Techniques Intended to Reduce the Impact of Program-Flow Errors on Embedded Systems

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Royan H. L. Ong
B. Eng

Department of Engineering
University of Leicester
Leicester, United Kingdom

August 2002
Techniques Intended to Reduce the Impact of
Program-Flow Errors on Embedded Systems

Royan H. L. Ong

Abstract

Embedded systems now have a key role in many safety-critical products, including automotive, aerospace and medical systems. Program-flow errors in such systems are thought to arise from environmental factors, such as electromagnetic interference. Such errors can severely disrupt the system behaviour with potentially devastating consequences. This thesis is concerned with techniques which are feasible for cost-conscious, mass-production, systems that can assist in the detection and – where possible – the correction of program-flow errors in embedded systems.

The thesis makes four important contributions. First, a detailed, quantitative assessment is carried out on ‘NOP Fill’ and ‘Function Token’. These software-based techniques are intended to reduce the impact of program-flow errors. The studies presented here are the first to rigorously assess and compare their effectiveness.

The focus of the work described here is on program-flow errors that result from corruption of the Program Counter. The second contribution made by this thesis is to develop and describe a detailed model showing the impact of PC corruption on program-flow.

The third contribution is the development of integrated hardware-based techniques that assist the detection and/or correction of program-flow errors.

The fourth contribution made by this thesis is the development of a novel mechanism to prevent register changes that might otherwise occur as a result of program-flow errors.

Overall, the results of the studies described in this thesis provide the designer of embedded systems with an effective set of tools which – when used individually or in various combinations – can greatly improve system reliability in the presence of program-flow errors.
This thesis is dedicated to my parents and sister

Ong Chee Sieng, Leow Mee Chai & Sharon Ong
Acknowledgement

This thesis is an accumulation of over 30 months of hard work, which at times, has been an incredibly stressful and lonely journey. Although I have reached my port of call, I could not have done so without all the support and encouragements that I have received along my journey, especially illuminating my path during the darkest hours.

First and foremost, I owe my sanity and gratitude to my girlfriend, Sayako Kadowaki, for all the love and support that she has given me over the years, for being with me whenever I feel low, and for the long wait that she has to endure. Without her, this thesis would not have been possible.

I would like to thank my supervisor, Dr. Michael Pont, for guiding me in my conquest of knowledge, and for all the time that he spent correcting the trivial grammar mistakes that I tend to make. My gratitude also goes out to Dr. Fernando Schlindwein, my tutor, project supervisor, and examiner. My journey would have been much harder without the light-hearted moments of my friends and colleagues, Keng Boon Goh, Woan Chiun Ng, Yi Bin, Wong, Yee Ping Teoh, Hsiu Chin Hung, Chinmay & Sumati Parikh, Aaron Cheng, Chiang Liang Lim, John Twiddle, Anita Boardman, Gerald Cheng, Emanuela Moreale and Chee Pin Tan.
Introduction

Chapter 1: Introduction
1.1 Types of embedded processors
1.2 The importance of embedded processors
1.3 The impact of program-flow errors
1.4 Causes of program-flow errors
1.5 Electromagnetic interference
1.6 Techniques to reduce the chances of processor corruption
1.7 Potential drawbacks with off-chip techniques
1.8 Objectives of this thesis
1.8.1 Quantifying software-based error detection and/or correction techniques
1.8.2 Investigating Program Counter corruption
1.8.3 Developing chip-based program-flow error detection and/or correction techniques
1.8.4 Developing mechanisms to prevent processor state changes after program-flow errors
1.9 Factors governing the feasibility of program-flow error detection and/or correction techniques
1.9.1 Overall cost overhead
1.9.2 Error detection rate
1.9.3 Error detection and/or correction latencies
1.9.4 Processing resource overheads
1.10 Conclusion

Chapter 2: 8051 family of microcontrollers
2.1 Why use the 8051 microcontroller?
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
</tr>
<tr>
<td>2.3</td>
</tr>
<tr>
<td>2.3.1</td>
</tr>
<tr>
<td>2.3.2</td>
</tr>
<tr>
<td>2.4</td>
</tr>
<tr>
<td>2.4.1</td>
</tr>
<tr>
<td>2.4.2</td>
</tr>
<tr>
<td>2.4.3</td>
</tr>
<tr>
<td>2.4.4</td>
</tr>
<tr>
<td>2.4.5</td>
</tr>
<tr>
<td>2.5</td>
</tr>
<tr>
<td>2.6</td>
</tr>
</tbody>
</table>

Part One: Evaluating Software-Based Approaches

Chapter 3: Investigating PC corruption

- 3.1 Late Multi-read Instruction Trap |
- 3.1.1 Effects of LMIT errors |
- 3.1.2 LMIT example |
- 3.2 LMIT effects on von Neumann processors |
- 3.3 Conclusion |

Chapter 4: NOP Fill and Function Tokens

- 4.1 NOP Fill |
- 4.2 Function Tokens |
- 4.3 Modelling NF and FT |
- 4.3.1 Assumptions |
- 4.3.2 Static instruction-type classification |
- 4.3.3 Classifying static instruction-types as SL, UL and DL |
- 4.3.4 Classifying 8051 instruction-types as SL, UL and DL |
- 4.3.5 Mathematical model to calculate SL, UL and DL |
- 4.4 Simulation to evaluate NF and FT |
- 4.4.1 Description of test program |
- 4.4.2 Simulation procedure |
- 4.5 Results |
- 4.5.1 Static instruction-type profile |
- 4.5.2 Calculated SL, UL and DL values |
Techniques intended to reduce the impact of program-flow errors on embedded systems

6.5.1 Simulation speed comparison between 8051Sim, 8051Sim-NG and dScope ........................................................................................................... 73

6.5.2 Performance degradation due to the simulation of custom peripherals...... 75

6.6 Conclusion ................................................................................................................... 76

Chapter 7: Modelling the impact of MIT errors on program-flow ....................... 77

7.1 Modelling MIT errors ................................................................................................. 77

7.1.1 Assumptions ........................................................................................................ 77

7.1.2 Definition and description of symbols and schematic representations.... 77

7.1.3 EMIT error scenarios .......................................................................................... 79

7.1.4 LMIT error scenarios .......................................................................................... 81

7.2 Mathematical representation of the MIT error model........................................... 82

7.2.1 Dissecting the model .......................................................................................... 82

7.2.2 Instruction-type probability ................................................................................ 83

7.2.3 Probability of EMIT (α) ..................................................................................... 83

7.2.4 Probability of LMIT (β) ..................................................................................... 84

7.2.5 Sub-dividing the calculations for λ ................................................................. 85

7.2.6 Calculating eiLMIT for D_{EMIT} and T_{EMIT2} conditions (λ_1) ............... 85

7.2.7 Calculating eiLMIT for T_{EMIT1} conditions (λ_2) ........................................ 86

7.2.8 Overall model for PC errors ............................................................................... 87

7.3 Simulating the effects of MIT errors ........................................................................ 88

7.3.1 Description of test programs .............................................................................. 88

7.3.2 Simulation procedure ........................................................................................... 89

7.4 Results and discussion ................................................................................................ 91

7.4.1 Correlation between instruction-type and recorded errors ......................... 92

7.4.2 Impact of programming methodology on error classification .................... 94

7.4.3 Correlation between simulated and modelled results .................................... 95

7.4.4 The impact of NF and FT ................................................................................... 96

7.5 Conclusion ................................................................................................................... 98

Part Three: Chip-Based Program-Flow Error Detection Techniques

Chapter 8: Existing chip-based program-flow error detection and correction techniques: A review ................................................................. 99

8.1 Overview of chip-based techniques ....................................................................... 99

8.2 Data Duplication ........................................................................................................ 100
Techniques intended to reduce the impact of program-flow errors on embedded systems

10.4.2 Evaluating ST2 ................................................................. 133
10.5 Conclusion ........................................................................... 134

Chapter 11: Byte-Cycle Tracking .................................................. 135
11.1 Functional description of BCT ................................................... 135
11.2 Estimation of error detection and correction rates .................. 138
11.3 Simulation of BCT ................................................................. 140
11.4 Results and discussion ........................................................ 140
11.4.1 Implementation ............................................................... 141
11.4.2 Evaluating BCT ............................................................... 142
11.5 Conclusion ........................................................................... 142

Chapter 12: Part Three: Discussion ............................................. 143
12.1 Comparison of ST, ST2 and BCT ............................................ 143
12.2 Implementation of State Tracking 2 ........................................ 144
12.2.1 VHDL description of ST2 ................................................ 145
12.2.2 Behavioural simulation of ST2 ........................................... 147
12.2.3 Implementation size ........................................................ 151
12.3 Comparison between NF/FT and ST/ST2/BCT ....................... 152
12.4 Evaluating chip-based techniques ......................................... 153
12.4.1 Production cost overhead .................................................. 154
12.4.2 Error detection rates ....................................................... 155
12.4.3 Error detection latency ..................................................... 156
12.4.4 Processor resources ........................................................ 157
12.4.5 Program alterations ........................................................ 157
12.4.6 Development tools .......................................................... 157
12.4.7 Interrupt handling ........................................................... 157
12.5 Porting ST, ST2 and BCT to other processor families .......... 157
12.6 Conclusion ........................................................................... 158

Part Four: Peripheral-Based Program-Flow Error Detection Techniques

Chapter 13: The Watchdog Timer .................................................. 159
13.1 The Watchdog Timer ............................................................ 159
13.2 Advantages of the WDT ....................................................... 160
13.3 Disadvantages of the WDT .................................................... 160
13.4 Other peripheral-based techniques ....................................... 162
13.5 Conclusion ........................................................................... 162
Techniques intended to reduce the impact of program-flow errors on embedded systems

16.2 Functional description .............................................................................................. 192
  16.2.1 Memory Mapping.............................................................................................. 193
16.3 Error detection procedure ....................................................................................... 194
16.4 Simulation................................................................................................................. 194
  16.4.1 Test programs .................................................................................................... 194
  16.4.2 Simulation procedure ......................................................................................... 194
16.5 Results and discussion ............................................................................................. 195
16.6 Implementation ......................................................................................................... 199
16.7 Conclusion ................................................................................................................ 201

Chapter 17: SecurePorts ............................................................................................... 202
17.1 Overview of SecurePorts ........................................................................................ 202
17.2 Description of SP ..................................................................................................... 204
17.3 Memory mapping ..................................................................................................... 205
17.4 Sub-cycle level execution ....................................................................................... 206
  17.4.1 Execution at S1P1.............................................................................................. 206
  17.4.2 Execution at S1P2.............................................................................................. 206
  17.4.3 Execution at S2P1, S2P2 and S6P1 ................................................................. 207
17.5 Simulation ................................................................................................................. 208
  17.5.1 Test program ...................................................................................................... 208
  17.5.2 Results ................................................................................................................ 209
  17.5.3 Discussion ......................................................................................................... 210
17.6 Implementation ........................................................................................................ 212
17.7 Acknowledgement ................................................................................................... 214
17.8 Conclusion ................................................................................................................ 215

Chapter 18: Benchmarking CBG, HWFT and SP .......................................................... 216
18.1 Memory mapping ..................................................................................................... 216
18.2 Immediate vectoring ............................................................................................... 217
18.3 Benchmarking CBG ................................................................................................. 218
  18.3.1 Test programs .................................................................................................... 219
  18.3.2 Benchmarking procedure .................................................................................. 219
  18.3.3 Results and discussion ...................................................................................... 221
  18.3.4 Conclusion ......................................................................................................... 223
18.4 Benchmarking HWFT .............................................................................................. 223
  18.4.1 Test programs .................................................................................................... 223
  18.4.2 Benchmarking procedure .................................................................................. 224
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.4.3</td>
<td>Results and discussion</td>
<td>224</td>
</tr>
<tr>
<td>18.4.4</td>
<td>Conclusion</td>
<td>227</td>
</tr>
<tr>
<td>18.5</td>
<td>Benchmarking SecurePorts</td>
<td>227</td>
</tr>
<tr>
<td>18.5.1</td>
<td>Test programs</td>
<td>227</td>
</tr>
<tr>
<td>18.5.2</td>
<td>Benchmarking procedure</td>
<td>228</td>
</tr>
<tr>
<td>18.5.3</td>
<td>Results and discussion</td>
<td>228</td>
</tr>
<tr>
<td>18.5.4</td>
<td>Conclusion</td>
<td>230</td>
</tr>
<tr>
<td>18.6</td>
<td>Combined CBG, HWFT and SP benchmark</td>
<td>230</td>
</tr>
<tr>
<td>18.6.1</td>
<td>Test programs</td>
<td>230</td>
</tr>
<tr>
<td>18.6.2</td>
<td>Benchmarking procedure</td>
<td>230</td>
</tr>
<tr>
<td>18.6.3</td>
<td>Results and discussion</td>
<td>231</td>
</tr>
<tr>
<td>18.6.4</td>
<td>Conclusion</td>
<td>234</td>
</tr>
<tr>
<td>18.7</td>
<td>Conclusion</td>
<td>235</td>
</tr>
</tbody>
</table>

DISCUSSION AND CONCLUSIONS

Chapter 19: Discussion and conclusion ................................. 236
19.1 Program counter corruption ........................................ 236
19.2 Software-based error detection and/or correction techniques .......... 236
19.3 Chip-based error detection and/or correction techniques ............ 238
19.4 Peripheral-based error detection and/or correction techniques .......... 239
19.5 SecurePorts .......................................................... 240
19.6 Summary of program-flow error detection and/or correction techniques .... 241
19.7 Conclusion .......................................................... 242

Chapter 20: Future Work .................................................... 243
20.1 Increasing the robustness of peripheral-based techniques .......... 243
20.2 Improving ST, ST2 and BCT’s error detection and correction rate .......... 243
20.3 Multiple tokens for HWFT ........................................ 244
20.4 Reducing SP’s overhead ........................................ 245
20.5 Enhancements to 8051Sim/8051Sim-NG ................................ 246
20.6 Enhancements to the MCU Testbench ................................ 247
Techniques intended to reduce the impact of program-flow errors on embedded systems

