}} = 462e^{-6\lambda t} - 1980e^{-7\lambda t} + 3465e^{-8\lambda t} - 3080e^{-9\lambda t} + 1386e^{-10\lambda t} - 252e^{-11\lambda t}. \]

By plotting \( r(t) \) for these NMR systems, Figure C.7 shows that the initial reliability of redundancy is better at small \( t \). It is only as the system runs for a long time, i.e., at large \( t \), that the redundancy becomes worse, leading to an overall lower MTTF. If the system only needs a high reliability during its initial operation, then these NMR techniques become very beneficial.
C.2.3 TMR with Repair Model

TMR by itself only provides masking effects for the system. When a failure occurs in one of the modules, the TMR system will successfully mask that failure, provided the other two modules are still operating correctly. However, that module will continue failing as there is no inherent mechanism in the system to repair it. Even though the TMR system is still functionally correct, the individual module has still failed. A repair mechanism must be explicitly added to the system to repair the module.

Similar to the failure rate, $\lambda$, the repair rate, $\mu$, is represented as the number of module repairs per unit time. The Markov model for this is similar to TMR, with an additional arc from the impaired operation state ($S_1$) to the normal operation state ($S_0$) as shown in Figure C.8. A repair arc is not included from the failed operation state to any other state in this model because it is intended to measure reliability and MTTF.

The state transition matrix can be updated to include the repair arc,

$$T = \begin{bmatrix}
-3\lambda & 3\lambda & 0 \\
\mu & -(2\lambda + \mu) & 2\lambda \\
0 & 0 & 0
\end{bmatrix},$$

which can be used to generate the reliability and MTTF equations:

$$r(t) = e^{-\frac{1}{2}(5\lambda + \mu)t} \left[ \cosh(\frac{1}{2}t\sqrt{\lambda^2 + 10\lambda\mu + \mu^2}) + \frac{\sinh(\frac{1}{2}t\sqrt{\lambda^2 + 10\lambda\mu + \mu^2})(5\lambda + \mu)}{\sqrt{\lambda^2 + 10\lambda\mu + \mu^2}} \right],$$

$$\text{MTTF} = \frac{5}{6\lambda} + \frac{\mu}{6\lambda^2}. $$

---

3Repair can also be added to NMR and will provide similar improvements as presented for TMR.
TMR with repair can provide both higher reliability and higher MTTF values when the repair to failure rate, $\mu / \lambda$, is sufficiently high. TMR with repair only fails if two failures occur in two different modules before the system has been repaired. Figure C.9 shows plots for TMR with repair systems at different scrub speeds. As the scrub speed increases over the module failure rate, the reliability of the system can significantly improve. In fact, as $\mu / \lambda \to \infty$, the MTTF goes to infinity and $r(t)$ stays at 1 for all $t$. What this means is that the system is able to immediately correct any failures that occur in any module before another failure can occur in a different module.

C.2.4 TMR Partitioning

Partitioning can also be used to increase the system’s robustness to failures. In partitioning, the modules are divided into sub-modules with voters placed on the outputs of each group of sub-modules. This allows the system to tolerate multiple failures in multiple modules, as long as
those failures occur in different sub-modules. Generating the Markov model for the TMR with partitioning system is highly dependent on the system implementation. For the sake of simplicity, it is assumed that the system is evenly divided between the $k$ partitions. This is a valid assumption as long as for any two partitions the failure rate is the same, i.e., $\lambda_{\text{PAR}_i} \approx \lambda_{\text{PAR}_j}$. Making this simplifying assumption, the Markov chain for partitioning is shown in Figure C.10.

For this model, $\lambda$ refers to the failure rate of one module, the same as the other TMR systems\footnote{Note that $\lambda_{\text{PAR}} \approx \lambda_{\text{TMR}}$, but is slightly higher due to the presence of $k$ voting groups, $k - 1$ more voting groups than TMR (which has one voting group).}. For any single sub-module, the failure rate would be $\lambda_{\text{SUB}} \approx \lambda / k$. At state $S_i$, exactly $i$ of the $k$ partitions are impaired, i.e., have one failed sub-module. This means $k - i$ partitions are still operating normally, i.e., no sub-module failures, which implies that the model can transition into $S_{i+i}$ state at the failure rate of a module $3\lambda / k$ times the number of these correctly operating modules, $k - i$. The system can transition into the fail state, $S_{k+2}$, at the failure of two sub-modules, $2\lambda / k$, times that number of impaired partitions, $i$. 

Figure C.10: TMR Partitioning Reliability Model for $k$ Partitions
Similar to NMR, the state transition matrix for TMR with partitioning is variable, depending on the number of partitions $k$,

$$
T = \begin{bmatrix}
-3\lambda & 3\lambda & \ldots & 0 & \ldots & 0 & 0 \\
0 & -\left(\frac{3(k-1)}{k}\lambda + \frac{2}{k}\lambda\right) & \ldots & 0 & \ldots & 0 & \frac{2}{k}\lambda \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \ldots & 0 & \ldots & 0 & \frac{2i}{k}\lambda \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \ldots & 0 & \ldots & -2\lambda & 2\lambda \\
0 & 0 & \ldots & 0 & \ldots & 0 & 0
\end{bmatrix}
$$

The reliability for partitioning with different $k$, plotted against the reliability for the Simplex and TMR systems is shown in Figure C.11. There are two important takeaways from the graph and these derivations. First, partitioning yields higher reliability and MTTF than TMR (without repair). Second, partitioning has a higher MTTF than the Simplex system. This means, that even without a repair element, a redundant system with partitioning will, on average, survive longer than the Simplex system.

While the results for partitioning are interesting, most, if not all deployed, redundant systems would also employ a repair mechanism. Such a system would have the same Markov chain as partitioning with no repair, but with an added arc from each impaired operation state into the correct operation state, as shown in Figure C.12. An inherit assumption of this model is that there is a global repair mechanism for all of the partitioned systems, i.e., all failed partitions can be repaired at the same time. The transition matrix for this system is similar, except with an added $\mu$.
term along the first column of the matrix for each impaired state,

\[
T = \begin{bmatrix}
-3\lambda & 3\lambda & \cdots & 0 & \cdots & 0 & 0 \\
\mu & -\left(\frac{3(k-1)}{k} \lambda + \frac{2}{k} \lambda + \mu \right) & \cdots & 0 & \cdots & 0 & \frac{2k}{k} \lambda \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\
\mu & 0 & \cdots & \frac{3(k-i)}{k} \lambda & \cdots & 0 & \frac{2i}{k} \lambda \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\
\mu & 0 & \cdots & 0 & \cdots & -(2\lambda + \mu) & 2\lambda \\
0 & 0 & \cdots & 0 & \cdots & 0 & 0
\end{bmatrix}
\]

The reliability equations for this system derived for different values of k are presented in Figure C.13. As expected, both repair and increasing the number of partitions increases the system’s reliability.
Figure C.12: TMR with Repair Partitioning Reliability Model for $k$ Partitions
Figure C.13: Comparison of TMR with Repair and Partitioning ($\mu = 100\lambda$)
APPENDIX D. MODELING COMMON CAUSE FAILURES IN TMR

All of the models from the previous appendix carry an inherit assumption, each of the system’s modules fail independently. This assumption exists since in all of the Markov chains there is no connection between the normal operation state ($S_0$) and the failed state (last state). This implies that in order for the system to fail, it must pass through the impaired operation state which requires at least two separate events for the system to fail.

This assumption is not true for all systems. For some systems, it is possible for multiple modules to fail *simultaneously*. In the Markov chain, this translates to a connection between the normal operation state and the failed state. These types of failures are called common cause failures (CCF). Rather, CCF refers to any single event that simultaneously causes multiple TMR modules to fail. CCF is a more general term that is not system specific. CCF would encompass all SBF, which is a specific term for FPGA systems. Some theoretical causes and effects of CCF have been explored before [88–90]. This work contributes more insights into how CCF affects a system.

While these models are motivated by FPGA use cases, just like the models in the previous appendix, these new Markov chains can be applied to any system that they model. The goal of this appendix is to explore the limitations imposed by CCF when compared against the more ideal models from the previous appendix. This will lead to theoretical insights on how mitigation strategies that target CCF will impact the system reliability and the fundamental limitations of various strategies.

D.1 TMR with CCF Model

To understand how CCF effects TMR, the models presented from the last appendix can be altered to include a failure path for CCF. The following subsections present the new models for TMR with CCF and analyzes the impact CCF has on the models presented in the last appendix.
This will start with the least complicated system model, TMR without repair, and work up to showing the effects of CCF on a system with partitioning and repair.

Just as each module has a defined failure rate, \( \lambda \), a failure rate is used for CCF, \( \lambda_{\text{CCF}} \). This rate is different than the module failure rate, but can be expressed with the following ratio,

\[
\lambda_{\text{CCF}} = \rho \lambda, \tag{D.1}
\]

where \( \rho \) represents the scaling factor. Expressing CCF in this manner has two benefits. First, it allows the symbolic equations it is used in, i.e., the reliability and MTTF equations, to be simplified and made relative to the module failure rate. Second, it allows classifications for different ranges of rates for CCF, such as for CCF where \( \rho \geq 1 \) and CCF where \( \rho < 1 \).

Without having to generate any models, it should be apparent that when \( \rho \geq 1 \), any redundant system will necessarily have worse reliability. This is because there is a path directly to the failure state with at least the same failure rate as the Simplex system. It does not matter whether the system is TMR with repair, partitioning, etc., the best MTTF that system could achieve is \( 1/ (\rho \lambda) \), and that is only when assuming that the remaining part of the system is perfect. If \( \rho \geq 1 \) then it is better to use the Simplex system than to use redundancy. To prove this point, the reliability graphs for each system will include a plot with \( \rho = 1 \).