APPENDICES

Appendix A: References ........................................................................................................ A-1
Appendix B: Publications ....................................................................................................... B-1
Appendix C: Hardware- and design-based error detection techniques ....................... C-1
  C.1 Chip level EMI reduction ................................................................. C-1
  C.2 Circuit level EMI reduction ................................................................. C-2
  C.3 System level EMI protection ................................................................. C-3
  C.4 High-energy radiation protection ......................................................... C-3
  C.5 Conclusion .......................................................................................... C-4
Appendix D: Software- and chip-based error detection and correction techniques. D-1
  D.1 Overview ............................................................................................. D-1
  D.2 Inputs and Outputs ................................................................................. D-2
    D.2.1 Port Voting ...................................................................................... D-2
    D.2.2 Port Refreshing ............................................................................... D-3
    D.2.3 Range Rejection ............................................................................... D-3
    D.2.4 Digital Filtering ............................................................................... D-4
  D.3 RAM and SFR ..................................................................................... D-5
    D.3.1 Checksum ...................................................................................... D-5
    D.3.2 Range Rejection ............................................................................... D-5
    D.3.3 Data Duplication ............................................................................... D-6
  D.4 Program Flow ....................................................................................... D-6
  D.5 Conclusion .......................................................................................... D-7
Appendix E: User guide for 8051Sim ............................................................................... E-1
  E.1 Installation ............................................................................................ E-1
  E.2 User interface .......................................................................................... E-1
    E.2.1 ROM features .................................................................................. E-1
    E.2.2 Windows ......................................................................................... E-2
    E.2.3 Toolbox window ............................................................................... E-2
    E.2.4 Program Code window ................................................................... E-3
    E.2.5 Virtual RS232 Terminal window .................................................... E-3
    E.2.6 Port 0, Port 1, Port 2 and Port 3 windows ....................................... E-4
    E.2.7 Script Input window ........................................................................ E-4
    E.2.8 Settings window ............................................................................... E-5
    E.2.9 The Status Bar ............................................................................... E-5
  E.3 Program simulation .................................................................................. E-6
Techniques intended to reduce the impact of program-flow errors on embedded systems

E.3.1 Program loading ................................................................. E-6
E.3.2 Program execution................................................................. E-6
E.4 Breakpoints ................................................................................. E-7
  E.4.1 Adding Breakpoints........................................................... E-7
  E.4.2 Editing Breakpoints............................................................ E-8
  E.4.3 Removing Breakpoints......................................................... E-8
  E.4.4 Enabling/Disabling Breakpoints............................................. E-8
E.5 Scripting................................................................. E-8
E.6 Program Termination............................................................... E-9

Appendix F: Description of 8051Sim’s scripting functions and variables .......... F-1
  F.1 8051 Special Function Registers ............................................. F-3
  F.2 Simulator variables and constants ........................................... F-5
  F.3 Scripting Functions ............................................................... F-7

Appendix G: 8051Sim custom peripheral example ........................................ G-1
  G.1 Description of T16 ............................................................... G-1
  G.2 Script with T16 ................................................................. G-2
  G.3 Program code with T16 ...................................................... G-3

Appendix H: TE-51 ................................................................. H-1
  H.1 Description of TE-51 .......................................................... H-1
    H.1.1 States and phases .......................................................... H-1
    H.1.2 Instruction execution ...................................................... H-2
  H.2 TE-51 peripherals ............................................................... H-3
    H.2.1 SFR module ................................................................. H-4
    H.2.2 Timer 0 and Timer 1 ...................................................... H-5
    H.2.3 External interrupts ......................................................... H-5
    H.2.4 Interrupt system ............................................................ H-6
  H.3 Peripheral source codes ....................................................... H-7

Appendix I: MCU Testbench commands and scripting interface ...................... I-1
  I.1 MCU Host to MCU Testbench command format ...................... I-1
  I.2 MCU Testbench to MCU Host command format ..................... I-3
  I.3 Scripting variables and commands ........................................ I-4
    I.3.1 Testbench variables ...................................................... I-5
    I.3.2 MCU Testbench special function registers ...................... I-5
    I.3.3 Testbench commands .................................................... I-6
    I.3.4 Scripting functions ....................................................... I-7
<table>
<thead>
<tr>
<th>Appendix</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>Description of STATS</td>
<td>J-1</td>
</tr>
<tr>
<td>K</td>
<td>Field Programmable Gate Arrays</td>
<td>K-1</td>
</tr>
<tr>
<td>L</td>
<td>Integrating peripherals with processors</td>
<td>L-1</td>
</tr>
<tr>
<td>M</td>
<td>NF and FT's simulation script</td>
<td>M-1</td>
</tr>
<tr>
<td>N</td>
<td>MIT simulation scripts</td>
<td>N-1</td>
</tr>
<tr>
<td>O</td>
<td>ST, ST2 and BCT's simulation script</td>
<td>O-1</td>
</tr>
<tr>
<td></td>
<td>Simulation script for ST</td>
<td>O-1</td>
</tr>
<tr>
<td></td>
<td>Simulation script for ST2</td>
<td>O-5</td>
</tr>
<tr>
<td></td>
<td>Simulation script for BCT</td>
<td>O-9</td>
</tr>
<tr>
<td>P</td>
<td>CBG, HWFT and SP simulation script</td>
<td>P-1</td>
</tr>
<tr>
<td></td>
<td>Simulation script for CBG</td>
<td>P-1</td>
</tr>
<tr>
<td></td>
<td>Simulation script for HWFT</td>
<td>P-4</td>
</tr>
<tr>
<td></td>
<td>Simulation script for SecurePorts</td>
<td>P-8</td>
</tr>
<tr>
<td>Q</td>
<td>VHDL source for ST2, CBG, HWFT and SP</td>
<td>Q-1</td>
</tr>
<tr>
<td></td>
<td>VHDL source for ST2</td>
<td>Q-1</td>
</tr>
<tr>
<td></td>
<td>VHDL source for CBG, HWFT and SP (integrated)</td>
<td>Q-4</td>
</tr>
<tr>
<td>R</td>
<td>CD-ROM listing</td>
<td>R-1</td>
</tr>
</tbody>
</table>
# List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>General configuration of a von Neumann architecture processor</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Block-diagram representation of a typical microcontroller architecture</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Annual consumption of MPUs, MCUs and DSPs (in millions) [WSTS 2001]</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Semiconductor consumption annual growth rate [Leen et. al. 1999]</td>
<td>4</td>
</tr>
<tr>
<td>Figure 1.5</td>
<td>Electronic devices that are integrated into some high-end cars [Leen et. al. 1999].</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.6</td>
<td>The effects on registers of voltage spikes similar in characteristics to clock pulses</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.7</td>
<td>The effects of flip-flop errors on a register’s perceived value</td>
<td>8</td>
</tr>
<tr>
<td>Figure 1.8</td>
<td>Electromagnetic energy coupling methods between the source and receptor</td>
<td>10</td>
</tr>
<tr>
<td>Figure 1.9</td>
<td>Electromagnetic energy transfer modes</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Annual sales (in millions US$) for 4, 8, 16 and 32-bit microcontrollers [WSTS 2001]</td>
<td>21</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Block diagram of the 8051 microcontroller</td>
<td>22</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Generic 8051 memory layout</td>
<td>23</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Directly addressable RAM/SFR memory map</td>
<td>24</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>8051 instruction cycle layout</td>
<td>29</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Instruction fetch example</td>
<td>30</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>A program’s memory map with and without NF</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Implementation of Function Tokens</td>
<td>37</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Illustration of the modelling process</td>
<td>39</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Typical code memory map of a program implementing NF and FT.</td>
<td>41</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Code memory map of Prog’s layout</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>‘Function Token’ error detection scenarios</td>
<td>49</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>The impact of physical code memory size on SL, UL and DL for Prog_D</td>
<td>54</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>The impact of FT checks on SL, UL and DL for Prog_D (4kB code memory)</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>The impact of FT checks on SL, UL and DL for Prog_D (8kB code memory)</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>The impact of FT checks on SL, UL and DL for Prog_D (32kB code memory)</td>
<td>56</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>EMIT, LMIT and eiLMIT scenarios</td>
<td>59</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Screen-shot of 8051Sim</td>
<td>61</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Block diagram of 8051Sim</td>
<td>64</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Differences between the 8051 MCU and 8051Sim’s memory map</td>
<td>66</td>
</tr>
</tbody>
</table>

xiii
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 6.4: Interface for the non-graphical version of 8051Sim, 8051Sim-NG
Figure 6.5: Block diagram of 8051Sim-NG
Figure 6.6: The interactions of the program code and script with 8051Sim
Figure 7.1: Description of schematic diagram used in Table 7.2, Table 7.3 and Table 7.4
Figure 7.2: MIT error model
Figure 7.3: PC error simulation flowchart
Figure 9.1: ST operation cycles
Figure 9.2: Block diagram for State Tracking
Figure 9.3: 'Check' and 'Restore' operations at the 'quan-cycle' level
Figure 9.4: 'Store' operation at the 'quan-cycle' level
Figure 9.5: ST's COR and UNCOR periods during PC increment sub-cycles
Figure 9.6: ST's COR and UNCOR periods during PC writes (S6P2)
Figure 9.7: ST error simulation flow-chart
Figure 10.1: Block diagram for State Tracking 2
Figure 10.2: ST2 operation cycles
Figure 10.3: 'Checkstore' operation at 'quan-cycle' level
Figure 10.4: ST2's COR and UNCOR periods during PC increment sub-cycles
Figure 11.1: Block diagram for Byte Cycle Tracking
Figure 11.2: BCT's operations during I11-type instructions
Figure 11.3: BCT's operations during I22-type instructions
Figure 12.1: Timing diagram of ST2 detecting errors
Figure 12.2: Timing diagram of injected errors being overridden by PC writes
Figure 12.3: Timing diagram of ST2 detecting and correcting errors
Figure 12.4: Timing diagram for undetected PC errors
Figure 14.1: Block diagram of TE-51's connections
Figure 14.2: Block diagram of the MCU Testbench
Figure 14.3: Block diagram of MCU Host
Figure 14.4: MCU Testbench interface
Figure 15.1: Schematic of Code Boundary Guard
Figure 15.2: CBG's control registers
Figure 15.3: CBG error detection process
Figure 15.4: Flow chart of CBG's simulation
Figure 16.1: Block diagram of Hardware Function Tokens
Figure 16.2: HWFT's control registers
Figure 17.1: Normal and SP-implemented port connection
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 17.2: Block diagram of SecurePorts implemented on Port 1 204
Figure 17.3: SecurePort's control registers 205
Figure 17.4: SP's execution at sub-cycle S1P1 206
Figure 17.5: SP's execution at sub-cycle S1P2 207
Figure 17.6: SP's execution at sub-cycle S2P1 208
Figure 18.1: Combined CBG, HWFT and SP SFR map 217
Figure 18.2: CBG's benchmarking procedure flow-chart 220
Figure E.1: 'Fill ROM' dialogue box E-1
Figure E.2: 'Program Code' window E-3
Figure E.3: 'Virtual RS232 Terminal' window E-4
Figure E.4: 'Script Input' window E-4
Figure E.5: 'Settings' window E-5
Figure E.6: 'Breakpoint' window E-7
Figure E.7: 'Add/Edit Breakpoint' window E-8
Figure G.1: Special Function Register definitions for T16 G-1
Figure H.1: Schematic of TE-51's peripherals H-3
Figure K.1: Schematic of an FPGA configurable logic block [Van den Bout 1999] K-1
Figure K.2: Block diagram of an FPGA showing arrays of CLBs and PSMs [Van den Bout 1999] K-2
Figure L.1: Peripheral control via SFRs L-1
Figure Q.1: Schematic of ST2's implementation Q-1
Figure Q.2: Schematic of CBG, HWFT and SP's implementation Q-5
Techniques intended to reduce the impact of program-flow errors on embedded systems

List of tables

Table 1.1: Categorising techniques that reduces the level of EMI and high-energy particles, or detect/correct their effects 12
Table 2.1: Commercially available processor families 20
Table 2.2: Alternative functions for generic 8051 ports 25
Table 2.3: Interrupt vector addresses 28
Table 2.4: 8051 Instruction set statistics 29
Table 4.1: Static instruction-type classification 40
Table 4.2: Instruction byte classification 41
Table 4.3: Description of symbols used 42
Table 4.4: Instruction breakdown for Prog 47
Table 4.5: Calculated SL, UL and DL values for Prog 47
Table 4.6: Results of 1000 error simulation cycles 48
Table 4.7: Reclassified simulation results 49
Table 4.8: Comparison between calculated and simulated SL, UL and DL 51
Table 5.1: MIT susceptibility of von Neumann and Harvard processor architectures with single- and multi-read instruction sets 60
Table 6.1: Sub-module execution priority 66
Table 6.2: Execution speed comparison for 10,000,000 instruction cycles 74
Table 6.3: Comparison of simulation times 75
Table 7.1: Static and dynamic instruction-type symbols 78
Table 7.2: Outcome of ‘take-offs’ at position 1 for double-byte instructions (DEMIT) 79
Table 7.3: Outcome of ‘take-offs’ at position 1 for triple-byte instructions (TEMIT1) 80
Table 7.4: Outcome of ‘take-offs’ at position 2 for triple-byte instructions (TEMIT2) 81
Table 7.5: Outcome for all LMIT conditions 82
Table 7.6: PC error classification probability equations 88
Table 7.7: Static instruction-type profiles for test programs 92
Table 7.8: Dynamic instruction-type profiles for test programs (10,000,000 instruction cycles) 93
Table 7.9: Simulation results for 1000 error cycles 94
Table 7.10: Comparison between calculated and simulated results (%) 95
Table 7.11: Comparison of α, β and λ coefficients 96
Table 7.12: Effectiveness of NF and FT in detecting program-flow errors 97
Table 9.1: Summary of simulation error classification 117
Techniques intended to reduce the impact of program-flow errors on embedded systems

Table 9.2: Results of ST simulation 123
Table 9.3: Approximate gate count for macro blocks synthesized with Xilinx's XST 124
Table 9.4: ST’s gate count estimate 125
Table 10.1: ST2 correction rates of non-branching instruction-types 131
Table 10.2: Results of ST2 simulation 132
Table 10.3: ST2’s gate count estimate 133
Table 11.1: COR-classified ‘quan-cycles’ for various instruction-types 139
Table 11.2: Results of BCT simulation 140
Table 11.3: BCT’s gate count estimate 141
Table 12.1: Summary of ST’s, ST2’s and BCT’s error classification (average) 143
Table 12.2: Summary of ST, ST2 and BCT 144
Table 12.3: Summary of ST2's implementation gate count 151
Table 12.4: Revised estimate gate count for ST, ST2 and BCT 152
Table 12.5: Estimate gate count for ST, ST2, BCT and PM’s implementation on seven registers 154
Table 12.6: Estimated implementation overhead (% of TE-51) for combined techniques 155
Table 12.7: Estimate error detection rates for combined techniques 156
Table 15.1: Comparison between the interrupt and the immediate vectoring exception handling mechanisms 181
Table 15.2: Statistics of test programs 182
Table 15.3: Results of CBG’s simulation 185
Table 15.4: CBG's error detection latency 187
Table 15.5: Comparison between NF and CBG 188
Table 16.1: Statistics of test programs 195
Table 16.2: Results of HWFT simulation 196
Table 16.3: HWFT’s code memory and execution overheads 197
Table 16.4: HWFT’s error detection latency 198
Table 16.5: Comparison between FT and HWFT 199
Table 17.1: Results of SecurePort test 209
Table 18.1: CBG benchmarking test program statistics 219
Table 18.2: Results of CBG’s benchmarking 221
Table 18.3: Comparison between simulated and benchmarked results for CBG 222
Table 18.4: CBG's error detection latency 222
Table 18.5: HWFT benchmarking test program statistics 223
Table 18.6: Results of HWFT’s benchmarking 224
Techniques intended to reduce the impact of program-flow errors on embedded systems

Table 18.7: Comparison between simulated and benchmarked results for HWFT 225
Table 18.8: HWFT's error detection latency 226
Table 18.9: Execution overhead for test programs employing HWFT 226
Table 18.10: SP benchmarking test program statistics 228
Table 18.11: SP's benchmarking results 228
Table 18.12: SP’s error detection latency 229
Table 18.13: Test program statistics for the combined benchmark 230
Table 18.14: Results of combined CBG, HWFT and SP benchmarking 231
Table 18.15: Breakdown of CBG, HWFT and SP's detection rate for the combined benchmark 232
Table 18.16: Execution overhead for test programs employing CBG, HWFT and SP 233
Table 18.17: Implementation size comparison of processors with and without error detection 233
Table 19.1: ST, ST2, BCT and SP’s initial and subsequent implementation overhead 238
Table 19.2: Overall implementation overhead for CBG, HWFT and SP 239
Table 19.3: Summary of proposed program-flow error detection and/or correction techniques 241
Table D.1: Summary of software-based error detection and recovery techniques D-1
Table F.1: Interaction between DLL functions and 8051Sim-NG's script functions F-3
Table F.2: SFRs accessible by scripts (type 'byte') 234
Table F.3: SFR bits accessible by scripts (type 'boolean') 234
Table F.4: Simulator variables and constants 234
Table F.5: Enumerated constants (type 'integer') 234
Table H.1: Data and address operations for TE-51 H-2
Table H.2: Description of signals and buses to and from the peripheral module H-4
Table I.1: Host to MCU Testbench command format I-1
Table I.2: Module selection bit format I-1
Table I.3: MCU Testbench Commands I-2
Table I.4: Format of OBSERVE command (OBS module) I-3
Table I.5: MCU Testbench to host command format I-3
Table I.6: MCU Testbench variables and constants accessible by scripts I-5
Table I.7: SFRs accessible by scripts (type 'byte') I-5
Table R.1: Abbreviations of development tools R-1
Table R.2: CD-ROM layout for tools used in this project ('tools' directory) R-2
Table R.3: Source files for peripheral modules ('source' directory) R-2
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.4</td>
<td>Source files for test programs ('test_programs' directory)</td>
<td>R-3</td>
</tr>
<tr>
<td>R.5</td>
<td>Script and results of simulation experiments ('simulations' directory)</td>
<td>R-3</td>
</tr>
<tr>
<td>R.6</td>
<td>Benchmarking results ('benchmark' directory)</td>
<td>R-3</td>
</tr>
</tbody>
</table>
### List of listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 3.1</td>
<td>Original instruction sequence</td>
<td>32</td>
</tr>
<tr>
<td>Listing 3.2</td>
<td>Interpreted instruction sequence due to LMIT</td>
<td>32</td>
</tr>
<tr>
<td>Listing 4.1</td>
<td>Example C routine of Function Token implementation</td>
<td>38</td>
</tr>
<tr>
<td>Listing 5.1</td>
<td>Original instruction sequence</td>
<td>57</td>
</tr>
<tr>
<td>Listing 5.2</td>
<td>Interpreted instruction sequence due to EMIT</td>
<td>58</td>
</tr>
<tr>
<td>Listing 5.3</td>
<td>eiLMIT errors</td>
<td>58</td>
</tr>
<tr>
<td>Listing 6.1</td>
<td>Framework for simulating clock-multiplied custom peripherals</td>
<td>72</td>
</tr>
<tr>
<td>Listing 6.2</td>
<td>Example framework of a state machine-based custom peripheral</td>
<td>73</td>
</tr>
<tr>
<td>Listing 12.1</td>
<td>VHDL source code for modified PC implementing ST2</td>
<td>145</td>
</tr>
<tr>
<td>Listing 12.2</td>
<td>VHDL source code for ST2's CREG module</td>
<td>146</td>
</tr>
<tr>
<td>Listing 12.3</td>
<td>VHDL source for ST2's CTRL module</td>
<td>147</td>
</tr>
<tr>
<td>Listing 14.1</td>
<td>Sample script to control the MCU Testbench</td>
<td>175</td>
</tr>
<tr>
<td>Listing 14.2</td>
<td>PC error injection mechanism</td>
<td>176</td>
</tr>
<tr>
<td>Listing 15.1</td>
<td>Keil C51 definitions for CBG</td>
<td>180</td>
</tr>
<tr>
<td>Listing 15.2</td>
<td>Example setup for CBG in C</td>
<td>180</td>
</tr>
<tr>
<td>Listing 15.3</td>
<td>VHDL source for Code Boundary Guard</td>
<td>189</td>
</tr>
<tr>
<td>Listing 16.1</td>
<td>Sample C implementation of Hardware Function Tokens</td>
<td>194</td>
</tr>
<tr>
<td>Listing 16.2</td>
<td>VHDL source for Hardware Function Token’s implementation</td>
<td>200</td>
</tr>
<tr>
<td>Listing 17.1</td>
<td>Examples demonstrating SecurePorts</td>
<td>204</td>
</tr>
<tr>
<td>Listing 17.2</td>
<td>SecurePort test program</td>
<td>209</td>
</tr>
<tr>
<td>Listing 17.3</td>
<td>VHDL source for SecurePort’s implementation</td>
<td>214</td>
</tr>
<tr>
<td>Listing D.1</td>
<td>‘Port Voting’ C implementation example</td>
<td>D-2</td>
</tr>
<tr>
<td>Listing D.2</td>
<td>‘Range Rejection’ and ‘Input Averaging’ C implementation example</td>
<td>D-5</td>
</tr>
<tr>
<td>Listing G.1</td>
<td>Script to simulate T16</td>
<td>G-3</td>
</tr>
<tr>
<td>Listing G.2</td>
<td>Firmware using T16</td>
<td>G-4</td>
</tr>
<tr>
<td>Listing H.1</td>
<td>Package definition</td>
<td>H-7</td>
</tr>
<tr>
<td>Listing H.2</td>
<td>Source of te51SFR</td>
<td>H-9</td>
</tr>
<tr>
<td>Listing H.3</td>
<td>Source of te51TMR</td>
<td>H-10</td>
</tr>
<tr>
<td>Listing H.4</td>
<td>Source for te51TMR_INC</td>
<td>H-12</td>
</tr>
<tr>
<td>Listing H.5</td>
<td>Source of te51P3_STATE</td>
<td>H-13</td>
</tr>
<tr>
<td>Listing H.6</td>
<td>source of te51NOT_END_INS</td>
<td>H-14</td>
</tr>
<tr>
<td>Listing H.7</td>
<td>Source of te51INT_POLL</td>
<td>H-14</td>
</tr>
</tbody>
</table>
Techniques intended to reduce the impact of program-flow errors on embedded systems

Listing H.8: Source of te51INT
Listing J.1: Example of batch file entry
Listing M.1: Main file
Listing M.2: Initialising file (init.ini)
Listing M.3: Common function file (comfunc.ini)
Listing M.4: Abridged version of the MIT checking routines (mit.ini)
Listing N.1: Abridged version of the MIT simulation script for Prog_A
Listing O.1: Abridged version of ST's simulation script for Prog_A
Listing O.2: Abridged version of ST2's simulation script for Prog_A
Listing O.3: Abridged version of BCT's simulation script for Prog_A
Listing P.1: CBG's simulation script for Prog_A
Listing P.2: HWFT's simulation script for Prog_A
Listing P.3: SP's simulation script for Prog_A
Listing Q.1: VHDL source code for the Program Counter (PC)
Listing Q.2: VHDL source code for the Shadow Program Counter (SPC)
Listing Q.3: VHDL source code for the Cycle Register (CREG) module
Listing Q.4: VHDL source code for the Comparator (COMP) module
Listing Q.5: VHDL source code for the Control (CTRL) module
Listing Q.6: VHDL source code for the Control (CTRL) module
Listing Q.7: VHDL source code for the NOT_END_INS module
Listing Q.8: VHDL source code for the Code Boundary Guard module
Listing Q.9: VHDL source code for the Hardware Function Token module
Listing Q.10: VHDL source code for the SecurePorts module
Listing Q.11: VHDL source code for the PULSE_STRETCH module
Techniques intended to reduce the impact of program-flow errors on embedded systems

Abbreviations

ADC Analogue to Digital Converter
ALE Address Latch Enable
AMBA Advance Microcontroller Bus Architecture
ASIC Application Specific Integrated Circuit
BCT Byte-Cycle Tracking
BDM Background Diagnostic Mode
CAN Controller Area Network
CBG Code Boundary Goard
CISPR International Special Committee on Radio Interference (Comité International
Spécial des Perturbations Radioélectriques)
CLB Configurable Logic Block
CME Coronal Mass Ejection
CMOS Complementary Metal-Oxide Semiconductor
CY Cycle
DD Data Duplication
DES Distributed Electronics System
DL Dangerous Location
DLL Dynamic-Linked Library
DUT Device Under Test
DSP Digital Signal Processor
ECC Error Correcting Code
ECL Emitter-Coupled Logic
ECY Erroneous Cycle value
EDIF Electronic Design Interchange Format
eiLMIT EMIT-included Late Multi-read Instruction Trap
EMC Electromagnetic Compatibility
EME Electromagnetic Emission
EMI Electromagnetic Interference
EMIT Early Multi-read Instruction Trap
EMU Engine Management Unit
EPC Erroneous Program Counter value
EPROM Erasable Programmable Read-Only Memory
EEPROM Electrically Erasable Programmable Read-Only Memory (E²PROM)
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQC</td>
<td>Erroneous 'Quan-Cycle' value</td>
</tr>
<tr>
<td>ESC</td>
<td>Erroneous Sub-Cycle value</td>
</tr>
<tr>
<td>ESD</td>
<td>Electrostatic Discharge</td>
</tr>
<tr>
<td>FCC</td>
<td>Federal Communications Commission</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FISB</td>
<td>Fault Injection Support Board</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FT</td>
<td>Function Tokens</td>
</tr>
<tr>
<td>GCR</td>
<td>Galactic Cosmic Rays</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HWFT</td>
<td>Hardware Function Tokens</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>I²C</td>
<td>Inter-Interconnect</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>ID</td>
<td>Identifier</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Access Group</td>
</tr>
<tr>
<td>LCD</td>
<td>Liquid Crystal Display</td>
</tr>
<tr>
<td>LED</td>
<td>Light-Emitting Diode</td>
</tr>
<tr>
<td>LMIT</td>
<td>Late Multi-read Instruction Trap</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit/Byte</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up-Tables</td>
</tr>
<tr>
<td>LVTTL</td>
<td>Low-Voltage Transistor-Transistor Logic</td>
</tr>
<tr>
<td>MCM</td>
<td>Multi-Chip Module</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller (Unit)</td>
</tr>
<tr>
<td>MCK</td>
<td>Multiplied-Clock-cycle</td>
</tr>
<tr>
<td>MIT</td>
<td>Multi-read Instruction Trap</td>
</tr>
<tr>
<td>MPU</td>
<td>Microprocessor (Unit)</td>
</tr>
<tr>
<td>MR</td>
<td>Micro Rollback</td>
</tr>
<tr>
<td>MSC</td>
<td>Microsoft Scripting Control</td>
</tr>
<tr>
<td>NF</td>
<td>NOP Fills</td>
</tr>
<tr>
<td>NMI</td>
<td>Non-Maskable Interrupt</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
</tbody>
</table>
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IOB</td>
<td>Input/Output Block</td>
</tr>
<tr>
<td>OMF</td>
<td>Object Module Format</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PFEH</td>
<td>Program-Flow Error Handler</td>
</tr>
<tr>
<td>PLL</td>
<td>Phase-Locked Loop</td>
</tr>
<tr>
<td>PM</td>
<td>Parity-based Mechanism</td>
</tr>
<tr>
<td>PSM</td>
<td>Programmable Switching Matrix</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>SCY</td>
<td>Sub-cycle</td>
</tr>
<tr>
<td>SEU</td>
<td>Single Event Upset</td>
</tr>
<tr>
<td>SF</td>
<td>Scripting Functions</td>
</tr>
<tr>
<td>SFR</td>
<td>Special Function Registers</td>
</tr>
<tr>
<td>SL</td>
<td>Safe Location</td>
</tr>
<tr>
<td>SM</td>
<td>Signature Monitoring</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SP</td>
<td>SecurePorts</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>ST</td>
<td>State Tracking 1</td>
</tr>
<tr>
<td>ST2</td>
<td>State Tracking 2</td>
</tr>
<tr>
<td>SV</td>
<td>Scripting Variables</td>
</tr>
<tr>
<td>TTL</td>
<td>Transistor-Transistor Logic</td>
</tr>
<tr>
<td>UDI</td>
<td>User-definable Interrupt</td>
</tr>
<tr>
<td>UL</td>
<td>Unsafe Location</td>
</tr>
<tr>
<td>USART</td>
<td>Universal Synchronous Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td>WDT</td>
<td>Watchdog Timer</td>
</tr>
<tr>
<td>WP</td>
<td>Watchdog Processor</td>
</tr>
<tr>
<td>XRAM</td>
<td>External Random Access Memory</td>
</tr>
</tbody>
</table>
Register abbreviations