### D.1.1 TMR

Beginning with the TMR system without repair, CCF can be modeled by adding two arcs. The first arc is added from the normal operation state \( S_0 \) to the fail state \( S_2 \), with the CCF failure

![Figure D.1: TMR with CCF Reliability Model](image-url)
of \( \lambda_{CCF} \). Another arc must be added from the impaired operation state (\( S_1 \)) to the failed state as well. This additional arc is needed because the CCF event can still occur even when another module has already failed. The new TMR model is shown in Figure D.1. For the visual Markov chains, \( \lambda_{CCF} \) will be used, while the substitution shown in Equation D.1 will be used for all the equations and transition matrices. Using the Markov chain to derive the transition matrix,

\[
T = \begin{bmatrix}
-2\rho \lambda & 3\lambda & \rho \lambda \\
0 & -2\rho \lambda & \lambda(2\rho) \\
0 & 0 & 0
\end{bmatrix}.
\]

Due to the complex nature of the reliability equations, they have been plotted in Figure D.2 instead of expressed symbolically. The first plot of interest is \( \rho = 1 \). As was deduced earlier, this system is strictly worse than the Simplex system. The next plot of interest is \( \rho = .1 \). This is the only one
of the TMR plots that has any discernible difference from the ideal TMR system, i.e., $\rho = 0$. As $\rho \to 0$, the limitations of the TMR system itself begin to dominate and limit the reliability. The limitation of the system as $\rho \to 0$ is not due to CCF. At higher values of for CCF, $1 > \rho > .1$, CCF does impose some limitations on the system.

The MTTF equations can be used to better compare the higher values of $\rho$ since they are visually identical from the plot. The general MTTF equation for any value of $\rho$ is

$$MTTF = \frac{5 + \rho}{\lambda (\rho^2 + 5\rho + 6)}.$$  

As $\rho \to 0$, or $\lambda_{CCF} \to 0$, the MTTF of TMR with CCF becomes the same as the MTTF of TMR. Analyzing the MTTF for the three different $\rho$ from the figure yields

$$MTTF_{\rho=0.1} = \frac{170}{217\lambda} \approx \frac{0.78}{\lambda},$$

$$MTTF_{\rho=0.01} = \frac{16700}{20167\lambda} \approx \frac{0.83}{\lambda},$$

$$MTTF_{\rho=0.001} = \frac{1667000}{2001667\lambda} \approx \frac{0.83}{\lambda},$$

$$MTTF_{\rho=0.0} = \frac{5}{6\lambda} \approx \frac{0.83}{\lambda}.$$  

Again, as these MTTF calculations show only relatively high values of $\rho$, i.e., $\rho \geq 0.1$, causes discernible decreases in MTTF.

**D.1.2 TMR with Repair**

CCF can be added to the TMR with repair system just as it was added to the TMR system, that is adding two arcs, one from the correct operating state to the failed state and the other from the impaired operation state to the failed state. Each of these arcs transitions at the CCF failure rate, $\lambda_{CCF}$. These changes are shown in Figure D.3. Deriving the transition matrix from the model leads to

$$T = \begin{bmatrix}
-(3 + \rho)\lambda & 3\lambda & \rho\lambda \\
\mu & -(2 + \rho)\lambda + \mu & 2\lambda + \rho\lambda \\
0 & 0 & 0
\end{bmatrix}.$$  

153
The extra variable $\mu$ makes TMR with repair and CCF more difficult to analyze than the without repair TMR system as both the $\mu$ and $\rho$ variables need to be swept to draw conclusions. Figure D.4 shows charts sweeping $\rho$ ($\rho = 0.1, 0.01, 0.001$) and each chart is a different $\mu$ ($\mu = 100, 1000, 10000\lambda$).

Returning to the $\rho = 1$ argument addressed earlier, all of these plots clearly show no benefit to redundancy when $\rho = 1$. Even as these systems become more perfect, i.e., the repair to failure rate goes to infinity, $\mu/\lambda \to \infty$, the presence of CCF makes this system no better than the Simplex system. However, as $\rho \to 0$, the graphs do show the benefits of redundancy.

As the CCF rate becomes lower than the module failure rate, the reliability over time of the TMR system increases from the Simplex system. The amount of benefit varies according to $\rho$. Just as in the TMR without repair system, as $\rho \to 0$ the system approaches the reliability of the ideal TMR system with repair and no CCF. Unlike the TMR without repair system, there is a visually discernible difference between the different values of $\rho$.

As the repair rate $\mu$ becomes larger, even low values of $\rho$ can have a significant impact on the system. This is not to say that the system digresses back into the Simplex system, but the limitation on reliability that CCF imposes becomes apparent. This is evident when comparing the different charts in Figure D.4a-c. In (a) where $\mu = 100\lambda$, only the system where $\rho = .1$ is significantly different than the ideal system. Then in (b) where $\mu = 1000\lambda$, both the system where $\rho = .1$ and the system where $\rho = .01$ are significantly different from the ideal system. Finally, in (c) where $\mu = 10000\lambda$, all three of the TMR with CCF systems are significantly different from the ideal system. This result suggests that there is a maximum effective repair rate for each value of $\rho$. 

Figure D.3: TMR with Repair and CCF Reliability Model
Figure D.4: Comparison of TMR with Repair and CCF

The MTTF can be used to explore the maximum effective repair rate. From the transition matrix, the MTTF for TMR with repair and CCF is

\[ MTTF = \frac{(5 + \rho) + \mu / \lambda}{6\lambda + 5\rho \lambda + \lambda \rho^2 + \rho \mu / \lambda}. \]

It is possible to calculate the best possible MTTF by allowing an infinite repair rate, i.e., \( \mu \to \infty \). The best MTTF possible for this system would be

\[ \lim_{\mu \to \infty} MTTF = \frac{1}{\rho \lambda} = \frac{1}{\lambda_{CCF}}. \]
When CCF is not present, the MTTF would be infinite, so CCF clearly places a limitation. Since an infinite repair rate does not mean an infinite MTTF, this begs the question of how high the repair rate can increase before it no longer yields benefits in the MTTF. Figure D.5 plots the MTTF vs. the repair rate, relative to the failure rate, for various values of $\rho$. The graph shows that as the repair rate increases, so does the MTTF when there is no CCF.

The next conclusion from the graph is that with every order of magnitude reduction in CCF corresponds to a potential order of magnitude increase in the MTTF, if the repair rate is sufficiently high, which is generally the case in FPGA-based systems. But, the repair rate can also be the limiting factor. If the CCF rate is sufficiently low, than the MTTF of the system can only be improved by scaling the repair rate. For example, for $\rho = .001$, the maximum effective repair rate is about $10^4$. After reducing CCF by an order of magnitude, $\rho = .0001$, the new maximum effective repair rate would be about $10^5$.  

Figure D.5: MTTF Comparison of Different Repair Rates and CCF Rates
There are two ways this analysis can be used. First, if the repair rate is not constrained, then the designer can measure the CCF rate and use that to choose the maximum effective repair rate. Conversely, if the repair rate is constrained, but the CCF rate can be altered, the repair rate can be used to decide an allowable amount of CCF. For example, if the repair rate is constrained to $10^2$, a CCF rate of $\rho = .001$ would be permittable as it would not be the bottleneck for system reliability.

### D.1.3 Partitioning

Having analyzed the effects of CCF on a TMR with repair system, a similar analysis can be made on the most sophisticated of the system models, TMR with partitioning and repair. Due to the high number of unconstrained variables ($\mu$, $\rho$ and $k$), $k$ will be constrained to 4 for this analysis.

Figure D.6 shows the Markov chain for TMR with partitioning, repair and CCF. Similar to other models with CCF, this model allows for system failure from the correct operation state, $S_0$. Then to complete the model, $\lambda_{CCF}$ is added on the arcs from states $S_1$ through $S_k$ to the failure state $S_{k+1}$.

There are some inherit assumptions made from this model with regards to CCF. First, this model assumes the same initial CCF rate as the other models. This is might not be true for all systems. In all likelihood, some CCF events probably affect sub-modules in different partitions. While this would simultaneously impair several partitions, it would not cause system failure, which would leave a chance for repair. However, just as this implies that the failure CCF rate should initially be lower, it would also imply that the CCF rate should grow as partitions become impaired. This would occur because as more partitions become impaired, there is a greater chance that one of the partitions simultaneously affected by a single event is already impaired. Modeling these events would be highly system dependent, so the simplifying assumption that the CCF rate is constant as sub-module fails is made. This should still provide useful insights about how CCF impacts the system reliability.
Using the Markov chain to derive the transition matrix yields:

$$
T = \begin{bmatrix}
-(3 + \rho)\lambda & 3\lambda & \ldots & 0 & \ldots & 0 & \rho\lambda \\
\mu & -(\frac{3(k-1)}{k}\lambda + \frac{2}{k}\lambda + \lambda_{\text{CCF}} + \mu) & \ldots & 0 & \ldots & 0 & \left(\frac{2}{k}\lambda + \rho\lambda\right) \\
\vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \vdots \\
\mu & 0 & \ldots & \frac{3(k-i)}{k}\lambda & \ldots & 0 & \left(\frac{2i}{k}\lambda + \rho\lambda\right) \\
\vdots & \vdots & \cdots & \vdots & \ddots & \vdots & \vdots \\
\mu & 0 & \ldots & 0 & \ldots & -(2 + \rho\lambda + \mu) & (2 + \rho\lambda) \\
0 & 0 & \ldots & 0 & \ldots & 0 & 0
\end{bmatrix}
$$

Figure D.6: TMR Partitioning with CCF Reliability Model for $k$ Partitions
Figure D.7 shows the reliability of TMR with partitioning with \( k=4, \mu=(100, 1000, 10000)\lambda \) and \( \rho = (0.1, 0.01, 0.001) \). One observation from these graphs is that adding partitioning does provide some reliability benefit when there is CCF. At the lowest \( \mu = 100\lambda \), there is a clear benefit to partitioning, with the lowest \( \rho = 0.001 \) having nearly the higher reliability than the TMR system without CCF and no partitioning.