ALU    Arithmetic Logic Unit
ACC    Accumulator
CBGH   Code Boundary Guard threshold: High byte
CBGL   Code Boundary Guard threshold: Low byte
DPTH   Data Pointer High
DPTL   Data Pointer Low
DPTR   Data Pointer
EDC    Error Detection Control
EDINT  Error Detection Interrupt
EVECH  Error Vector: High byte
EVECL  Error Vector: Low byte
FTchk  Hardware Function Token Check
FTval  Hardware Function Token Identifier
IE     Interrupt Enable
IP     Interrupt Priority
IR     Instruction Register
P0     Port 0
P1     Port 1
P2     Port 2
P3     Port 3
PC     Program Counter
PCON   Power Control
PSW    Program Status Word
SBUF   Serial interface Buffer
SCON   Serial interface Control
SPCNT  SecurePort Counter
TCON   Timer Control
TH0    Timer 0 timing: high byte
TH1    Timer 1 timing: high byte
TL0    Timer 0 timing: low byte
TL1    Timer 1 timing: lower byte
TMOD   Timer Mode
Techniques intended to reduce the impact of program-flow errors on embedded systems

## Module abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREG</td>
<td>Byte Register Module</td>
</tr>
<tr>
<td>COMINT</td>
<td>Communication Interface Module</td>
</tr>
<tr>
<td>COMP</td>
<td>Comparator Module</td>
</tr>
<tr>
<td>CNT</td>
<td>Counter Module</td>
</tr>
<tr>
<td>CREG</td>
<td>Cycle Register Module</td>
</tr>
<tr>
<td>CTRL</td>
<td>Control Module</td>
</tr>
<tr>
<td>ECON</td>
<td>Error Control Module</td>
</tr>
<tr>
<td>EN</td>
<td>Enable Module</td>
</tr>
<tr>
<td>ERR</td>
<td>Error Injection Module</td>
</tr>
<tr>
<td>IDEC</td>
<td>Instruction Decoder Module</td>
</tr>
<tr>
<td>INC</td>
<td>Incrementor Module</td>
</tr>
<tr>
<td>OBS</td>
<td>Observer Module</td>
</tr>
<tr>
<td>SPC</td>
<td>Shadow Program Counter Module</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter Module</td>
</tr>
</tbody>
</table>
Symbol description

\( \alpha \) Probability of EMIT errors occurring

\( \beta \) Probability of LMIT errors occurring

\( \lambda \) Probability of eiLMIT errors occurring (Note: \( \lambda = \lambda_1 + \lambda_2 \))

\( \lambda_1 \) Probability of eiLMIT errors occurring for \( D_{\text{EMIT}} \) and \( T_{\text{EMIT2}} \)

\( \lambda_2 \) Probability of eiLMIT errors occurring for \( T_{\text{EMIT1}} \)

\( C \) Size taken by PFEH

\( D \) Double-byte instruction (Note: \( D = DN + D2 \))

\( D2 \) Double-byte instruction with 2\textsuperscript{nd} byte zero-valued

\( D_d \) Sum of double-byte one-cycle (I21) and double-byte two-cycle (I22) instructions.

\( DN \) Double-byte instruction with all bytes non zero-valued

\( DL \) Dangerous Locations

\( I11 \) Single-byte one-cycle instruction

\( I12 \) Single-byte two-cycle instruction

\( I14 \) Single-byte four-cycle instruction

\( I21 \) Double-byte one-cycle instruction

\( I22 \) Double-byte two-cycle instruction

\( I32 \) Triple-byte two-cycle instruction

\( M \) Physically implemented code memory

\( n \) Number of FT checks

\( P \) Programmed code memory (excluding error detection)

\( P^* \) Total code size include error detection (excluding NOP Fills region)

\( S \) Single-byte instruction

\( SL \) Safe Locations

\( \text{SUP} \) Sum of single-byte instructions (S) and unprogrammed ROM locations (UP)

\( T \) Triple-byte instruction (Note: \( T = TN + T23 + T3 \))

\( \bar{T} \) Average size of FT check

\( T2 \) Triple-byte instruction with 2\textsuperscript{nd} byte zero-valued

\( T23 \) Triple-byte instruction with 2\textsuperscript{nd} and 3\textsuperscript{rd} byte zero-valued

\( T3 \) Triple-byte instruction with 3\textsuperscript{rd} byte zero-valued

\( TN \) Triple-byte instruction with all bytes non zero-valued

\( UL \) Unsafe Locations

\( \text{UP} \) Unprogrammed ROM locations
Techniques intended to reduce the impact of program-flow errors on embedded systems

INTRODUCTION
This thesis is concerned with the study of techniques that are intended to reduce the impact of program-flow errors in embedded processors.

To place the discussions that follow in context, the material in this introductory chapter explains what is meant by a processor, and illustrates how ubiquitous such devices are in the modern world. Environmental threats to these processors – such as electromagnetic interference – are then considered, and the link between such threats and program-flow errors is discussed.

1.1 Types of embedded processors

A processor is an integrated circuit (IC) that can interpret and execute instructions to perform logic and arithmetic operations. The first processor – developed in 1971 by Fedrico Faggin [Wilson 2001] – was the 4004: it was used in the Intel Busicom desktop calculators [Kanellos 2001]. Since that time, in accordance with what is usually referred to as “Moore’s Law” [Moore 1965], processor complexity and computing power has doubled approximately every 18 months: this trend is expected to continue for at least another decade.

The basic operation of a processor is very simple: it performs operations on data based on a set of instructions [Hall 1989]. The processing data are either located within the processor (e.g. registers or cache memory) or in external memory elements, such as RAM (Random Access Memory). In general, the data memory is volatile: this implies that the data is lost when power is removed. By contrast, the program (sequence of instructions for specific functions) is usually held in non-volatile memory, such as ROM (Read-Only Memory).

Communication between the processor and memory is carried out by means of address, data and control buses (collection of wires of the same function). These buses allow communication and the exchange of data between the processor and other components attached to it, as shown in Figure 1.1. This processor-memory communication architecture
Techniques intended to reduce the impact of program-flow errors on embedded systems

broadly classifies a processor type: von Neumann (data and program share the same memory space) or Harvard (data and program have separate memory spaces) processors.

Figure 1.1: General configuration of a von Neumann architecture processor

There are three main kinds of processors in common use: the microprocessor (MPU), the microcontroller (MCU) and the digital signal processor (DSP). The different features of these families can be summarised as follows:

- **Microprocessors** – They can be regarded as general-purpose processors and have instruction sets that cover a wide range of operations. Examples of commercially available MPUs include the Intel Pentium, Motorola PowerPC and the Sun Sparc.

- **Digital signal processors** – DSPs have a processor core architecture that is optimised for processing digitised signals. Generally, the processor core is tailored for iterative multiply-accumulate-shift operations, and complemented with instructions that take advantage of this. Due to this difference in core architecture, DSPs are able to perform signal processing faster than equally specified MPUs. Examples of commercially available DSPs are the TMS320C54x devices from Texas Instruments.

- **Microcontrollers** – MCUs are the third commonly available form of processors. They can be best described as scaled-down MPU cores fabricated with data/program memory and peripherals (e.g. analogue-to-digital converters – ADCs, timers) on a single integrated circuit (see Figure 1.2) [Horowitz & Hill 1989, Rennels et. al. 1997]. Microcontrollers normally have smaller instruction sets that are usually the subset of commonly used MPU/DSP instructions.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Many microcontroller families also differ from MPUs in their memory architecture. MCUs such as the PICmicro and 8051 families, employ the Harvard architecture, as shown in Figure 1.2. Some MCUs (and DSPs) even have multiple data buses and can access multiple memory spaces. Most MPUs however, employ von Neumann architectures.

![Figure 1.2: Block-diagram representation of a typical microcontroller architecture](image)

The discussions in this thesis will relate mainly to microcontrollers, since such devices have a number of advantages in modern embedded designs. Their widespread use is reflected in Figure 1.3. In fact, not only do MCUs make up a huge portion of the processor market, their rate of growth exceeds that of MPUs and DSPs.

![Figure 1.3: Annual consumption of MPUs, MCUs and DSPs (in millions) [WSTS 2001]](image)

1.2 The importance of embedded processors

A good indication of the importance of embedded processors can be gained by considering the current use of such devices in the automotive sector [Leen et. al. 1999].
Economic, legislative and technological developments in this sector mean that an increasing number of road vehicles contain sophisticated distributed embedded systems (DESs), consisting of a number of microcontrollers and/or microprocessors linked by one or more computer networks. Such is the appetite for electronics components that the highest annual growth rate for semiconductor consumption is in the automotive sector, as shown in Figure 1.4.

![Figure 1.4: Semiconductor consumption annual growth rate [Leen et al. 1999]](image)

Microcontrollers and microprocessors were first used in vehicles over two decades ago after stringent exhaust emission legislations began in California in 1961 [Flis 1983]. Although processors were initially used in engine management units (EMU) to reduce exhaust emissions (case studies can be found in [Carley 1979, Cuatto et al. 1998, Flis 1983, Marley 1978]), these devices were soon incorporated to improve vehicle safety, vehicle handling and passenger comfort [Kiencke et al. 1996].

Apart from the EMU, processors have found their way into numerous automotive systems – as shown in Figure 1.5 – ranging from anti-lock braking systems [Banyai & Gerke 1996, Cruttenden 1987], active suspension systems [Ball et al. 1992], airbags [Thompson 1996] and adaptive cruise control [Martin 1995, Martin 1998] to central alarm, climatic control, GPS (global positioning system) navigation [Leen et al. 1999], car tracking, night/fog enhanced vision, radar collision detection [Martin 1998] and internet connectivity. Processors also allow system parameters to be fine-tuned for different driving schemes based on the driving condition and driver combinations [Holve et al. 1996].
Techniques intended to reduce the impact of program-flow errors on embedded systems

The wide spread processors usage meant over 60 processors (mainly microcontrollers) could be present in some vehicles, as estimated by Simsek et al. [Simsek et al. 1999], and the entire automotive electronic/electrical systems is forecast to account for between 15% and 35% of a vehicle’s cost [Bosley et al. 1995, Frank et al. 1998, Joselyn 1994].

![Diagram of electronic devices in a vehicle](image)

Figure 1.5: Electronic devices that are integrated into some high-end cars [Leen et al. 1999].

1.3 The impact of program-flow errors

Although modern road vehicles are an excellent example of how embedded processors have changed the way people live, it is just one of numerous examples of systems where processors play a vital role. In many such applications – including automotive, aerospace, industrial or medical systems – the failure of the processor has very real safety implications [Burnham & Cowling 1984, Cruttenden 1987]. It is therefore essential that potential failure mechanisms for embedded processors are understood, and that appropriate error detection and recovery mechanisms are developed.

The focus of this thesis is on program-flow errors. Such errors can arise through the corruption of the Program Counter (PC), the register which holds the address of the next instruction to be executed by the processor. Corruption of the PC is arguably the most serious of all forms of register corruption [Asenek et al. 1998, Li et al. 1984, MISRA 1995, Oberg et al]. This is because when the PC is corrupted, the next executed instruction will be
Techniques intended to reduce the impact of program-flow errors on embedded systems

taken from a code memory location chosen at random. This location may contain instructions or data. Whatever the content is, the processor will attempt to ‘execute’ the values at those locations as instructions, which can lead to temporary or permanent collapse of the normal program flow [Burnham & Cowling 1984]. The results of program-flow errors – as they are generally known – are unpredictable [Armstrong 1998], ranging from calculation errors, wrong sequence of execution to complete system failure [Armstrong 1998, Rebaudengo & Reorda 1999].

There is no evidence to suggest that the PC is any more (or less) susceptible to corruption than any other registers. The following factors make PC corruption arguably more significant than the corruption of other registers, however:

- The period for which a register is holding useful data, expressed as a percentage of the application’s execution time (referred to as the register’s ‘duty cycle’), is close to 100% for the PC. The PC’s high duty cycle increases the risk that a corrupted value will be read by the processor [Asenek et. al. 1998, Velazco et. al. 2000].

- Corruption of the PC need only occur once for a program to execute unpredictably and/or uncontrollably, which can lead to catastrophic consequences [Burnham & Cowling 1984, Niaussat 1998, Rebaudengo & Reorda 1999].

- The effect of PC corruption may not be immediate: program-flow errors may corrupt other register or data memory locations that may cause unpredictable system behaviour [Coulson 1999].

Program-flow errors are not exclusively caused by PC corruption. Indeed, corruption of the internal address bus, Stack Pointer and registers or data memory that are directly or indirectly linked with program execution, can all cause program-flow errors [Coulson 1999]. This thesis however, focuses on PC corruption.

When the PC is corrupted, one of the following scenarios occurs:

- The corrupted PC is written by the processor due to the execution of branch instructions or PC increments, before it is read (i.e. the PC is corrupted after it is read, but before it is written).

- The corrupted PC is read by the processor.

In the first scenario, PC errors that occur will be masked and no side effects will be observed [Asenek et. al. 1998, Caldwell & Rennels 2000, Velazco et. al. 2000]. In the second
Techniques intended to reduce the impact of program-flow errors on embedded systems

scenario, the corrupted PC is read by the processor and acted upon [Caldwell & Rennels 2000, Velazco et. al. 2000]. This would almost certainly lead to some form of program-flow related error. Hence, the latter scenario is the concern of this thesis.

1.4 Causes of program-flow errors

Assuming that a system's software and processor are free from design errors, most mechanisms involved in register corruption will be of an electrical nature. For example, unexpected voltage spikes may be regarded by flip-flops as legitimate clock pulses [Lee et. al. 1998] (see Figure 1.6), and may trigger them into storing the state of their inputs [Banyai & Gerke 1996, Coulson 1998]. Since their inputs may be in an undetermined state when voltage spike occurs, the flip-flops may store the wrong value [AN435 1998, Campbell 1995].

![Figure 1.6: The effects on registers of voltage spikes similar in characteristics to clock pulses](image)

The outcome of flip-flops changing state is the corruption of its perceived value, as shown in Figure 1.7. If such events happen to program-flow related registers, they can lead to program-flow errors [Burnham & Cowling 1984].

The source of 'unexpected' voltage spikes may be from components within a system itself (e.g. fast switching transistors and relay coils), or from external sources (e.g. lightning, high tension power lines) that induce voltages in to a system [Glenewinkel 1996]. Intrinsic sources are usually known in advance and can be dealt with during system design (e.g. filtering and buffering 'noisy' components). Extrinsic sources are less predictable; their strength and characteristics may vary considerably.
Techniques intended to reduce the impact of program-flow errors on embedded systems

![Diagram showing before and after register corruption with binary values](image)

Figure 1.7: The effects of flip-flop errors on a register's perceived value

One major extrinsic source of voltage spikes are those induced by electromagnetic energy. Electromagnetic energy propagates by conduction on wiring [Banyai & Gerke 1996, Glenewinkel 1996] and by radiation [Lee et. al. 1998]. Hence, voltages can be directly induced into the bond-out wires connecting the silicon with the pins, or of greater concern, the connections to the processor: this is a significantly larger target for electromagnetically-induced voltages [AN435 1998, John et. al. 2000]. In general, the phenomenon of electromagnetic energy affecting the workings of processors (or other electronic components) is known as electromagnetic interference (EMI) [Armstrong 1998, Armstrong 1999a, Armstrong 1999b, Douglass 1999]. An overview of EMI is presented in the following section.

Another well-documented mechanism of register corruption involves the accumulation of charges due to the interaction of high-energy particles (e.g. alpha particles and neutrons) with the semiconductor [Grey 2000, Tosaka et. al. 1996a, Tosaka et. al. 1996b, Tosaka et. al. 1998, Zille & Arkam 1992]. These high-energy particles originate from space (e.g. galactic cosmic rays (GCR) [Tosaka et. al. 1998], coronal mass ejection (CME) [Caldwell & Rennels 2000]) or from impurities in the semiconductor’s packaging material [Zille & Arkam 1992]. When the accumulated charge reaches the threshold that determines the state of a transistor, it can change this state, which in turn may change the state of a flip-flop [Caldwell & Rennels 2000]. The probability of such situations occurring is increasing as fabrication geometries decrease due to the reduction of the charge threshold that differentiates one logic state from another [Oberg et. al., Rebaudengo & Reorda 1999, Tosaka et. al. 1996a, Tosaka et. al. 1998]. Although this mechanism differs from electromagnetically induced voltage spikes, the end effect is the same: register corruption.

---

1 Gaisler [Gaisler 1997] showed that combinational logic could also be corrupted, not just registers
The literature suggests that only one bit is likely to be affected by high-energy particles at a particular moment (this is known as single-event upset – SEU) [Caldwell & Rennels 2000, Grey 2000, Tosaka et. al. 1998]. There is no mention of single-event upsets with EMI-induced register corruption. Due to its electrical nature, it is probable that more than a single bit could be affected by EMI at a particular moment.

It must be emphasized that there is no conclusive proof to suggest that EMI and high-energy particles are the only mechanisms of register corruption. Oberg et. al. [Oberg et. al.] mentioned that temperature seemed to play a role in CMOS (complementary metal-oxide semiconductor) latchups², a condition that is similar to SEU. EMI is regarded as the main mechanism of processor corruption, however, and a significant proportion of literature on processor corruption has been dedicated to it [Armstrong 1998, Kolodziejski & Kucinski 2000].

1.5 Electromagnetic interference

Electromagnetic energy is produced whenever charged particles, such as electrons, changes velocity [Britannica 2002]. This energy propagates through conductors (conducted emission) and space itself (radiative emission). In general, the Federal Communications Commission (FCC) and the International Special Committee on Radio Interference (CISPR – Comité International Spécial des Perturbations Radioélectriques) considers emission below 30MHz as conducted emission, and those above as radiated emission [Paul 1992].

The number of applications that use electromagnetic energy is staggering, and it forms a vital part of many systems, particularly communication-based. On the other hand, electromagnetic energy may also interfere with the workings of other systems. When this occurs, the phenomenon is termed electromagnetic interference. The effects of EMI on electronic systems – especially those containing processors – is wide-ranging, as the following examples show:

- Electronic cat-flap that rattled continuously when Microsoft Windows was started on a nearby computer [Armstrong 1998].
- A crane starting due to the strong emissions produced when another machine was started [Paul 1992].

² A condition where a flip-flop is permanently held at a particular state.
Techniques intended to reduce the impact of program-flow errors on embedded systems

- The electronically controlled gearbox of a well-known car model shifting into third gear whenever the vehicle passes under overhead power lines [Armstrong 1998].
- A remote-controlled overhead crane prematurely dropping its load, causing deaths, due to EMI from a radio [Armstrong 1998].

![Figure 1.8: Electromagnetic energy coupling methods between the source and receptor](image)

For EMI to occur, there has to be a source of electromagnetic energy (e.g. electro-mechanical relays, motors, contact breakers, high-tension wires, cellular communication base stations, microwave ovens), a propagation medium, and a receiver, as shown in Figure 1.8 [Paul 1992]. Typically for many electrical/electronic systems, a major source of electromagnetic energy is a processor’s clock (and associated circuitry), while the coupling mechanism is usually the wiring harness connecting the system with the ‘outside world’, and the printed circuit board (PCB) tracks connecting one component to another [AN435 1998, Deb 1998, Vitek 1998].

The EMI phenomenon can be classified into four categories: radiated emission, radiated susceptibility, conducted emission and conducted susceptibility\(^3\), as shown in Figure 1.9. In most cases however, more than one transfer mode would be involved (e.g. a Dodge Dakota reradiating electromagnetic energy from the engine management unit via the radio antenna [Gaul et al. 1997]).

---

\(^3\) Electrostatic discharge (ESD) is a special form of conducted and radiated susceptibility. This phenomenon is not discussed in this thesis.
Techniques intended to reduce the impact of program-flow errors on embedded systems

![Electromagnetic energy transfer modes](image)

Figure 1.9: Electromagnetic energy transfer modes

1.6 Techniques to reduce the chances of processor corruption

Commercially available processors have low tolerances to EMI [Coulson 1998, Coulson 1999, Engel et al. 1996] and radiation [Caldwell & Rennels 2000], and generally lack fault-tolerant features for highly reliable systems [Burnham & Cowling 1984, Caldwell & Rennels 2000, Rennels et al. 1997]. Although the ideal solution would be to eliminate the cause (e.g. elevated levels of EMI) rather than attempting to deal with the effects (e.g. program-flow errors, arithmetic errors [Armstrong 2000, Engel et al. 1996, MISRA 1995, Oberg et al.]), prevention is not always feasible, and it is not possible to completely remove noise; it can only be (substantially) reduced [MISRA 1994].

There are many techniques that can decrease a processor's susceptibility to EMI and high-energy particles. Before some of these techniques are discussed, they are classified into five categories listed in Table 1.1.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Technique classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware-based</strong></td>
<td>Techniques that require discrete off-chip hardware components such as capacitors, inductors and metal shields.</td>
</tr>
<tr>
<td><strong>Design-based</strong></td>
<td>Techniques that rely on component placement and track routing schemes without requiring additional components.</td>
</tr>
<tr>
<td><strong>Software-based</strong></td>
<td>Coding frameworks that are capable of detecting, and possibly correcting, anomalies in a processor’s behaviour.</td>
</tr>
<tr>
<td><strong>Chip-based</strong></td>
<td>Techniques that require logic gates and similar structures that are integrated on the processor’s die.</td>
</tr>
<tr>
<td><strong>Peripheral-based</strong></td>
<td>Chip-based techniques that can be controlled by the program.</td>
</tr>
</tbody>
</table>

Table 1.1: Categorising techniques that reduces the level of EMI and high-energy particles, or detect/correct their effects

The term ‘off-chip techniques’ collectively refers to hardware- and design-based techniques. Similarly, ‘on-chip techniques’ collectively refers software-, chip- and peripheral-based techniques. To ease discussion, design-based techniques for protecting the processor from high-energy particles (e.g. silicon-on-insulator – see Appendix C) are considered as hardware-based techniques.

Off-chip techniques can be employed in most systems. Hardware-based techniques are those that require some components (e.g. capacitors, inductors) to provide the necessary protection, which will increase the cost of each system that is produced. On the other hand, the cost of design-based techniques, which rely on the placement of components and how they are connected, will be one-off.

On-chip techniques differ from off-chip techniques in two aspects: 1) they are based within the processor itself, and 2) they are generally error detection techniques (i.e. they deal with the result of errors due to EMI or high-energy particles (or other factors)). Software-based techniques are based on the use of specific styles of coding to detect processor corruption. Chip-based techniques are those that require some electronics circuitry – typically logic gates – to perform error detection, and possible correction. Peripheral-based techniques are chip-based techniques that can also be controlled by the program (firmware).
Hardware- and design-based techniques that can reduce the levels of EMI reaching the processor are discussed in Appendix C. Software-based techniques to detect program-flow errors are discussed in Chapter 4 and Appendix D, while chip-based techniques are discussed in Chapter 8. Peripheral-based techniques are discussed in Chapter 13.

1.7 Potential drawbacks with off-chip techniques

Although design-based techniques [AN435 1998, Armstrong 1999a, Armstrong 1999b, Banyai & Gerke 1996, Bosley et. al. 1995, Deb 1998, Glenewinkel 1996, Ham et. al. 1998, Hubing et. al. 2000, John et. al. 2000, Lankford et. al. 1997, Marot & DallAgnese 1998, Simsek et. al. 1999, Steinecke & Anafang 1998, Vitek 1998, Williamson 1984] have been shown (see Appendix C) to be effective in reducing the level of EMI, they cannot actively prevent some hazards, such as high voltage transients, and should be complemented with other techniques (i.e. transients can be filtered by hardware-based techniques such as gas-discharge tubes). In addition, design-based techniques cannot protect processors from high-energy particles.

Hardware-based techniques (see Appendix C) [Campbell 1995, Carson et. al. 1997, Deb 1998, Jackson & Bleeks 1999, Klemmer 1996, Lee et. al. 1998, O’shea 1996, Soohoo & Wu 1998, Williamson 1984] have been proven to be effective in increasing a system’s noise immunity and reduce processor corruption due to high-energy particle, and should be employed as the first line of defence [Coulson 1998, Coulson 1999, MISRA 1994]. These techniques however, have some drawbacks:

- **They do not detect and correct processor errors** – Off-chip techniques can only reduce the level of EMI (and some high-energy particles) from entering a system. If processor errors occur – due to the failure of filtering hardware or from internal sources (e.g. alpha particle [Grey 2000]) – these techniques may be ineffective.

- **They increase production cost** – This is an important factor for mass-produced systems [Banyai & Gerke 1996, Leen et. al. 1999, Simsek et. al. 1999]. For systems that require protection from high-energy particle, radiation hardening – an extremely expensive process [Caldwell & Rennels 2000] – may be the only option.

- **They may reduce yield** – The increase in component count can lead to more rejected systems since there is more room for manufacturing errors.

- **They may reduce reliability** – The more components a system has, the higher is the chance that one (or more) component will fail [Bosley et. al. 1995].
Techniques intended to reduce the impact of program-flow errors on embedded systems

- **They may increase system weight, size and power consumption** – Size is a critical factor for some devices (e.g. engine sensors and handheld devices). The reduction in weight and power consumption should benefit hand-held, battery-powered, devices. An indirect advantage of these reductions is the possible increase in ‘product appeal’.

- **Their characteristics may change due to aging** – Components will degrade due to natural (or premature) ageing [Kolodziejski & Kucinski 2000]. For Example, conductive coatings may degrade when exposed to heat and humidity [Klemmer 1996] (this can lead to potentially costly maintenance). More often than not however, such ageing is not detected until abnormal situations are encountered.

- **They cannot be used on some electronic devices** – Some devices (e.g. Hall-effect sensors and photodiodes) cannot be totally shielded or screened since it will impair their performance. As a result, EMI can find its way into the system through this route, and affect the integrity of the sensors.

- **Their filtering characteristics may not be easily altered** – Hardware-based filtering schemes are generally inflexible, or not feasible, when it comes to alterations of filtering parameters after implementation.

- **They may create a false sense of security** – Designers may be inclined to believe that hardware-based techniques alone are capable of dealing with a situation. In this sense, backup techniques are not implemented, which could prove disastrous if the hardware-based techniques fail or degrade.

In many situations, on-chip techniques [John et. al. 2000, Saha 1996] may be able to address some of the disadvantages of hardware-based techniques when used as a replacement for, or as an adjunct to them. Indeed, Banyai and Gerke [Banyai & Gerke 1996] argue that software-based techniques used in vehicular applications have shown very good results. MISRA [MISRA 1994] also advocates the use of software-based techniques for automotive applications.