As \( \mu \) increases, the benefits of partitioning decrease. In Figure D.7b where \( \mu = 1000\lambda \), when \( \rho = 0.1 \), TMR and partitioning are almost identical, with partitioning being slightly higher. At the lower values of CCF, \( \rho = (0.01, 0.001) \), the benefit for partitioning is more clear. However, in Figure D.7c, when \( \rho = 0.01 \) the difference between TMR and partitioning is nearly indistinguishable. The takeaway is the same as a TMR system. To realize the benefits of partitioning over TMR with repair and CCF, as the repair rate increases, there needs to be an equal decrease in the CCF rate.
Figure D.7: Comparison of TMR with Partitioning and Repair and CCF
APPENDIX E. ADDITIONAL CMF REMOVAL ALGORITHMS

Throughout the body of the dissertation, several algorithms were discussed, some supplementary to the actual mitigation techniques and algorithms for the mitigation techniques themselves. One of those algorithms was the fail set algorithm used to determine whether or not a set of cells formed a fail set. The algorithm as explained in the body of the dissertation was sufficient for explanation, but a better heuristic exists that will be explained in this Appendix. Also, another mitigation technique, called RCMF, will be explained along with the results of a fault injection campaign employing the RCMF technique.

E.1 Fail Set Algorithm for Partitioned Circuits

As introduced in Chapter 5, there are multiple ways to determine a fail set. The way presented in the body of the text is solely based on the domains of each of the cells. This is an effective strategy to quickly determine the likelihood of any set of cells actually forming a fail set. While quick, this method may classify many false fail sets, i.e., classifying sets as fail set that are not actual fail sets (false positives).

Many advanced TMR designs are highly partitioned. As discussed in Chapter 3, partitioned designs can tolerate failures in multiple modules, as long as those failures are isolated in different partitions. The number of sub-module failures that a design can tolerate is equal to the number of partitions in a design. When determining fail sets, this implies that cells for multiple domains do not form a fail set if they belong to different partitions.

A better algorithm would take the domain and partition information of each cell into account. There are several steps to execute this new algorithm. The first step is to determine which partition a cell is in. However, unlike the simplified circuits shown throughout this work, partitions do not make clean cuts along the circuit. Cells can actually pertain to many different partitions. In this context, partitioning is not the right term to use as it does not make sense to support a notion
that a cell can belong to more than one partition. What really matters is that a voting group, i.e.,
the triplicated set of voters on the output of a partition, does not fail. Any single domain failure
from a group of cells feeding the voting group will be masked by the voting group. Thus, the first
step is to determine the dependent voters for each cell, or the voters that the cell directly feeds.

Determining the dependent voters for each cell in a circuit can be done by reverse traversing
the circuit’s graphical netlist using a depth first search. This can be done with the buildCellDepen-
dencies function shown in Algorithm 4. The main procedure of this algorithm initializes the
variables for the recursive call to the depth first search algorithm and then calls the algorithm. The
initialization consists of iterating over all of the output cells in the circuit and starting a search along
each input net. The reverse depth first search procedure takes two inputs, the most recent voterID
and the current net. The algorithm then grabs the source cell of the net, updates the voterID if
applicable, and then recursively calls itself on all source nets for the curCell.

Not shown in this algorithm is how to determine the voterID for each voting group. This is
dependent based on how TMR was implemented or which tool was used. For this work, each voter
in the voting group has the same name except for the TMR domain specifier, i.e., one, two or three.
In this case, voting groups can be differentiated based on their names (since the BL-TMR tool uses
similar names for a group of voters). This means that the voterID can be calculated dynamically as
new voting groups are found during the depth first search. An object is maintained which contains
all of the known voting groups and their associated IDs. When a new voting group is found, the
group is added to the known groups and assigned an unique ID.

After finding the voter dependencies for each cell a new fail set algorithm can be proposed,
shown in Algorithm 5. It is similar to the previous fail set algorithm (found in Chapter 5.2.1),
except that this algorithm classifies fail sets whether there are multiple domains cells that feed
a single voting group or not. The algorithm starts by iterating over each cell from the input set
cells. A map is used to record which domains would fail for each voting group. If a second
domain is added to any voting group, this would represent two or more domain failures for a single
voting group and a possible TMR failure. The algorithm would immediately return true under this
circumstance. If the iteration completes then there would be at most one single domain failure for
every voting group in the circuit and the algorithm would return false.
Algorithm 4 Build Cell Dependencies

procedure BUILDCELLDEPENDENCIES(\texttt{design})
  Initialize collection <Cell> outputCells
  Initialize collection <Net> sourceNets
  outputCells ← \texttt{design}.getOutputCells()
  for each cell in outputCells do
    sourceNets ← cell.getSourceNets()
    for each net in sourceNets do
      REVERSEDEPTHFIRSTSEARCH(net, cell.voterID)
    end for
  end for
end procedure

procedure REVERSEDEPTHFIRSTSEARCH(net, voterID)
  Initialize Cell curCell
  Initialize collection <Net> sourceNets
  curCell ← net.getSourceCell()
  if \texttt{curCell} is a voter then
    voterID ← curCell.voterID
  else
    curCell.addDependentVoter(voterID)
  end if
  sourceNets ← curCell.getSourceNets()
  for each net in sourceNets do
    REVERSEDEPTHFIRSTSEARCH(net, voterID)
  end for
end procedure

Algorithm 5 Identify Fail Set

procedure ISFAILSET(cells)
  Initialize map <voterID, set <Domain>> possFailSets
  Initialize collection <voterID> dependentVoters
  for each cell in cells do
    dependentVoters ← cell.getDependentVoters()
    for each voter in dependentVoters do
      possFailSets.add(voter, cell.domain)
      if possFailSets.get(voter).size() ≥ 2 then
        return True
      end if
    end for
  end for
  return False
end procedure
Calculating the cell dependency list is an extra step in the PCMF algorithm that takes time, but the algorithm only has to be run once. The time it takes to build the list, the number of voting groups, the number cells and the average number of voting groups dependencies per cell for each circuit is reported in Table E.1. Results are consistent for each TMR variation in a circuit. The MD5 circuit takes about an order of magnitude more time than the other circuits. This is likely due to the higher number of voter group dependencies for each cell, higher number of cells and a higher amount of voting groups.

### E.2 Incremental Routing (RCMF)

The first technique invented to remove CMF was incremental routing (RCMF), which after fault injection proved to not be an effective option. Incremental routing was first explored because initial testing showed that rerouting data nets associated with clock CMF proved to be effective. Before rerouting a bit would cause clock CMF, but after rerouting that same bit was not sensitive. RCMF will be described below followed by an analysis of why it is not as effective as PCMF.
E.2.1 Identifying Routing CMF

As explained in Chapter 5, there are two muxes in every switchbox on the device that can contain multiple clock nets and cause clock CMF. To identify CMF, then, only these muxes need to be considered. The algorithm for identifying CMF consists of the following steps:

- Do the mux inputs contain two clock nets?
- If yes, are these clocks nets in different columns?
- If yes, are there other nets in the corresponding columns?
- If yes, do these clock nets have downstream cells in the same fail set?
- If yes, then this mux could cause CMF.

To visualize what is occurring, consider Figure E.1. In (a) there is CMF because there is a single bit that can cause multiple shorts between multiple clock nets and other data nets. However, the example mux in (b) would not cause CMF because there is no single bit to simultaneously make both of the clocks short to another data net.

E.2.2 Routing CMF Removal

The RCMF process starts with an already routed design from the vendor’s tool. The first step is to identify all locations that could cause CMF and all data nets associated with those CMF
locations using the just described algorithm. Only these data nets need to be ripped up and rerouted during the first iteration. The second step is to perform routing until the design is completely routed without any clock CMF.

Routing is done using a variation of the PathFinder algorithm [83]. The only difference in the proposed PathFinder algorithm is the cost calculation used during detailed routing. The cost function needs to account for the possibility that using a proposed wire would introduce CMF into the design. This is done by using the following updated cost equation:

\[
\text{cost} = (\text{curCost} + A \times \text{wireCost} + (1 - A) \times ((\text{history} \times \text{usage}) + \text{estimateCost} + \text{restrictedCost})).
\]

This equation includes the term “restrictedCost” to account for the possibility of clock CMF. This equation does not prohibit a net from exploring routing options using this wire, but might discourage exploration because of the clock CMF possibility. If at the end of a routing iteration there are nets that would cause CMF, these nets are ripped up and rerouted during the next routing iteration. Routing concludes when there is no clock CMF and there is no congestion on any of the routing wires.

### E.2.3 Fault Injection Results

The described RCMF algorithm was implemented in RapidSmith2 [62]. Fault injection test were conducted for the b13 circuit, shown in Table E.2. As the table shows RCMF slightly worse than trip-IO. Inspection of the bits that caused failure in the RCMF design revealed that clock CMF was still occurring. Instead of shorting with a data net, clock nets could short with unused lines and still cause CMF. This is likely due to the buffer on the output of each mux in the device which drives unused wires to VCC. While initially promising, RCMF was not explored further because of these results and the success of the PCMF and striping techniques.
Table E.2: RCMF Fault Injection Results for b13

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Bit Sensitivity</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>45,542</td>
<td>617</td>
<td>$1.35 \times 10^{-2}$</td>
<td>1×</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>2,000,000</td>
<td>26</td>
<td>$1.30 \times 10^{-5}$</td>
<td>1,042×</td>
</tr>
<tr>
<td>RCMF</td>
<td>2,400,791</td>
<td>39</td>
<td>$1.62 \times 10^{-5}$</td>
<td>834×</td>
</tr>
<tr>
<td>PCMF</td>
<td>2,000,000</td>
<td>6</td>
<td>$3 \times 10^{-6}$</td>
<td>4,516×</td>
</tr>
</tbody>
</table>
APPENDIX F. IMPLEMENTATION AND FAULT INJECTION RESULTS FOR EACH CIRCUIT

As explained in Chapters 2.3.4 and 6.1.1, four different benchmark circuits were used throughout this work: the b13, md5, sha3, and aes128 circuits. These circuits have been replicated and their outputs have been stitched together so that the circuit utilizes more of the device resources. An explanation for why these circuits were chosen can be found in Chapter 6.1.1. This appendix contains the specific implementation results (Tables F.2-F.5) for each circuit and mitigation technique as well as the specific fault injection results (Tables F.6-F.9) for each circuit and mitigation technique.