### 1.8 Objectives of this thesis

This thesis concentrates on the effects of program-flow errors that could have been the result of EMI-induced voltage spikes, high-energy particles and other environmental conditions. Although corruption of the Program Counter is not the only source of program-flow errors, the PC is the key register involved with program-flow, and the discussion in this thesis focuses on this register.
Microcontrollers are the focus in this thesis as they are the most widely used processor and are frequently subjected to environmental hazards such as high levels of EMI (e.g. automotive sector [Joselyn 1994, Leen et. al. 1999]). The sectors that are being targeted are those involved with mass-production safety related systems. In such sectors, cost, reliability and error detection are key factors that must be considered.

The four main objectives of this thesis are described in the following sections.

1.8.1 Quantifying software-based error detection and/or correction techniques
As briefly discussed in the Section 1.7, hardware-based techniques can reduce the chances of an MCU being affected by EMI, and thereby reduce the chances of program-flow errors. These techniques however, have their limitations.


The first objective of this thesis is to quantify the effectiveness of these existing software-based program-flow error detection and/or correction techniques.

1.8.2 Investigating Program Counter corruption
Although the effect of PC corruption is reasonably clear at the code level, questions remain. For example, what is the immediate effect of PC corruption on program-flow? Does PC corruption have the same effect on different processor architectures/families? Is there a relationship between a processor's instruction set and its vulnerability to program-flow errors?

The second objective of this thesis is to investigate the outcomes of PC corruption in detail.

1.8.3 Developing chip-based program-flow error detection and/or correction techniques
Program-flow error detection and/or correction techniques are not exclusively the domain of software-based techniques. Indeed, chip-based techniques – mainly designed for large computing systems – have been used for many years (e.g. [Beuscher & Toy 1970, Shuette &
Techniques intended to reduce the impact of program-flow errors on embedded systems

Shen 1987, Sridhar & Thatte 1982). The suitability of these techniques for embedded systems however, has never been assessed.

The third objective of this thesis is to evaluate the feasibility of existing program-flow error detection and/or correction techniques, and to develop other novel, feasible techniques.

1.8.4 Developing mechanisms to prevent processor state changes after program-flow errors

Although off-chip and on-chip techniques can prevent, detect and correct processor errors, the fact is that – due to a processor’s execution speeds and the latency of error detection systems – they could have executed some ‘critical’ code segments (i.e. tasks that significantly influence a system’s behaviour, such as those involved in controlling the amount of drugs to be delivered to a patient, [Coulson 1998]) before the errors are detected. This is particularly true for real-time systems where the speed of fault detection is a key factor [Beuscher & Toy 1970].

The last objective of this thesis is to develop some form of mechanism that will prevent register changes – particularly to registers linked to the MCU ports – before errors are detected and corrected.

1.9 Factors governing the feasibility of program-flow error detection and/or correction techniques

The technique that best fits a particular implementation should be selected based on a number of key factors. Although the factors involved – and their importance – will vary between implementation, some of the common factors are discussed here; with a bias towards the mass-produced, cost sensitive embedded systems.

1.9.1 Overall cost overhead

Cost is a major factor [Banyai & Gerke 1996, Leen et. al. 1999, Simsek et. al. 1999], especially for mass-produced systems, and should be kept to the minimum. In general, the overall cost overhead must take into account the following factors:

- The component overhead.
- The time overhead involved in implementing the technique.
- The purchase of new or modified development tools to implement a technique.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The component overhead (this includes the processor's cost overhead if chip- and peripheral-based techniques are employed) is usually small when compared to the other factors. The component cost will increase the cost of each embedded system manufactured, however. When mass-production is involved, this cost overhead can be substantial.

The time overhead involved in implementing a technique may be a one-off cost (i.e. it does not vary with the quantity of a product) if it is design-, software-, chip- or peripheral-based. Even so, this factor can be significant if a lot of time is spent on implementing and verifying a technique, more so when they are not mass-produced. On the other hand, the time overhead for hardware-based techniques not only includes the one-off cost for its design and verification, but the time spent inserting the additional components in each product must also be considered.

The overhead involved in purchasing or modifying development tools is another one-off cost. If a technique is not widely used, this cost may be significant: the tools have to be modified to support only a few (or one) products. On the other hand, if the tools could be used in other products, this potential cost will be split across them.

1.9.2 Error detection rate
A good error detection rate is vital to ensure that a significant proportion of errors are detected, and possibly corrected. This directly translates to higher levels of safety and reliability for a system.

A technique's error detection rate can be based on the overall rate, which is the ratio of errors detected to the number of errors that occurred; or the detected rate, which is the ratio of errors detected to the number of errors that could be detected. The latter is based on a technique's error detection coverage; i.e. the region where a technique can detect errors.

1.9.3 Error detection and/or correction latencies
In some applications, it is important that errors are detected as soon as possible to prevent the processor's state from changing. This is especially vital for real-time applications [Beuscher & Toy 1970]. Short error detection latency is also one of the keys to achieving a high degree of fault tolerance since it gives the processor less time to execute other instructions which may exacerbate the situation [Tamir et. al. 1988, Tamir & Tremblay 1990].
1.9.4 **Processing resource overheads**

Processors in embedded environments – especially microcontrollers – often have very limited RAM and code memory. Therefore, error detection and/or correction techniques should not impose excessive code and data memory overhead [Burnham & Cowling 1984]. The processing overhead must also be kept to the minimum in order for the processor to execute the program within the time constraints. Again, this is especially important for real-time programs.

1.10 **Conclusion**

Processors – microcontrollers in particular – are increasingly used in many systems, especially those of the embedded nature. Their widespread usage meant they are increasingly exposed to environmental threats such as electromagnetic interference, which could lead to processor damage or intermittent faults.

Although hardware- and design-based techniques do reduce such environmental threats (e.g. EMI), they are usually costly, bulky, and they do not detect errors. Hence, software-, chip- and peripheral-based techniques that could overcome some of these drawbacks were briefly introduced. This chapter ends by spelling out the objectives of this thesis.

The studies in this thesis involve the use of the 8051 microcontroller. Many issues however, apply to processors in general, unless specifically stated. An introduction to the 8051 MCU is given in Chapter 2.
It was made clear in Chapter 1 that this thesis is concerned with the impact of program-flow errors on embedded applications. To avoid the studies described here becoming too general, the remainder of this document will focus on one particular group of devices: the 8051 family of microcontrollers.

This chapter begins by explaining why the 8051 family of devices was used in the present studies. The architecture of this family of devices is then described in an appropriate level of detail.

This chapter is only intended to provide an overview of the 8051 microcontroller. Please refer to their datasheets [Atmel 1997, Siemens 1996a] for more details.

2.1 Why use the 8051 microcontroller?

The three classes of processors (microprocessors, microcontrollers and digital signal processors) were considered in Chapter 1. Of these, MCUs are the most widely used; the decision to use a microcontroller as the basis of this thesis was, therefore, easily made.

There are many companies dealing with microcontrollers, either fabricating them (e.g. Dallas Semiconductor, Intel and Atmel), or selling the microcontroller core in the form of intellectual property (e.g. Dolphin, Trenz). In many cases, companies developing products prefer to work with microcontrollers that can be ‘second sourced’ in the event of problems with one supplier. This is one reason why some families of microcontroller (such as the 8051) are manufactured by a range of different companies.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Family</th>
<th>Manufacturer</th>
<th>Size (bits)</th>
<th>Instruction length</th>
</tr>
</thead>
<tbody>
<tr>
<td>8051</td>
<td>Intel, Atmel, Infineon etc.</td>
<td>8</td>
<td>1 to 3 bytes</td>
</tr>
<tr>
<td>C166 (ST0)</td>
<td>Infineon, ST</td>
<td>16</td>
<td>1, 2 words</td>
</tr>
<tr>
<td>ARM7</td>
<td>Atmel, Samsung, Triscend etc.</td>
<td>32</td>
<td>1 dword</td>
</tr>
<tr>
<td>PIC 16Cxxx</td>
<td>Microchip</td>
<td>8</td>
<td>1 word</td>
</tr>
<tr>
<td>PIC 18Cxxx</td>
<td>Microchip</td>
<td>8</td>
<td>1, 2 words</td>
</tr>
<tr>
<td>AVR</td>
<td>Atmel</td>
<td>8</td>
<td>1, 2 words</td>
</tr>
<tr>
<td>H300L</td>
<td>Hitachi</td>
<td>8</td>
<td>1, 2 words</td>
</tr>
<tr>
<td>H8S</td>
<td>Hitachi</td>
<td>16</td>
<td>1, 2 words</td>
</tr>
<tr>
<td>COP8</td>
<td>National Semiconductor</td>
<td>8</td>
<td>1 to 3 bytes</td>
</tr>
<tr>
<td>68HC05</td>
<td>Motorola</td>
<td>8</td>
<td>1 to 3 bytes</td>
</tr>
<tr>
<td>68HC08</td>
<td>Motorola</td>
<td>8</td>
<td>1 to 4 bytes</td>
</tr>
<tr>
<td>68HC11</td>
<td>Motorola</td>
<td>8</td>
<td>1 to 5 bytes</td>
</tr>
<tr>
<td>ST6</td>
<td>SGS-Thompson</td>
<td>8</td>
<td>1 to 3 bytes</td>
</tr>
<tr>
<td>ST7</td>
<td>SGS-Thompson</td>
<td>8</td>
<td>1 to 6 bytes</td>
</tr>
</tbody>
</table>

Table 2.1: Commercially available processor families

Of the commercially available family of microcontrollers, such as those shown in Table 2.1, the 8051 family of microcontrollers was chosen as the basis of the studies carried out in this thesis for the following reasons:

- The 8051 is one of the most widely used MCU in the current market. Therefore, improvements in understanding of program-flow errors for this device is likely to be of widespread benefit.

- The 8051 family has what is known as ‘multi-read instructions’, which is examined in detail in Chapter 3 and Chapter 5. Briefly, ‘multi-read instructions’ are instructions that are wider than the processor’s data bus. Hence, they have to be read in more than one clock-cycle\(^4\). The instruction set of most MCU families (see Table 2.1), and some MPUs and DSPs, are of this nature.

- Development tools for the 8051 family were available at the time the work described in this thesis was carried out.

---

\(^4\) ‘Multi-read instructions’ were referred to as ‘multibyte instructions’ in earlier publications by the author.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Although the generic 8051 architecture – introduced over 20 years ago – does not have the computational power of modern architectures (e.g. PIC, AVR), it is still more than sufficient for many applications that are not processing intensive\(^5\). Indeed, Figure 2.1 shows 8-bit MCUs – generally made up of older architectures – still leads the field by a comfortable margin.

![Figure 2.1: Annual sales (in millions US$) for 4, 8, 16 and 32-bit microcontrollers [WSTS 2001]](image)

2.2 Functional description of the 8051 microcontroller

The 8051 family of microcontrollers – first developed by Intel in the early 1980s [Vault 2001] – are manufactured by many companies such as Atmel, Infineon and Dallas Semiconductors. Although the 8051 compatible MCUs offered by these companies implement the generic peripherals, most of them are also integrated with manufacturer-specific peripherals such as SPI (serial peripheral interface) and CAN (controller area network) [Kattwinkel 1995, Navet 1998]. This thesis however, concentrates on the generic 8051 MCU. The block diagram of a generic 8051 microcontroller is illustrated in Figure 2.2.

There are two main buses within the processor core: the data bus and the address bus. At the heart of the microcontroller is the arithmetic/logic unit (ALU), which is responsible for all the mathematical and logic operations. The accumulator (ACC), B register, TMP1, TMP2 and Program Status Word (PSW) are the ancillary registers of the ALU. Note: TMP1 and TMP2 are neither addressable, nor memory-mapped.

---

\(^5\) Some manufacturers (e.g. Dallas Semiconductor) offer 8051 compatible MCUs that has higher performance.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 2.2: Block diagram of the 8051 microcontroller

All the registers connected to the address bus deal with program flow, i.e. the instruction execution sequence and the code memory (internal/external) access.

2.3 Memory layout and access

The 8051 MCU is a Harvard architecture device with an 8-bit data bus and a 16-bit address bus. It has 256 bytes and 128 bytes of integrated RAM and special function registers (SFRs) respectively. The MCU can also address up to 64kB of external data memory (XRAM) and code memory (ROM) respectively. Its memory map is shown in Figure 2.3.
Techniques intended to reduce the impact of program-flow errors on embedded systems

2.3.1 ROM and XRAM

The 8051 MCU can address up to 64kB of code memory implementable as EPROM (erasable programmable read-only memory), EEPROM (electrically erasable programmable read-only memory) or recently FlashROM. This code memory can either be integrated as part of the MCU, or can be an external chip. In common with most MCUs, the integrated code memory is usually much less than the maximum addressable.

For external code and data memory access (ROM and XRAM respectively), Port 0 (P0) and Port 2 (P2) are used to form the external address/data bus, as shown in Figure 2.2. To save the number of ports required, P0 is time-multiplexed between the lower address lines and the data; the address latch enable signal (ALE) indicates whether data or address is present at P0’s output. The PSEN signal (see Figure 2.2) determines if program or data memory is to be accessed while the read (RD) and write (WR) signals are derived from Port 3 bit 7 (P3.7) and bit 6 (P3.6) respectively.

For XRAM memory, two SFR locations (DPTL and DPTH, collectively known as DPTR – see Figure 2.2) are used as a 16-bit data pointer to determine the XRAM location to read or write. The MOVX instruction is then used to read and write the location pointed by DPTR registers. During XRAM reads/writes, the PSEN signal is held high.