F.1 Implementation Metrics

Five metrics were chosen to study the implementation impacts of each of the techniques. The first metric is the maximum achievable frequency for the circuit. This metric helps measure how each technique impacts the ability to meet timing. The second metric is the number of routing nodes. This metric directly relates to power consumption and routing congestion, which impact routeability. The third, fourth and fifth metrics are all area measurements. They measure the number of cells, sites and tiles used by the design.

The vendor’s tool is designed to achieve the given user constraints for timing, and not necessarily to optimize for the maximum timing. With too loose a timing constraint, the tool will simply find a good implementation that meets the timing constraint. With too tight of a timing constraint, the tool will give up and just try and produce a valid implementation. This complicates trying to find the maximum achievable frequency as the chosen constraint may not force the tool to search for a better implementation. To remedy this, a script was developed to find the maximum frequency. The steps used were:

1. Choose a timing constraint;
Table F.1: Mean Implementation Metrics

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>1×</td>
<td>1×</td>
<td>1×</td>
<td>1×</td>
<td>1×</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>0.77×</td>
<td>3.48×</td>
<td>3.16×</td>
<td>3.75×</td>
<td>3.62×</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>0.73×</td>
<td>3.66×</td>
<td>3.44×</td>
<td>3.73×</td>
<td>3.56×</td>
</tr>
<tr>
<td>Split-IO</td>
<td>0.80×</td>
<td>3.69×</td>
<td>3.44×</td>
<td>3.67×</td>
<td>3.55×</td>
</tr>
<tr>
<td>Split-clock</td>
<td>0.73×</td>
<td>3.70×</td>
<td>3.44×</td>
<td>3.83×</td>
<td>3.68×</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>0.73×</td>
<td>3.76×</td>
<td>3.44×</td>
<td>3.83×</td>
<td>3.69×</td>
</tr>
<tr>
<td>ES</td>
<td>0.73×</td>
<td>3.68×</td>
<td>3.44×</td>
<td>3.67×</td>
<td>3.55×</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>0.73×</td>
<td>3.78×</td>
<td>3.44×</td>
<td>3.76×</td>
<td>3.68×</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>0.79×</td>
<td>3.62×</td>
<td>3.17×</td>
<td>3.86×</td>
<td>3.74×</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>0.77×</td>
<td>3.89×</td>
<td>3.44×</td>
<td>3.87×</td>
<td>3.75×</td>
</tr>
<tr>
<td>PCMF</td>
<td>0.76×</td>
<td>3.95×</td>
<td>3.44×</td>
<td>3.87×</td>
<td>3.77×</td>
</tr>
<tr>
<td>Striping</td>
<td>0.78×</td>
<td>4.20×</td>
<td>3.51×</td>
<td>3.46×</td>
<td>3.24×</td>
</tr>
</tbody>
</table>

2. Route the design;

3. Record the achieved frequency;

4. Tighten the timing constraints if timing was met, or loosen the constraints if timing was not met;

5. Repeat by rerouting the design, until the maximum achieved frequency is not changed for two iterations.

The exact script used can be found in Appendix H. Once the maximum frequency is found, the number of nodes, cells, sites and tiles is reported for the maximum frequency implementation. The geomean change across all circuits can be found in Table F.1 and the circuit specific results can be found in Tables F.2-F.5. A higher change is better for the maximum frequency while a lower change is better for all of the other metrics.

An important observation from this table is that routing was not able to complete for TMR with striping for the MD5 and SHA3 circuits. The geomean for these TMR variations was calculated using only the B13 and AES128 circuits. The mean for the number of routing nodes is 4.2× that of the unmitigated circuit, .25 greater than any other TMR variation. This suggests that striping has a great affect on the circuit’s routing.
### Table F.2: Implementation Metrics for b13

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>75.53</td>
<td>226,748</td>
<td>25,388</td>
<td>4,269</td>
<td>2,405</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>68.98</td>
<td></td>
<td>956,331</td>
<td>87,180</td>
<td>18,966</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>54.16</td>
<td>1,050,721</td>
<td>104,014</td>
<td>17,902</td>
<td>9,107</td>
</tr>
<tr>
<td>Split-IO</td>
<td>57.60</td>
<td>1,053,805</td>
<td>104,044</td>
<td>17,902</td>
<td>9,107</td>
</tr>
<tr>
<td>Split-clock</td>
<td>54.46</td>
<td>1,072,431</td>
<td>104,016</td>
<td>17,954</td>
<td>9,149</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>54.91</td>
<td></td>
<td>1,083,673</td>
<td>104,016</td>
<td>17,954</td>
</tr>
<tr>
<td>ES</td>
<td>56.84</td>
<td>1,074,569</td>
<td>104,046</td>
<td>18,164</td>
<td>9,303</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>57.08</td>
<td>1,085,475</td>
<td>104,046</td>
<td>18,164</td>
<td>9,360</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>66.34</td>
<td>1,076,254</td>
<td>87,170</td>
<td>19,049</td>
<td>9,782</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>58.24</td>
<td>1,259,660</td>
<td>104,258</td>
<td>18,751</td>
<td>9,754</td>
</tr>
<tr>
<td>PCMF</td>
<td>58.05</td>
<td>1,266,892</td>
<td>104,258</td>
<td>18,751</td>
<td>9,835</td>
</tr>
<tr>
<td>Striping</td>
<td>57.46</td>
<td>1,275,462</td>
<td>104,258</td>
<td>18,325</td>
<td>9,619</td>
</tr>
</tbody>
</table>

### Table F.3: Implementation Metrics for md5

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>107.68</td>
<td>701,011</td>
<td>65,561</td>
<td>7,872</td>
<td>4,109</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>89.95</td>
<td>2,518,583</td>
<td>202,419</td>
<td>30,683</td>
<td>15,754</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>90.29</td>
<td>2,674,886</td>
<td>214,479</td>
<td>30,778</td>
<td>15,765</td>
</tr>
<tr>
<td>Split-IO</td>
<td>89.29</td>
<td>2,686,487</td>
<td>214,509</td>
<td>30,779</td>
<td>15,765</td>
</tr>
<tr>
<td>Split-clock</td>
<td>87.52</td>
<td>2,680,478</td>
<td>214,481</td>
<td>31,260</td>
<td>15,968</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>85.53</td>
<td>2,816,906</td>
<td>214,481</td>
<td>31,260</td>
<td>16,141</td>
</tr>
<tr>
<td>ES</td>
<td>60.57</td>
<td>2,695,277</td>
<td>214,511</td>
<td>30,497</td>
<td>15,668</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>59.58</td>
<td>2,795,225</td>
<td>214,511</td>
<td>30,497</td>
<td>15,889</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>75.44</td>
<td>2,549,704</td>
<td>202,663</td>
<td>31,688</td>
<td>16,242</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>75.47</td>
<td>2,723,574</td>
<td>214,723</td>
<td>31,729</td>
<td>16,244</td>
</tr>
<tr>
<td>PCMF</td>
<td>71.00</td>
<td>2,836,215</td>
<td>214,723</td>
<td>31,729</td>
<td>16,468</td>
</tr>
<tr>
<td>Striping</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### F.2 Circuit Specific Fault Injection Results

This section contains the circuit specific fault injection results for each mitigation technique and circuit, found in Tables F.6-F.9. An explanation of the fault injection metrics can be found in Chapter 6.2.
Table F.4: Implementation Metrics for sha3

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>sha3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unmitigated</td>
<td>82.51</td>
<td>399,059</td>
<td>20,690</td>
<td>3,452</td>
<td>1,790</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>54.66</td>
<td>1,291,541</td>
<td>65,138</td>
<td>14,242</td>
<td>7,507</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>52.48</td>
<td>1,341,883</td>
<td>71,886</td>
<td>14,567</td>
<td>7,554</td>
</tr>
<tr>
<td>Split-IO</td>
<td>57.24</td>
<td>1,381,222</td>
<td>71,916</td>
<td>13,961</td>
<td>7,141</td>
</tr>
<tr>
<td>Split-clock</td>
<td>52.38</td>
<td>1,382,390</td>
<td>71,888</td>
<td>15,841</td>
<td>8,376</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>52.45</td>
<td>1,383,097</td>
<td>71,888</td>
<td>15,841</td>
<td>8,384</td>
</tr>
<tr>
<td>ES</td>
<td>57.13</td>
<td>1,331,938</td>
<td>71,918</td>
<td>14,447</td>
<td>7,414</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>57.44</td>
<td>1,411,757</td>
<td>71,918</td>
<td>15,842</td>
<td>8,384</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>53.85</td>
<td>1,317,872</td>
<td>65,382</td>
<td>15,189</td>
<td>8,233</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>55.46</td>
<td>1,412,958</td>
<td>72,130</td>
<td>15,573</td>
<td>8,286</td>
</tr>
<tr>
<td>PCMF</td>
<td>55.83</td>
<td>1,426,590</td>
<td>72,130</td>
<td>15,573</td>
<td>8,301</td>
</tr>
<tr>
<td>Striping</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table F.5: Implementation Metrics for aes128

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>fmax (MHz)</th>
<th># nodes</th>
<th># cells</th>
<th># sites</th>
<th># tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unmitigated</td>
<td>114.39</td>
<td>627,345</td>
<td>56,090</td>
<td>9,531</td>
<td>5,237</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>86.49</td>
<td>1,884,149</td>
<td>167,997</td>
<td>26,602</td>
<td>13,779</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>87.23</td>
<td>1,884,785</td>
<td>168,013</td>
<td>26,602</td>
<td>13,779</td>
</tr>
<tr>
<td>Split-IO</td>
<td>108.27</td>
<td>1,890,578</td>
<td>168,043</td>
<td>25,175</td>
<td>13,625</td>
</tr>
<tr>
<td>Split-clock</td>
<td>87.10</td>
<td>1,880,774</td>
<td>168,015</td>
<td>26,668</td>
<td>13,867</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>87.25</td>
<td>1,881,436</td>
<td>168,015</td>
<td>26,668</td>
<td>13,872</td>
</tr>
<tr>
<td>ES</td>
<td>112.42</td>
<td>1,894,084</td>
<td>168,045</td>
<td>25,165</td>
<td>13,598</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>113.13</td>
<td>1,894,716</td>
<td>168,045</td>
<td>25,165</td>
<td>13,612</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>112.42</td>
<td>1,883,758</td>
<td>168,241</td>
<td>26,716</td>
<td>13,891</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>112.03</td>
<td>1,884,604</td>
<td>168,257</td>
<td>26,716</td>
<td>13,891</td>
</tr>
<tr>
<td>PCMF</td>
<td>112.54</td>
<td>1,885,631</td>
<td>168,257</td>
<td>26,716</td>
<td>13,897</td>
</tr>
<tr>
<td>Striping</td>
<td>90.83</td>
<td>1,971,683</td>
<td>168,257</td>
<td>26,576</td>
<td>13,727</td>
</tr>
<tr>
<td>Metric/Technique</td>
<td>Number of Injections</td>
<td>Number of Failures</td>
<td>Bit Sensitivity</td>
<td>+95% Confidence -95% Confidence</td>
<td>Improvement</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------</td>
<td>--------------------</td>
<td>-----------------</td>
<td>----------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Unmitigated</td>
<td>45,542</td>
<td>617</td>
<td>1.35 × 10⁻²</td>
<td>1.46 × 10⁻², 1.25 × 10⁻²</td>
<td>1×</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>2,000,000</td>
<td>12,027</td>
<td>6.01 × 10⁻³</td>
<td>6.12 × 10⁻³, 5.90 × 10⁻³</td>
<td>2.3×</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>2,000,000</td>
<td>2,791</td>
<td>1.40 × 10⁻³</td>
<td>1.45 × 10⁻³, 1.34 × 10⁻³</td>
<td>9.7×</td>
</tr>
<tr>
<td>Split-IO</td>
<td>2,000,000</td>
<td>39</td>
<td>1.95 × 10⁻⁵</td>
<td>2.67 × 10⁻⁵, 1.39 × 10⁻⁵</td>
<td>695×</td>
</tr>
<tr>
<td>Split-clock</td>
<td>373,836</td>
<td>307</td>
<td>8.21 × 10⁻⁴</td>
<td>9.15 × 10⁻⁴, 7.27 × 10⁻⁴</td>
<td>16.5×</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>336,939</td>
<td>200</td>
<td>5.94 × 10⁻⁴</td>
<td>6.78 × 10⁻⁴, 5.10 × 10⁻⁴</td>
<td>22.8×</td>
</tr>
<tr>
<td>ES</td>
<td>3,255,262</td>
<td>70</td>
<td>2.15 × 10⁻⁵</td>
<td>2.66 × 10⁻⁵, 1.64 × 10⁻⁵</td>
<td>630×</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>2,000,000</td>
<td>31</td>
<td>1.55 × 10⁻⁵</td>
<td>2.20 × 10⁻⁵, 1.05 × 10⁻⁵</td>
<td>874×</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>2,000,000</td>
<td>14,872</td>
<td>7.44 × 10⁻³</td>
<td>7.56 × 10⁻³, 7.31 × 10⁻³</td>
<td>1.8×</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>2,000,000</td>
<td>26</td>
<td>1.30 × 10⁻⁵</td>
<td>1.90 × 10⁻⁵, 8.95 × 10⁻⁶</td>
<td>1,042×</td>
</tr>
<tr>
<td>PCMF</td>
<td>2,000,000</td>
<td>6</td>
<td>3.00 × 10⁻⁶</td>
<td>6.55 × 10⁻⁶, 1.10 × 10⁻⁶</td>
<td>4,516×</td>
</tr>
<tr>
<td>Striping</td>
<td>2,000,000</td>
<td>0</td>
<td>5.00 × 10⁻⁷</td>
<td>1.85 × 10⁻⁶, 0.00 × 10⁰</td>
<td>27,096×</td>
</tr>
</tbody>
</table>
Table F.7: Fault Injection Results for md5

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Bit Sensitivity</th>
<th>+95% Confidence</th>
<th>-95% Confidence</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>2,000,000</td>
<td>129,677</td>
<td>6.48 × 10⁻²</td>
<td>6.52 × 10⁻²</td>
<td>6.45 × 10⁻²</td>
<td>1×</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>37,318</td>
<td>326</td>
<td>8.74 × 10⁻³</td>
<td>9.70 × 10⁻³</td>
<td>7.77 × 10⁻³</td>
<td>7.4×</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>132,724</td>
<td>375</td>
<td>2.83 × 10⁻³</td>
<td>3.12 × 10⁻³</td>
<td>2.53 × 10⁻³</td>
<td>23×</td>
</tr>
<tr>
<td>Split-IO</td>
<td>2,000,000</td>
<td>5,275</td>
<td>2.64 × 10⁻³</td>
<td>2.71 × 10⁻³</td>
<td>2.56 × 10⁻³</td>
<td>24.6×</td>
</tr>
<tr>
<td>Split-clock</td>
<td>2,000,000</td>
<td>325</td>
<td>1.63 × 10⁻⁴</td>
<td>1.81 × 10⁻⁴</td>
<td>1.44 × 10⁻⁴</td>
<td>399×</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>2,000,000</td>
<td>151</td>
<td>7.55 × 10⁻⁵</td>
<td>8.78 × 10⁻⁵</td>
<td>6.32 × 10⁻⁵</td>
<td>859×</td>
</tr>
<tr>
<td>ES</td>
<td>2,000,000</td>
<td>198</td>
<td>9.90 × 10⁻⁵</td>
<td>1.13 × 10⁻⁴</td>
<td>8.49 × 10⁻⁵</td>
<td>655×</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>12,000,000</td>
<td>291</td>
<td>2.43 × 10⁻⁵</td>
<td>2.71 × 10⁻⁵</td>
<td>2.14 × 10⁻⁵</td>
<td>2,674×</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>2,000,000</td>
<td>12,356</td>
<td>6.18 × 10⁻³</td>
<td>6.29 × 10⁻³</td>
<td>6.07 × 10⁻³</td>
<td>10.5×</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>2,000,000</td>
<td>195</td>
<td>9.75 × 10⁻⁵</td>
<td>1.11 × 10⁻⁴</td>
<td>8.35 × 10⁻⁵</td>
<td>665×</td>
</tr>
<tr>
<td>PCMF</td>
<td>2,000,000</td>
<td>43</td>
<td>2.15 × 10⁻⁵</td>
<td>2.90 × 10⁻⁵</td>
<td>1.56 × 10⁻⁵</td>
<td>3,016×</td>
</tr>
<tr>
<td>Striping</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Table F.8: Fault Injection Results for sha3

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Bit Sensitivity</th>
<th>+95% Confidence</th>
<th>-95% Confidence</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>2,000,000</td>
<td>78,376</td>
<td>3.92 × 10⁻²</td>
<td>3.95 × 10⁻²</td>
<td>3.89 × 10⁻²</td>
<td>1×</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>2,000,000</td>
<td>4,513</td>
<td>2.26 × 10⁻³</td>
<td>2.32 × 10⁻³</td>
<td>2.19 × 10⁻³</td>
<td>17.3×</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>2,000,000</td>
<td>96</td>
<td>4.80 × 10⁻⁵</td>
<td>5.78 × 10⁻⁵</td>
<td>3.82 × 10⁻⁵</td>
<td>816×</td>
</tr>
<tr>
<td>Split-IO</td>
<td>2,000,000</td>
<td>45</td>
<td>2.25 × 10⁻⁵</td>
<td>3.01 × 10⁻⁵</td>
<td>1.64 × 10⁻⁵</td>
<td>1,742×</td>
</tr>
<tr>
<td>Split-clock</td>
<td>2,000,000</td>
<td>68</td>
<td>3.40 × 10⁻⁵</td>
<td>4.22 × 10⁻⁵</td>
<td>2.58 × 10⁻⁵</td>
<td>1,153×</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>2,000,000</td>
<td>64</td>
<td>3.20 × 10⁻⁵</td>
<td>4.00 × 10⁻⁵</td>
<td>2.40 × 10⁻⁵</td>
<td>1,225×</td>
</tr>
<tr>
<td>ES</td>
<td>2,000,000</td>
<td>25</td>
<td>1.25 × 10⁻⁵</td>
<td>1.84 × 10⁻⁵</td>
<td>8.10 × 10⁻⁶</td>
<td>3,135×</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>2,000,000</td>
<td>31</td>
<td>1.55 × 10⁻⁵</td>
<td>2.20 × 10⁻⁵</td>
<td>1.05 × 10⁻⁵</td>
<td>2,528×</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>10,000,000</td>
<td>23,608</td>
<td>2.36 × 10⁻³</td>
<td>2.39 × 10⁻³</td>
<td>2.33 × 10⁻³</td>
<td>16.6×</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>12,000,000</td>
<td>2</td>
<td>1.67 × 10⁻⁷</td>
<td>6.00 × 10⁻⁷</td>
<td>1.67 × 10⁻⁸</td>
<td>235,128×</td>
</tr>
<tr>
<td>PCMF</td>
<td>12,000,000</td>
<td>0</td>
<td>8.33 × 10⁻⁸</td>
<td>3.08 × 10⁻⁷</td>
<td>0.00 × 10⁰</td>
<td>470,256×</td>
</tr>
<tr>
<td>Striping</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Table F.9: Fault Injection Results for aes128