---

6 RD and WR is only for external memory (XRAM) access.
2.3.2 RAM and SFR

As shown in Figure 2.3, the upper 128 bytes (0x80 to 0xFF) of RAM are shared with the SFR: indirect addressing accesses the former while the SFRs are directly addressable. The lower 32 bytes of RAM (0x00 to 0x1F) form four banks of 8 byte-wide registers (R0 to R7), as shown in Figure 2.4. The desired register bank is selected by two bits located in the PSW. The first two registers (R0 and R1) also form two separate data pointers for the indirectly addressed RAM (0x80 to 0xFF).

Thirty-two RAM/SFR bytes are bit addressable: meaning each bit has an associated address. These bit-addressable locations, as shown in inverted colours in Figure 2.4, are located between 0x20 and 0x2F of RAM, and SFR locations evenly divisible by 0 or 8. Each bit at these locations have separate 8-bit addresses and can be directly manipulated with special
8051 instructions (e.g. SETB). Note: the SFR locations shown in Figure 2.4 are only those present on the generic 8051. Unused SFR locations (shaded grey) do not implement registers.

2.4 Generic peripherals
The generic 8051 peripherals include four 8-bit quasi bi-directional input/output ports, two configurable general-purpose 16-bit timers/counters and one universal synchronous asynchronous receiver transmitter (USART). Five maskable interrupt sources with two level of priority are also present: two for the timers, two for external interrupts (via input/output – I/O – pins), and one for the USART module. All these peripherals are configurable to work in different modes, as described in their respective sub sections.

2.4.1 Input/Output ports
Of the four 8-bit I/O ports, three of them can alternatively function as signals for other peripherals. Two ports, P0 and P2, are alternatively used as address/data buses to access external data and program memory. P3 is used by the timers as control inputs, by the interrupt system as external interrupt pins, by the USART as the transmit and receive lines, and as the external memory’s read and write signals. Table 2.2 summarises the port’s alternative functions.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>Alternate function</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0&lt;7:0&gt;</td>
<td>-</td>
<td>Data/Address bus (LSB)</td>
</tr>
<tr>
<td>P2&lt;7:0&gt;</td>
<td>-</td>
<td>Address bus (MSB)</td>
</tr>
<tr>
<td>P3.0</td>
<td>RXD</td>
<td>USART receive input</td>
</tr>
<tr>
<td>P3.1</td>
<td>TXD</td>
<td>USART transmit output</td>
</tr>
<tr>
<td>P3.2</td>
<td>INT0</td>
<td>External interrupt 0</td>
</tr>
<tr>
<td>P3.3</td>
<td>INT1</td>
<td>External interrupt 1</td>
</tr>
<tr>
<td>P3.4</td>
<td>T0</td>
<td>Timer 0 external count input</td>
</tr>
<tr>
<td>P3.5</td>
<td>T1</td>
<td>Timer 1 external count input</td>
</tr>
<tr>
<td>P3.6</td>
<td>WR</td>
<td>External data memory write strobe</td>
</tr>
<tr>
<td>P3.7</td>
<td>RD</td>
<td>External data memory read strobe</td>
</tr>
</tbody>
</table>

Table 2.2: Alternative functions for generic 8051 ports

2.4.2 Timers/Counters
Each timing unit consists of two 8-bit registers (TL0 and TH0 for Timer 0, TL1 and TH1 for Timer 1) that hold the count value. Two other SFRs (TMOD and TCON) are used to determine the mode of the timing units and to control the timing register increments. Both
Techniques intended to reduce the impact of program-flow errors on embedded systems

timing units function identically in modes 0, 1 and 2. In mode 3, the units are interlinked and can be used as three separate timers/counters; this would be discussed later.

Timer 0\textsuperscript{7} can be enabled by the program, or externally via the INT0 signal (see Table 2.2). When Timer 0 is enabled, it is either increment at $1/12^{th}$ the oscillator clock frequency (1 increment per instruction cycle), or by the high-to-low transition of the TO signal (counter mode). The timer modes – which determine how Timer 0 is incremented – can be summarised as follows:

- **Mode 0** – the timing registers form an 8-bit timer/counter (TH0) and a 5-bit prescaler (TL0).
- **Mode 1** – both 8-bit timing registers are concatenated to form a 16-bit timer/counter.
- **Mode 2** – TL0 forms an 8-bit timer/counter that is reloaded with the value stored in TH0 whenever TL0 overflows.

Whenever the timing registers overflow, the interrupt request flag is set. This flag (TF0 for Timer 0, TF1 for Timer 1) is polled by the interrupt system, which determines if an interrupt is to be generated.

In Mode 3, Timer 0 functions as two separate 8-bit timers/counter. TL0 is used as an 8-bit timer/counter, while TH0 can only be used as a timer. TH0 is enabled by Timer 1's circuitry instead, and uses Timer 1's interrupt request flag. If Timer 1 is not in Mode 3, it runs freely in whatever mode it was in without its control circuitry, and does not set its interrupt request flag when it overflows. If Timer 1 is in Mode 3, it stops.

### 2.4.3 USART

The USART in the 8051 is controlled by the SCON and SBUF SFR. Although the receive and transmit registers share the same SFR address (SBUF), they are actually two physical registers. Transmission is initiated by writing to SBUF register. Transmission and reception is governed by the USART's operating modes, configured and controlled by the SCON register.

---

\textsuperscript{7} Timer 0 and Timer 1 operate in the same manner in modes 0, 1 and 2; hence, only Timer 0 is described.
The USART has four operating modes as summarised in the following list:

- **Mode 0** – data enters and exits through RXD (P3.0) while TXD (P3.1) is the bit-shift clock. 8 data bits are transmitted (least significant bit – LSB – first) and the baud rate is 1/12\(^{th}\) the oscillator frequency.

- **Mode 1** – 10 bits – 1 start, 8 data, and 1 stop bit – are transmitted and received. The baud rates are variable and controlled by Timer 1’s overflow rate.

- **Mode 2** – 11 bits – 1 start, 8 data, 1 programmable and 1 stop bit – are transmitted and received. The programmable bit can be used as the 9\(^{th}\) data bit. The baud rate is fixed at either 1/16\(^{th}\) or 1/32\(^{nd}\) of the oscillator frequency.

- **Mode 3** – the same as Mode 2 apart from the baud rates being governed by Timer 1’s overflow rate.

Upon the completion of each transmission or reception, its interrupt request flag is set. The USART has two flags – TI (transmit) and RI (receive) – that are ORed to form a single interrupt. It is up to the program to distinguish whether an interrupt is caused by transmission or reception.

### 2.4.4 External interrupts

Two external interrupts are present through pins INT0 and INT1 (see Table 2.2). The state of these pins is polled by the interrupt system (see next sub section) at SSP2 (see Section 2.5) and can be configured to be edge or level triggered.

In edge-triggered mode, the interrupt request flag is set when a high-to-low transition on the respective external interrupt pins is detected. In level-triggered mode however, the external interrupt pin would set the interrupt request flag whenever it is low, i.e. the state of interrupt request flag is opposite of the pin’s state.

### 2.4.5 Interrupt system

Every interrupt source has three flags associated with it:

- An interrupt request flag to denote an interrupt that has occurred.
- An interrupt enable flag to determine if the source would generate interrupts.
- An interrupt priority flag to set its priority (low or high priority).

The interrupt enable and priority flags are located in the IE and IP SFRs respectively (see...
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 2.4). The interrupt request flags are usually in SFRs that belong to respective peripherals. Note: interrupt request flags are only read by the interrupt system when the global interrupt bit (EA - bit 7 in IE SFR) is set.

The interrupt system reads the interrupt request flags once every instruction cycle at S5P2 (see Section 2.5). When an interrupt request flag is detected at S5P2 (i.e. it is high), it is polled at S5P2 of the following instruction cycle. Only when it remains set during the polling cycle (i.e. high for two consecutive instruction cycles) will an interrupt be recognised.

A recognised interrupt can still be blocked by the following conditions:
- An interrupt of equal or higher priority is in progress.
- An instruction is in progress (for multi-read instructions).
- A RETI instruction is in progress or any instruction that is writing to IE or IP.

The first condition ensures that only high priority interrupt sources can interrupt low priority sources. The second condition prevents vectoring to interrupt service routines (ISR) while a multi-read instruction is still in progress. The last condition gives the processor time to recognise changes to the interrupt system. In this condition, one additional instruction is executed before a recognised interrupt source is vectored to the ISR.

Once all the conditions are met, interrupt vectoring takes place. This procedure is similar to executing an 8051 'Long Call' (LCALL) instruction. The vector location and flags for the interrupt sources are shown in Table 2.3. Once vectoring completes (after the LCALL instruction), the interrupt system resets the interrupt request flag of the respective source, unless it is the USART or external interrupt’s (level-triggered mode only) interrupt request flags.

<table>
<thead>
<tr>
<th>Interrupt source</th>
<th>Vector address</th>
<th>Int. request flag</th>
<th>Int. enable flag</th>
<th>Int. priority flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Bit add.</td>
<td>Name</td>
<td>Bit add.</td>
</tr>
<tr>
<td>External interrupt 0</td>
<td>IE0</td>
<td>0x89</td>
<td>EX0</td>
<td>0xA8</td>
</tr>
<tr>
<td>Timer 0 overflow</td>
<td>TF0</td>
<td>0x8D</td>
<td>ET0</td>
<td>0xA9</td>
</tr>
<tr>
<td>External interrupt 1</td>
<td>IE1</td>
<td>0x8B</td>
<td>EX1</td>
<td>0xAA</td>
</tr>
<tr>
<td>Timer 1 overflow</td>
<td>TF1</td>
<td>0x8F</td>
<td>ET1</td>
<td>0xAB</td>
</tr>
<tr>
<td>USART</td>
<td>RI/TI</td>
<td>0x98/9</td>
<td>ES</td>
<td>0xAC</td>
</tr>
</tbody>
</table>

Table 2.3: Interrupt vector addresses
2.5 Instruction set and execution

Each 8051 instruction cycle comprises of six states, each state having two phases. As each phase is equivalent to an oscillation cycle (the term 'sub-cycle' is used in this thesis), the length of one instruction cycle is equivalent to 12 sub-cycles. The states and phases are denoted as $S_x$ and $P_y$, where $x$ is 1 to 6 and $y$ is 1 or 2 (e.g. $S_1P_2$, $S_5P_1$) as shown in Figure 2.5.

![Figure 2.5: 8051 instruction cycle layout](image)

The 8051 instruction set comprises of 255 instructions. Of the 111 distinct instructions$^8$, 64 are single-cycle instructions, 45 double-cycle instructions and two are four-cycle instructions. Forty-nine instructions require a single byte, 46 are double byte instructions and 16 are triple byte instructions [Boyet & Katx 1982]. Table 2.4 categorises the 8051 instruction set into the various byte-cycle combinations (e.g. 111, 132). Note: the ‘Total’ column includes all variants of each instruction.

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Length (bytes)</th>
<th>Execution time (cycles)</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I11</td>
<td>1</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>I12</td>
<td>1</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>I14</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>I21</td>
<td>2</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>I22</td>
<td>2</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>I32</td>
<td>3</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 2.4: 8051 Instruction set statistics

---

$^8$ Instructions in the form of 'MOV R0, A' and 'MOV R7, A' are considered the same.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Instruction bytes are only read at sub-cycles S1P2 and S4P2 (see Figure 2.5). If code memory reads are unnecessary (e.g. at S4P2 for I11-type instructions), a byte is still read, but is discarded. Under such situations, the PC is not incremented, and the instruction register (IR)\(^9\) is not updated. For multi-read instructions, the operands are read one after the other at S1P2, S4P2 and, if necessary, S1P2 of the next clock-cycle (I32-type instructions). The instruction fetch cycle is illustrated in Figure 2.6.

2.6 Conclusion
The decision to use the 8051 family of MCUs as the basis of this thesis was justified at the start of this chapter. Briefly, the 8051 family of MCUs is the one of the most widely used processor, and the necessary tools are at the disposal of this project. The second half of this chapter describes in the generic 8051. The 8051’s instruction set and fetch/decode cycle are described in detail: this is important when it comes to dealing with program-flow errors.

\(^9\) The Instruction Register is a temporary buffer to store an instruction’s byte [Beuscher & Toy 1970].
PART ONE
EVALUATING SOFTWARE-BASED APPROACHES

The effects of PC corruption on program-flow were discussed in *Chapter 1*. This was followed – in *Chapter 2* – by a description of the 8051 microcontroller, the hardware platform that will be used throughout this thesis.

In *Part One*, software-based program-flow error detection techniques described by other authors are discussed. These techniques are then modelled and a quantitative assessment of their effectiveness is carried out.
The system-level effects of program-flow errors were discussed in Chapter 1. Although program-flow errors are not exclusively due to PC corruption, it was suggested that the PC’s high duty cycle [Asenek et. al. 1998] and direct relationship with program-flow makes it the primary source of such errors.

In this chapter, the effects of PC corruption on program-flow are investigated. This understanding is necessary to evaluate the effectiveness of the program-flow error detection techniques that are presented in the next chapter.

3.1 Late Multi-read Instruction Trap

Assuming Harvard architectures processors for the time being, some processor families (e.g. PIC16x, see Table 2.1) have instructions that are read in a single read cycle, i.e. the entire instruction plus the operands can be read by the processor at once. Such single-read instructions – as they are called – can be read in one cycle as their width (in bits) is the same, or smaller than, the processor’s data bus. Due to this fact, they do not occupy more than one code memory location; hence, the PC will always point to an instruction, and not one of its operands. Processor families with only single-read instructions have what is termed single-read instruction sets.

Processor families with multi-read instruction sets (e.g. PIC18x, 8051, ARM) have some (or all) instructions that must be read more than once by the processor since they are wider than the processor’s data bus. Multi-read instructions will also require more than one code memory location. Thus, it is possible for the PC to point to code memory locations that are not the first byte of an instruction [Li et. al. 1984]. Coupled with the fact that processors cannot determine the overall instruction boundary (processors only determine the boundary of the next instruction while executing the current instruction), they will execute any value in the code memory locations pointed by the PC as instructions [Burnham & Cowling 1984]. Therefore, if the PC erroneously points to a code memory location that is not the boundary of
Techniques intended to reduce the impact of program-flow errors on embedded systems

an instruction, the processor will execute 'new' instructions: instructions that are not part of the original program [Li et. al. 1984]. This phenomenon is known as the 'Late Multi-read Instruction Trap' (LMIT) error condition, 'late' in the sense that the error occurs where the PC points. The terms 'LMIT errors' and 'LMIT error conditions' are interchangeably used in this thesis.