<table>
<thead>
<tr>
<th>Metric/Technique</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Bit Sensitivity</th>
<th>+95% Confidence</th>
<th>-95% Confidence</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>2,000,000</td>
<td>121,780</td>
<td>6.09 × 10^{-2}</td>
<td>6.12 × 10^2</td>
<td>6.05 × 10^2</td>
<td>1×</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>2,000,000</td>
<td>101</td>
<td>5.05 × 10^{-5}</td>
<td>6.05 × 10^{-5}</td>
<td>4.05 × 10^{-5}</td>
<td>1,206×</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>2,000,000</td>
<td>105</td>
<td>5.25 × 10^{-5}</td>
<td>6.27 × 10^{-5}</td>
<td>4.23 × 10^{-5}</td>
<td>1,160×</td>
</tr>
<tr>
<td>Split-IO</td>
<td>2,000,000</td>
<td>88</td>
<td>4.40 × 10^{-5}</td>
<td>5.34 × 10^{-5}</td>
<td>3.46 × 10^{-5}</td>
<td>1,384×</td>
</tr>
<tr>
<td>Split-clock</td>
<td>2,164,484</td>
<td>78</td>
<td>3.60 × 10^{-5}</td>
<td>4.42 × 10^{-5}</td>
<td>2.79 × 10^{-5}</td>
<td>1,690×</td>
</tr>
<tr>
<td>Split-clock-PCMF</td>
<td>2,000,000</td>
<td>71</td>
<td>3.55 × 10^{-5}</td>
<td>4.39 × 10^{-5}</td>
<td>2.71 × 10^{-5}</td>
<td>1,715×</td>
</tr>
<tr>
<td>ES</td>
<td>2,000,000</td>
<td>20</td>
<td>1.00 × 10^{-5}</td>
<td>1.54 × 10^{-5}</td>
<td>6.10 × 10^{-6}</td>
<td>6,089×</td>
</tr>
<tr>
<td>ES-PCMF</td>
<td>2,000,000</td>
<td>24</td>
<td>1.20 × 10^{-5}</td>
<td>1.78 × 10^{-5}</td>
<td>7.70 × 10^{-6}</td>
<td>5,074×</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>10,000,000</td>
<td>4</td>
<td>4.00 × 10^{-7}</td>
<td>1.02 × 10^{-6}</td>
<td>1.00 × 10^{-7}</td>
<td>152,225×</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>12,000,000</td>
<td>0</td>
<td>8.33 × 10^{-8}</td>
<td>3.08 × 10^{-7}</td>
<td>0.00 × 10^{0}</td>
<td>730,680×</td>
</tr>
<tr>
<td>PCMF</td>
<td>5,541,693</td>
<td>0</td>
<td>1.80 × 10^{-7}</td>
<td>6.68 × 10^{-7}</td>
<td>0.00 × 10^{0}</td>
<td>337,434×</td>
</tr>
<tr>
<td>Striping</td>
<td>2,000,000</td>
<td>0</td>
<td>5.00 × 10^{-7}</td>
<td>1.85 × 10^{-6}</td>
<td>0.00 × 10^{0}</td>
<td>121,780×</td>
</tr>
</tbody>
</table>
APPENDIX G. GLOSSARY & ACRONYMS

This appendix contains the glossary, list of acronyms and list of the TMR circuits (including mitigation techniques) used in this work. In addition, after the glossary is an explanation of the differences between similar terms.

G.1 Glossary

Absorbing state - Any state in a Markov chain where there are no paths to any other state.

Antenna, routing - A routing antenna occurs when an SEU connects a wire to an existing net in the circuit. This differs from a short as the newly connected wire was not driven by another net in the circuit.

Bitgen - The final step in the implementation flow for an FPGA design. In bitgen, a bitstream is generated for a completely placed and routed design for the target device.

BL-TMR tool - An automatic, netlist TMR tool created by BYU and LANL. The BL-TMR tool takes an EDIF file as input and outputs a new EDIF file with TMR applied. The BL-TMR tool was configured to apply feedback TMR for this work.

Carry chain dependency - A dependency between two slices that requires both slices to be placed in consecutive tiles in the same column. If the two slices are not placed on consecutive tiles in the same column it creates an invalid design implementation.

Clock CMF - CMF that is caused by the simultaneous shorting of two different TMR domain clock nets due to an SEU in a column bit of a routing mux.

Column bit - Any bit in a routing mux that controls a column wire. Setting a column bit in a routing mux allows the source wires in the routing mux pertaining to the set column to drive the
row wires of the mux. The output wire of the mux is not driven unless a row bit in the routing mux has also been set.

**Common-IO** - A reference to TMR designs on FPGAs where the I/O pins have not been triplicated.

**Common cause failure (CCF)** - A common cause failure is any failure in a TMR system due to a single event. Similar to a single bit failure, common cause failure is a general term applicable to any system, not just FPGAs.

**Common mode failure (CMF)** - Any event that simultaneously causes failures in multiple TMR modules and causes the system to fail. These failures occur in the redundant network of the system, as opposed to single point failures which occur in untriplicated components.

**Component** - Any singular part of a system. A component could refer to an input line, a specific voter or part of a module/sub-module.

**Confidence intervals** - A range with a probability describing the likelihood of the true value of a measurement lying within it. For example, when 95% confidence intervals are presented, they represent the 95% confidence that the true value of that measurement lies between those bounds.

**Continuous time Markov chain** - A model describing the probability of an event based solely on the state of the system from a previous event.

**Cross-section** - A measurement obtained during radiation testing that expresses the likelihood of a radioactive effect occurring. Expressed in units of cm$^2$.

**Decomposition** - Dividing a unreplicated system into smaller sub-systems which are then chained together to form the system. The division points are defined so that during triplication, each sub-system can be triplicated to form partitions.

**Device/design under test (DUT)** - The device or design that is subject to testing. In radiation testing the DUT would be exposed to radiation and in fault injection the DUT would be the device/design that is being injected.
**Domain** - The specific number for a redundant copy in a redundant system. For example, if cell A pertained to the second copy of a TMR system, it would be in a domain 2. Valid domains for a TMR system would be 1, 2 and 3.

**Error** - In this work it is defined the same as a fault in that it represents some module, system, etc. containing some sort of impairment that may or may not cause a failure. For example, in fault injection an error/fault would represent an injected bit which may or may not cause a detectable circuit failure.

**Failure** - A module, sub-module, sub-system, partition or system that is operating incorrectly. A failure does not mean that module, etc. is always incorrect, but may be incorrect under certain input vectors.

**Failure function** $F(t)$ - A reliability metric that is similar to the reliability function $r(t)$. It represents that probability that a system has failed at time $t$.

**Failures in time (FIT)** - A reliability metric that represents the average number of failures in a billion hours of operation. FIT is usually calculated at a sea-level flux.

**Fail set** - A set of cells in a circuit that cause circuit failure when they all simultaneously fail.

**Fault** - See error.

**Fault Injection** - A form of reliability testing where configuration bits are upset within the FPGA to observe whether or not they cause circuit failure. If a single bit is injected during each fault injection iteration, then after testing the bit sensitivity can be calculated by dividing the observed failures by the total number of bits injected. This work uses random fault injection where the configuration bits that are injected are selected randomly.

**Feedback TMR** - A variation of TMR where voters are placed on all of the feedback paths in the circuit.

**Fluence** - A number representing the number of radioactive particles per unit area.

**Flux** - A rate representing the number of radioactive particles per unit area per unit of time.
**General I/O pin** - An I/O pin that can be used for general purpose inputs. This constitutes most of the input pins of the device, but would exclude clock input pins and MGT input pins, among others.

**Ghost, routing** - A routing ghost occurs when an SEU connects two wires in a design that are not driven by any net in the circuit.

**Golden device/design** - The device/design that is not subject to radiation/faults. In theory, the golden device/design should never fail.

**Hazard rate** - The rate at which a module or component fails. A constant hazard rate is typically used for radiation failures.

**Impaired Partition** - Any partition that contains a failed sub-module.

**Impaired State(s)** - The states in between the normal operation and failed operation states of a Markov chain. This is any state were any number of modules or sub-modules have failed, but the system itself has not failed.

**Implementation** - A circuit is said to be implemented on an FPGA once it has been packed, placed and routed for that FPGA. It is partially implemented if some of the packing, placement and/or routing has been defined.

**Incremental (Placement & Routing)** - Incremental placement/routing refers to a technique of performing placement/routing on a circuit that has already been partially or fully placed/routed. The proposed mitigation techniques of PCMF and RCMF use incremental placement/routing to alter an already placed/routing circuit to remove clock CMF.

**JTAG configuration manager (JCM)** - A custom device developed at BYU to handle all communication between the user and the FPGA device. During radiation testing and fault injection the JCM is used to handle the JTAG communication to detect/inject faults in the CRAM, configure the device and detect and log circuit failures.

**Lookup table (LUT)** - A memory array that replaces gate-level runtime computations by storing the precomputed function outputs in a memory. A 7-Series Xilinx FPGA is built on LUT6’s, which are LUTs that can implement any 6-input combinational function.
Mean time to failure (MTTF) - A reliability metric that expresses the average amount of time before a system fails.

Multi-bit upset (MBU) - An SEU where multiple, logically adjacent bits are simultaneously upset.

Multi-cell upset (MCU) - An SEU where multiple, physically adjacent bits are simultaneously upset.

Module - A redundant copy in a TMR system. The module refers to the entire collection of components that would constitute a singular system as well as its associated voters.

Net - A logic net in an FPGA circuit that represents a connection between two cells in the circuit's netlist.

Open, routing - A routing open occurs when an SEU disconnects a net’s source wire from it’s sink wire.

Packing - Part of the FPGA circuit implementation flow. In packing, cells are grouped together to form sites that can then be placed on the FPGA device during the placement process.

Partition - A triplicated sub-system including the associated voters.

Row bit - Any bit in a routing mux that controls a row wire. Setting a row bit in a routing mux allows the specified row wire to drive the mux output wire. A source wire of the mux does not drive the mux output unless a column bit has also been set.

Programmable interconnect point (PIP) - A programmable connection between a source and a sink wire. This is a theoretical construct that does not actually represent how the configuration memory interfaces with the routing network.

PIP Junction - See routing mux. The term PIP junction is included in this dissertation as Vivado refers to these structures as PIP junctions within the tool, but they are more generally referred to as routing muxes.
**Placement** - Part of the FPGA circuit implementation flow. In placement, groups of cells are placed on sites with the goal to optimize the amount of routing that needs to occur during the routing process.

**Pseudo-buffer** - For this work a pseudo-buffer refers to the pass-through LUT used to apply the split-IO technique.

**Radiation replay** - A form of fault injection where the logs from a radiation test are used to replay the test. The SEUs detected during each scrub cycle are used to inject faults during each fault injection iteration. This type of testing can be used to determine which combination of CRAM bits caused failures during radiation testing as well as determining which failures during the radiation test were caused by CRAM upsets.

**Radiation testing** - A form of reliability testing where a circuit is exposed to radioactive particles and the behavior of the circuit is monitored for interesting events such as SEUs or circuit failures. Common types of particles used during radiation testing include neutron, protons and heavy ions.

**Reliability** $r(t)$ - A reliability metric which represents the probability that a system has not failed at a given time $t$.

**Routing** - Part of the FPGA circuit implementation flow. In routing, each net is assigned to specific wires on the device where no wire on the device can be driven by more than one net.

**Routing Mux** - Specialized structure on the device that interfaces the configuration memory and the routing network. The routing mux decides which source wire, if any, is allowed to drive the sink wire of the mux. A source wire is chosen by setting a row and a column bit. A routing mux has many input wires, but only a single output wire.

**Routing Node** - Similar to a wire, a routing node is a Xilinx specific term that refers to each electrical node on the device. In Vivado, a wire specifically refers to an electrical node in a single tile. When that electrical node crosses a tile boundary, it becomes a new wire. The collection of electrically connected wires across tiles is called a routing node.

**Sea-level flux** - The neutron flux designated at sea-level. According to the JESD89A standard [41], the neutron sea-level flux is 13 n/cm²h.
Short, routing - A routing short occurs when two nets become connected through an SEU in a routing bit.

Simplex - The singular, unreplicated system.

Single bit failure (SBF) - A failure on an FPGA caused by a single bit upset in the configuration memory. These are subdivided into single point failures (SPF) and common mode failures (CMF).

Single event effect (SEE) - A radiation induced effect due to a single interaction. There are many types of SEEs, including SEUs, SETs and SEFIs.

Single event transient (SET) - A change in the voltage level on an electrical node in the device.

Single event upset (SEU) - A change in value in a single memory cell or multiple memory cells from a single particle interaction.

Single point failure - Any event that causes a system failure that can be attributed to an untriplicated component, such as a singular voter that is untriplicated.

Sink Wire - The sink wire is defined in a source/sink wire connection as the wire that is driven by the other (source) wire. A wire may be a source wire in one connection, but a sink wire in another connection. A sink wire also has reference in a routing mux as the wire that is the output of that mux.

Site CMF - CMF caused by the simultaneous failure in two different domain LUTs. This is caused by an SEU in a configuration bit of a SLICEM.

SLICEL - A logic slice in a Xilinx FPGA. A SLICEL contains LUTs and FFs, but unlike a SLICEM, the LUTs can only be configured as LUTs.

SLICEM - A logic slice in a Xilinx FPGA. A SLICEM contains LUTs and FFs just like other logic slices in the device, but in a SLICEM the LUTs can be configured into other memory modes such as LUTRAMs and SRLs.

Source Wire - The source wire is defined in a source/sink wire connection as the wire that drives the other (sink) wire. A wire may be a source wire in one connection, but a sink wire in another connection.
connection. A source wire also has reference in a routing mux as a wire that is an input to that mux.

**Sub-Module** - One of the redundant sub-systems in a partition. Each partition contains three sub-modules.

**Sub-system** - A system in and of itself, with defined inputs and outputs. A sub-system is part of a larger system that consists of various sub-systems connected together.

**Switchbox** - A routing tile on the FPGA that represents a group of routing muxes that share some source wires.

**Synthesis** - Part of the FPGA circuit implementation flow. During this process HDL code is compiled into a gate level schematic and then technology mapped into specific cells that can placed onto the FPGA device.

**System** - A function that has defined inputs and outputs. A system could be singular or redundant.

**Triple modular redundancy (TMR)** - A mitigation scheme where a circuit is triplicated three times and a voter placed on their outputs so only the majority value is propagated.

**Trip-IO** - A reference to TMR designs on FPGA where the I/O pins have been triplicated.

**TURTLE** - A testing platform developed for this work. The TURTLE setup uses two different FPGA boards connected via the FMC slot. One of the boards is the DUT while the other board is the golden copy. The DUT is exposed to radiation/faults while the golden copy is not and the outputs of the circuits are compared in real time to detect failures on the DUT. Setups can be stacked to increase the amount of data that can be collected.

**Voting Group** - The collection of triplicated voters on the outputs of each partition. Each group contains three voters if the voters are triplicated or 1 voter if the voters are not triplicated.

**Wire** - A physical node on the FPGA device. For a given circuit implementation the wire may or may not be used. In a valid implementation, only one net can be driving any given device wire.
G.2 Differences Between Similar Terms

**MCU vs. MBU** - An MCU represents upsets in physically adjacent bits while an MBU represents upsets in logically adjacent bits. If the logical adjacency of the bits is the same as the physical than an MBU and an MCU are the same. In newer devices it is more common for the memory to be interleaved so that logically adjacent bits are not physically adjacent [91].

**Net vs. Wire** - A net represent a connection between two cells in a circuit, while a wire is a physical/metal node on the actual device. For a valid implementation a wire may be driven by a single net, or may be undriven, but may not be driven by multiple nets. Clock CMF occurs when two clocks nets are driving source wires in the same routing mux and can be simultaneously shorted to other wires in that mux.

**PIP vs. PIP junction/Routing mux** - A PIP is a theoretical construct that represents a possible connection between a source and a sink wire while a PIP junction is the actual structure that is used to interface the configuration memory and the routing network. A PIP shows a single possible connection between a source and sink wire and is useful for describing/visualizing how wires could be connected on the FPGA.

**SPF vs. CMF** - SPF and CMF are both types of single bit failures (SBF) in that they are caused by single bit upsets that cause failure. An SBF is a SPF when the failure occurs due to an untriplicated component such as an untriplicated voter or net. An SBF is a CMF when the failure occurs in the redundant circuitry, such as when the bit controls a column in a routing mux which simultaneously causes two different clock domains to short.

**Failure vs. Fault/Error** - A fault or an error refers to some impairment to a system, circuit, etc., such as when a bit is upset during fault injection. That upset bit would be a fault in the circuit. The effect the fault/error has on the system is unknown until it has been tested. If as a result of the fault/error the circuit begins to have incorrect outputs, then that fault/error caused a circuit failure. A failure specifically refers to when a module, circuit, system, sub-system has failed and is operating incorrectly.
G.3 Acronyms

**BL-TMR** - BYU-LANL TMR

**BRAM** - Block RAM

**BUFG** - BUffer Global

**CAD** - Computer Aided Design

**CCF** - Common Cause Failure

**CLB** - Configurable Logic Block

**CMF** - Common Mode Failure

**CRAM** - Configuration RAM

**CRC** - Cyclic Redundancy Check

**DCE** - Domain Crossing Event

**DCP** - Design CheckPoint

**DRAM** - Dynamic RAM

**DSP** - Digital Signal Processor

**DUT** - Design/Device Under Test

**ECC** - Error Correction Code

**EDIF** - Electronic Design Interchange Format

**ES** - Early Split

**FIT** - Failures In Time

**FPGA** - Field Programmable Gate Array

**FF** - Flip Flop

**GFLOPS** - Giga-Floating point Operations Per Second

**GOPS** - Giga-Operations Per Second

**HDL** - Hardware Description Language

**HLS** - High Level Synthesis

**I/O** - Input/Output
JCM - JTAG Configuration Manager
JTAG - Joint Test Action Group
LANSCE - Los Alamos Neutron Science CEnter
LUT - LookUp Table
MBU - Multi-Bit Upset
MCU - Multi-Cell Upset
MGT - Multi-Gigabit Transceiver
MTTF - Mean Time To Failure
NMR - N Modular Redundancy
PCMF - Placement CMF (Incremental Placement)
PIP - Programmable Interconnect Point
RAM - Random Access Memory
RCMF - Routing CMF (Incremental Routing)
RSCP - RapidSmith CheckPoint
SBF - Single Bit Failure
SBU - Single Bit Upset
SECDED - Single Error Correct Double Error Detect
SEE - Single Event Effect
SEFI - Single Event Functional Interrupt
SET - Single Event Transient
SEU - Single Event Upset
SPF - Single Point Failure
SRAM - Static RAM
SRL - Shift Register Lookup
TCL - Tool Command Language
TCP - Tincr CheckPoint
TMR - Triple Modular Redundancy
TURTLE - Testing Ultra-Reliability Techniques using Low-cost Equipment
XDC - Xilinx Design Constraints
G.4 TMR Variations

In total 12 different TMR variations were tested for each circuit.

- **Unmitigated** - The original circuit with no mitigation techniques applied to it.

- **Common-IO** - The original circuit with TMR applied to it but without any of the input or output pins being triplicated. This design is further subdivided into 1-voter and 3-voter designs where only one voter is used for each partition or where three voters are used for every partition.

- **Split-IO** - The common-IO three voter circuit with the split-IO mitigation technique applied.

- **Split-Clock** - The common-IO three voter circuit with the split-clock mitigation technique applied.

- **Split-Clock-PCMF** - The split-clock circuit with PCMF also being applied.

- **ES** - The common-IO three voter circuit with the split-IO and split-clock techniques applied.

- **ES-PCMF** The ES circuit with PCMF also being applied.

- **Trip-IO** The original circuit with TMR applied to it where all of the input and output pins are triplicated. Like the common-IO circuits, these are also subdivided into one voter and three voter designs.

- **PCMF** - The trip-IO circuit with PCMF applied.

- **Striping** - The trip-IO circuit with striping applied.
APPENDIX H.  SCRIPTS USED IN THIS WORK

This appendix contains the code used to generate the reliability plots, find the maximum frequency and the TCL code used to generate the different TMR variations (split-clock, split-IO, etc.).
H.1 Reliability Graphs

All of the reliability graphs used in the modeling chapters were generated using Matlab scripts. The following is an example for TMR with repair and CCF. The other graphs were generated using similar code.

```matlab
1 clear all;
2 sym s t lam mu cf;
3 A_SIM = [s + lam , lam ; 0 s ];
4 A_SIM = A_SIM(1,1);
5 is_SIM = A_SIM(1,1);
6 MTF2SIM = limit(is_SIM , s , 0);
7 rt_sim_sym = 1 ilaplace(is_SIM , s , t);
8 A_T = [3*lams , 3*lams , 0 ; 0 , 2*lams , 2*lams ; 0 , 0 , s ];
9 A_Ti = A_T^-1;
10 A_Ti = A_Ti(1,1)*A_Ti(1,2);
11 MTF2TM = limit(is_T , s , 0);
12 t_sim_sym = 1 ilaplace(is_T , s , t);
13 A_TMR = [3*lams , 3*lams , 0 ; mu , 2*lams , 2*lams ; 0 , 0 , s ];
14 A_TMR = A_TMR(1,1)*A_TMR(1,2);
15 MTF2MR = limit(is_TMR , s , 0);
16 t_sim_sym = 1 ilaplace(is_TMR , s , t);
17 A_TMR_CCF = [3*lams , 3*lams , ccf ];
18 A_TMR_CCF = A_TMR_CCF(1,1)*A_TMR_CCF(1,2);
19 MTF2MR_CCF = limit(is_TMR_CCF , s , 0);
20 t_sim_sym = 1 ilaplace(is_TMR_CCF , s , t);
21 lam = 3*10^5;
22 t = logspace(1,2.50)/lam;
23 mu = 1000*lam;
24 ccf = 1*lam;
25 sim = subs(rt_sim_sym);
26 tmr_n = subs(rt_tmr_sym);
27 tmr_m = subs(rt_tmr_sym);
28 ttmr_ccf = subs(ttmr_ccf_sym);
29 ccf = 1*lam;
30 ttmr_ccf2 = subs(ttmr_ccf_sym);
31 t = t*lam;
32 figure;
33 semilogx(t , sim , 'bo' , t , ttmr_ccf , 'r' , t , ttmr_ccf0 , 'g*' , t , ttmr_ccf1 , 'c' , t , ttmr_ccf2 , 'm' , t , ttmr_ccf3 , 'k+' );
34 semilogy(t , sim , 'bo' , t , ttmr_n , 'r' , t , ttmr_m , 'c' , t , par2 , 'm' , t , par3 , 'k' );
35 set(gca , 'x tick label', '{10^1}\{10^0\} \{10^{-1}\} \{10^{-2}\}');
36 xlabel('Time (log)');
37 ylabel('Probability of Failure (log)');
38 xlim([10^0 10^2]);
```

The graphs were generated using similar code. The example here is for TMR with repair and CCF. The other graphs were generated using similar code.
H.2 Find Maximum Frequency

A TCL script was used in Vivado to find the maximum frequency for each of the generated circuits. This worked by gradually constraining the clock period in force the tool to find the best optimizations possible. The script then reported the maximum achievable clock frequency and the utilization metrics.

```tcl
set clocks [get_clocks]
set min_period 1000
set period 0
foreach clk $clocks {
    set period [get_property PERIOD [get_clocks $clk]]
}
set search "true"
set count 0
while { $search == "true" } {
    foreach clk $clocks {
        set period [get_property SOURCE_PINS [get_clocks $clk]]
        create_clock period $search $clk [get_ports Sport]
    }
    route_design unroute
    route_design
    set output [report_timing worst 1 return_string quiet]
    set lines [split $output "\n"]
    set lines [search all inline not exact $lines {}]
    set length [expr [llength $lines]]
    set line [index $lines $length]
    set words [split $line "\n"]
    set words [search all inline not exact $words {}]
    set length [expr [llength $words]]
    set slack [index $words $length]
    set new_period [expr $period + $slack]
}
```

H.3 TMR Variation Generator

A TCL script was used to generate most of the TMR variations. It would import the VHDL or Verilog of each circuit design, apply the proper constraints, dependent on the TMR variation, and then generate the bitstream. This script references two files, io_ports.xdc and dut_striping.xdc, which are shown afterwards.

```tcl
set name "<circuit_name>"
set part "xc7a200tshg4841"
set challenge_width 10
set response_width 10
set JVM_CP "/home/cthurst/workspace/BYUEditToolsCHREC:/home/cthurst/workspace/"$JVM_CP"/JHDL_provisional.jar:/home/ctthur/workspace/jars/JSAP 2.1.jar"
set JVMX_OPT "-mx3G"
```
# Early Split Constraints

1. **H.3.1 Early Split Constraints**

2. "Create pblocks"

3. create_pblock pblock_fmc_data_u_0

4. create_pblock pblock_fmc_data_u_1

5. create_pblock pblock_fmc_data_u_2

6. create_pblock pblock_fmc_data_u_3

7. create_pblock pblock_fmc_data_u_4

8. create_pblock pblock_fmc_data_u_5

9. create_pblock pblock_fmc_data_u_6

10. create_pblock pblock_fmc_data_u_7

execute java X5 {JVMX_QOPT} cp "$JVMX_CP"

edu.byu.cec.edif.edit.JEditBuild /Output/$name.$variant_name
slave.jdif o /Output/$name.$variant_name,slave jdif

execute java X5 {JVMX_QOPT} cp "$JVMX_CP"

edu.byu.cec.edif.edit.JEditNetlist /Output/$name.$variant_name
slave.jdif o /Output/$name.$variant_name,slave jdif

remove_files /get_files

# Slave Flattening

1. "Create slave"

2. create_pblock pblock_slave

3. create_pblock pblock_slave

4. create_pblock pblock_slave

5. create_pblock pblock_slave

6. create_pblock pblock_slave

7. create_pblock pblock_slave

8. create_pblock pblock_slave

9. create_pblock pblock_slave

10. create_pblock pblock_slave

11. create_pblock pblock_slave

12. create_pblock pblock_slave

13. create_pblock pblock_slave

14. create_pblock pblock_slave

15. create_pblock pblock_slave

16. create_pblock pblock_slave

17. create_pblock pblock_slave

18. create_pblock pblock_slave

19. create_pblock pblock_slave

20. create_pblock pblock_slave

21. create_pblock pblock_slave

22. create_pblock pblock_slave

23. create_pblock pblock_slave

24. create_pblock pblock_slave

25. create_pblock pblock_slave

26. create_pblock pblock_slave

27. create_pblock pblock_slave

28. create_pblock pblock_slave

29. create_pblock pblock_slave

30. create_pblock pblock_slave

31. create_pblock pblock_slave

32. create_pblock pblock_slave

33. create_pblock pblock_slave

34. create_pblock pblock_slave

35. create_pblock pblock_slave

36. create_pblock pblock_slave

37. create_pblock pblock_slave

38. create_pblock pblock_slave

39. create_pblock pblock_slave

40. create_pblock pblock_slave

41. create_pblock pblock_slave

42. create_pblock pblock_slave

43. create_pblock pblock_slave

44. create_pblock pblock_slave

45. create_pblock pblock_slave

46. create_pblock pblock_slave

47. create_pblock pblock_slave

48. create_pblock pblock_slave

49. create_pblock pblock_slave

50. create_pblock pblock_slave

51. create_pblock pblock_slave

52. create_pblock pblock_slave

53. create_pblock pblock_slave

54. create_pblock pblock_slave

55. create_pblock pblock_slave

reset_project

remove_files /get_files

# Slave Implementation

56. "Slave implementation"

57. "Slave implementation"

58. "Slave implementation"

59. "Slave implementation"

60. "Slave implementation"

61. "Slave implementation"

62. "Slave implementation"

63. "Slave implementation"

64. "Slave implementation"

65. "Slave implementation"

66. "Slave implementation"

67. "Slave implementation"

68. "Slave implementation"

69. "Slave implementation"

70. "Slave implementation"

71. "Slave implementation"

72. "Slave implementation"

73. "Slave implementation"

74. "Slave implementation"

75. "Slave implementation"

76. "Slave implementation"

77. "Slave implementation"

78. "Slave implementation"

79. "Slave implementation"

80. "Slave implementation"

81. "Slave implementation"

82. "Slave implementation"

83. "Slave implementation"

84. "Slave implementation"

85. "Slave implementation"

86. "Slave implementation"

87. "Slave implementation"

88. "Slave implementation"

89. "Slave implementation"

90. "Slave implementation"

91. "Slave implementation"

92. "Slave implementation"

93. "Slave implementation"

94. "Slave implementation"

95. "Slave implementation"

96. "Slave implementation"

97. "Slave implementation"

98. "Slave implementation"

99. "Slave implementation"

100. "Slave implementation"

101. "Slave implementation"
add_cells_to_pblock quiet pblock_fmc_data.a_8 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[8] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_9 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[9] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_10 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[10] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_12 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[12] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_13 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[13] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_14 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[14] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_15 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[15] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_16 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[16] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_17 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[17] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_18 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[18] *}]]
add_cells_to_pblock quiet pblock_fmc_data.a_19 [get_cells quiet [list {slave_io/ fmc_data_buf.a.io_buf[19] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_0 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[0] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_1 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[1] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_2 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[2] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_3 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[3] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_4 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[4] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_5 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[5] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_6 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[6] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_7 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[7] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_8 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[8] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_9 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[9] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_10 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[10] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_12 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[12] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_13 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[13] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_14 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[14] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_15 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[15] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_16 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[16] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_17 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[17] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_18 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[18] *}]]
add_cells_to_pblock quiet pblock_fmc_data.b_19 [get_cells quiet [list {slave_io/ fmc_data_buf.b.io_buf[19] *}]]
add_cells_to_pblock quiet pblock_fmc_data.c_0 [get_cells quiet [list {slave_io/ fmc_data_buf.c.io_buf[0] *}]]
REFERENCES


[31] “UltraScale architecture configuration,” Xilinx, User Guide UG570, August 2018. 16


[33] D. Lee et al., “Single-event characterization of the 28 nm Xilinx Kintex-7 field-programmable gate array under heavy ion irradiation,” in *2014 IEEE Radiation Effects Data Workshop (REDW)*, July 2014, pp. 1–5. 18

[34] ——, “Single-event characterization of the 20 nm Xilinx Kintex UltraScale field-programmable gate array under heavy ion irradiation,” in *2015 IEEE Radiation Effects Data Workshop (REDW)*, July 2015, pp. 1–6. 18


[51] B. Pratt et al., “Improving FPGA reliability in harsh environments using triple modular redundancy with more frequent voting.” 34


[63] B. White and B. Nelson, “Tincr – a custom CAD tool framework for Vivado,” in 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), Dec 2014. [Online]. Available: https://doi.org/10.1109/ReConFig.2014.7032560 79, 124