Note: if the PC corruption branches program-flow from, say, the boundary of the first and second instruction (i.e. the first instruction has been executed, but not the second) to location 0x0106 (see Listing 3.1), the LMIT error condition does not occur. This situation is termed 'Normal error condition', or just 'Normal', in this thesis.

3.1.1 Effects of LMIT errors
At the instruction level, the effect of LMIT errors is the execution of instructions that have not been explicitly coded. From the program's point of view, this translates to program-flow errors, which can result in unpredictable system behaviour [Burnham & Cowling 1984]. This may eventually lead to catastrophic system failure, injury or even death. Hence, LMIT errors should not be taken lightly.

3.1.2 LMIT example
An example of LMIT error condition occurring – based on the 8051 instruction set – is illustrated with the help of Listing 3.1 and Listing 3.2. The original instruction sequence is shown in Listing 3.1. If, for example, the PC is corrupted with the value of 0x0105 after reading the last byte of the first instruction, the processor would then interpret and execute the following instructions based on the values from location 0x0105 onwards. The resultant instruction sequence is shown in Listing 3.2.

0100 759850 MOV 98H,#50H
0103 438920 ORL 89H,#20H
0106 758DAB MOV 8DH,#ABH
0109 43FF ORL A,#FFH

Listing 3.1: Original instruction sequence

0100 759850 MOV 98H,#50H
0105 20758D JB 75H,8DH
0108 AB43 MOV R3,43H
010A FF MOV R7,A

Listing 3.2: Interpreted instruction sequence due to LMIT

In this situation, different instructions than that intended would have been executed by the processor ('new' instructions created). The problem may not end here: the instruction
Techniques intended to reduce the impact of program-flow errors on embedded systems

boundaries may no longer be the same (skewed) as the original, which can lead to the execution of more ‘new’ instructions. Some of these ‘new’ instructions however, may not be valid for certain instruction sets and some processors can trap these [Li et. al. 1984]; this is unlikely for the 8051 family.  

3.2 LMIT effects on von Neumann processors

LMIT errors are not only confined to Harvard architecture processor families. Indeed, von Neumann architecture processor families are more susceptible to LMIT errors as data and code are stored in the same memory space. Any PC corruption branching program-flow to memory locations containing data would therefore cause the processor to execute data values as instructions. Although the PC is not pointing to memory locations between multi-read instructions (e.g. location 0x0107 in Listing 3.1), the phenomenon is also termed LMIT since ‘new’ instructions are created.

LMIT errors affect von Neumann architecture processor families with and without multi-read instruction sets. Processor families in the latter category will be less susceptible since branches to memory locations holding code do not create ‘new’ instructions and will not cause LMIT errors.

LMIT errors can also affect Harvard architecture processor families with single-read instruction sets when data stored in code memory (constants) are misinterpreted as instructions; hence, creating ‘new’ instructions. The chances of this situation occurring are slim: it is based on the ratio of code memory locations holding constants to the program size.

---

10 Of the possible 256 instructions, the 8051 MCU has 255 valid instructions.
3.3 Conclusion
The effect of PC corruption is not as simple as initially thought. Processor families with multi-read instruction sets have the ability to cause LMIT error conditions. LMIT error conditions occur when the PC points to code memory locations that are not instruction boundaries, or code memory locations that contain data, thus causing the processor to execute 'new' instructions. The end effect would be program-flow errors.

LMIT errors affect processor architecture/instruction set combinations differently. For Harvard architecture processors, those with multi-read instruction sets – such as the 8051 – are affected by LMIT errors. On the other hand, single-read instruction set families are generally unaffected by LMIT errors.
PC corruption, as discussed in the previous chapter, not only leads to erroneous branches, it may also cause ‘new’ instructions to be executed as the result of instruction misinterpretation: a condition termed the ‘Late Multi-read Instruction Trap’. This will result in program-flow errors. Due to the impact of such errors, this thesis focuses on techniques to detect and/or correct them.

In this chapter, two software-based techniques that have been presented by various authors [AN435 1998, Banyai & Gerke 1996, Burnham & Cowling 1984, Campbell 1998, Coulson 1998, Coulson 1999, Li et. al. 1984, Niaussat 1998] are discussed. ‘NOP Fills’ (NF) and ‘Function Tokens’ (FT) – as they are called – are described in detail, followed by the model used to evaluate their effectiveness. This model takes the effect of LMIT errors – discussed in Chapter 3 – into consideration. Experiments are then carried out to evaluate NF and FT’s effectiveness. The discussion of the results concludes this chapter.

Note: as the main concentration of this thesis is on program-flow errors, only NF and FT are described. Other software-based error detection and/or correction techniques are outlined in Appendix D. The use of the Watch Dog Timer (WDT) as a means of detecting program-flow errors is discussed in Part Four.

4.1 NOP Fill

NOP Fill [AN435 1998, Burnham & Cowling 1984, Campbell 1995, Campbell 1998, Coulson 1999, Niaussat 1998] involves filling the unprogrammed (empty) code memory locations with a specific value. Typically, program code is contiguously programmed into code memory from location 0x0000, leaving some contiguous unprogrammed code memory
Techniques intended to reduce the impact of program-flow errors on embedded systems

locations at the end of the physical code memory. Under normal circumstances, these locations contain the value of 0xFF, equivalent to the ‘MOV R7, A’ 8051 instruction\(^1\).

What NF does is to fill these unprogrammed code memory locations with the value of 0x00, equivalent to the ‘No Operation’ (NOP) 8051 instruction; hence its name. The contiguously filled locations are termed the NF region. A special function, the ‘Program-Flow Error Handler’ (PFEH), is located at the end of the NF region to correct program-flow errors. In some cases, it is sufficient for the PFEH to stop the processor when errors are detected; in other cases it is important to place the processor in a known state [Beuscher & Toy 1970].

Under normal processor execution, the NF region and the PFEH are unreachable. If a PC error branches execution to the NF region however, the processor would then execute NOP instructions until it reaches PFEH; hence, the error will be corrected. When such a situation occurs, the state of the processor – apart from the PC and timing related registers – remains the same. Figure 4.1 shows a schematic of the memory layout when NF is implemented.

![Figure 4.1: A program's memory map with and without NF](image)

If NF is not implemented, program-flow branches to unprogrammed code memory locations will cause execution of the ‘MOV R7, A’ instruction until the last physically implemented code memory location. Due to memory aliasing, program-flow will continue at code memory location 0x0000, restarting the program\(^2\). Not all systems can tolerate this.

---

\(^1\) Move the value stored in the Accumulator to register R7.

\(^2\) This is not equivalent to a processor reset. Some internal registers (e.g. interrupt related) may be at states different from an actual reset.
An alternative approach to NOP-filling unprogrammed code memory locations is to fill them with a value like 0x02. In this case, the PFEH should be located at ROM location 0x0202. If the processor erroneously branches into the NF region, the processor will execute consecutive 0x02 values (equivalent to the ‘LJMP 0x0202’ instruction) and jump to the PFEH. This variation – identical to that described by Campbell [Campbell 1995] and Banyai & Gerke [Banyai & Gerke 1996] for Motorola microcontrollers – greatly reduces the error detection latency.

The NF region can be created with most chip programmers by performing a memory fill before the program is loaded, or by using assembly directives. Relocation of the PFEH can be done with most linkers, or can be performed with suitable assembler/compiler directives.

4.2 Function Tokens

The idea behind Function Tokens [Banyai & Gerke 1996, Campbell 1995, Campbell 1998, Coulson 1998, Coulson 1999, Li et. al. 1984, Niaussat 1998] is to use a RAM location (known as the ‘token’) to store the unique identifier (ID) assigned to each function. Before a function is called, the token is assigned the function’s ID. Within the function, the value of the token is checked: if the token does not have the required value, it is assumed that execution had been ‘thrown in’ to the function as the result of a program-flow error. The PFEH will then be called to handle this error.

![Figure 4.2: Implementation of Function Tokens](image-url)
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 4.2 illustrates a simple implementation of FT, assuming a one-to-one relationship between calling and called functions. The first line of code in the called function checks that the token is equivalent to its ID. Before exiting the called function, the token value is changed to the ID of the calling function, the calling function then checks to make sure the token matches its ID. If the token’s value differs from a function’s ID when a check (FT check) is carried out, a program-flow error has been detected and the appropriate routine will be executed.

The relationship between calling and called functions for most real-world programs is not as straightforward as that shown in Figure 4.2. For real-world programs, this relationship is usually many-to-many, which complicates the implementation. Under such circumstances, it is necessary to store the ID of the calling function in the called function.

Listing 4.1 shows how FunctionA (from Figure 4.2) can be coded in C for a one-to-one function calling relationship. The token should be a global variable to allow its use in ISRs, and reduce the function’s set up time.

```c
int FunctionA(int A, int B)
{
    char FT_TEMP = FT;
    if(FT != 2) PFEH();
    .
    if(FT != 2) PFEH();
    .
    FT = 3;
    FunctionB();
    if(FT != 2) PFEH();
    .
    FT = FT_TEMP;
    return 2;
}
```

Listing 4.1: Example C routine of Function Token implementation

To implement FT, the relationship between calling and called functions needs to be completely mapped out to ensure that FT checks are checking the correct ID (under normal execution). This is a straightforward but tedious and error prone procedure, especially when FT is implemented as an afterthought.
4.3 Modelling NF and FT

To evaluate NF and FT's effectiveness in detecting program-flow errors, a statistical model – considering LMIT errors – was developed. This model predicts the effectiveness of NF and FT, in terms of the probability of certain conditions occurring due to PC corruption, based on the static instruction-type profile (described in Section 4.3.2) of a program (firmware). A program's static instruction-type profile is determined by STATS, an in-house program specially written to do so (see Appendix J). This modelling process is shown in Figure 4.3.

Although the model presented in this chapter is specifically tailored for the 8051 instruction set, the same methodology used to deriving this model is applicable to other Harvard processor families.

4.3.1 Assumptions

The key assumption is that, in response to PC corruption, the PC has the same probability of taking on any value in the range of 0 to 65535 (the 8051's PC is 16-bits wide).

It is also assumed that no other errors occur before the first instructions at the 'landing' code memory location is executed. That instruction is also assumed to be non-branching.

4.3.2 Static instruction-type classification

The instructions of a program is categorised based on two criteria: its size (in bytes) and if it contains any trailing zero-valued byte. This classification, known as the 'static instruction-type', is described in Table 4.1. The term 'static instruction-type profile' would refer to the sum of each static instruction-type in a program.
4.3.3 Classifying static instruction-types as SL, UL and DL

To quantify the effectiveness of NF and FT, a classification – based on the threat of each memory location – divides the physical code memory into 'Safe', 'Unsafe' and 'Dangerous' code memory locations. The following list defines the characteristics of each of the classes:

- **Safe Locations (SL)** – In the event of PC corruption, vectoring to these code memory locations means detection will be guaranteed to happen and processor state changes (excluding PC and timing related registers) will not occur.

- **Unsafe Locations (UL)** – In the event of PC corruption, vectoring to these code memory locations mean error detection is not guaranteed. The error however, may not be fatal, and program execution may not be affected.

- **Dangerous Locations (DL)** – In the event of PC corruption, vectoring to these code memory locations means an LMIT error will occur; possibly followed by unpredictable program flow.

4.3.4 Classifying 8051 instruction-types as SL, UL and DL

Based on these characteristics, a table denoting the relationship between the instruction-types and SL, UL and DL, is shown in Table 4.2. In the table, non-zero valued bytes are denoted by 'X', while '0' denotes zero-valued ones.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands (Hex)</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Byte 1</td>
<td>Byte 2</td>
</tr>
<tr>
<td>Size</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>D2</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>DN</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>T23</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>T2 (TN)</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>T3</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>TN</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 4.2: Instruction byte classification

The contents of Table 4.2 may be summarised as follows: the first byte of every instruction is considered UL since that instruction will not be misinterpreted when the corrupted PC ‘lands’ just before it. Subsequent non-zero bytes (in multi-read instructions) however, are classified as DL: they will cause LMIT errors.

Even though the second byte of T2-type instructions is zero, these instructions are classified as ‘DL’ since instruction misinterpretation will still occur (the third byte is non-zero valued). Therefore, T2-type instructions are essentially the same as ‘TN’; hence, labelled as such.

Note that no bytes are classified as SL. SL is classified in a different manner, as discussed in the next section.

4.3.5 Mathematical model to calculate SL, UL and DL

Figure 4.4: Typical code memory map of a program implementing NF and FT.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 4.4 shows a typical memory layout of a program implementing NF and FT on a processor with M-sized physical code memory. A description of the main symbols used in this section is shown in Table 4.3. Note that the amount of unprogrammed code memory locations within a program (e.g. at unused interrupt vectors) is usually insignificant, and are considered here to be part of the program code, P.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Total physical code memory size</td>
</tr>
<tr>
<td>P</td>
<td>Original program size (without error detection and correction techniques)</td>
</tr>
<tr>
<td>P*</td>
<td>Total program size (excluding NF region)</td>
</tr>
<tr>
<td>UP</td>
<td>Unprogrammed code memory locations (when NF not implemented)</td>
</tr>
<tr>
<td>T</td>
<td>Code memory locations occupied by FT checking routines</td>
</tr>
<tr>
<td>NF</td>
<td>Code memory locations used as NF region</td>
</tr>
<tr>
<td>C</td>
<td>Code memory locations occupied by Program-Flow Error Handler</td>
</tr>
</tbody>
</table>

Table 4.3: Description of symbols used

Referring to Figure 4.4, P* is calculated as:

\[
P* = P + (\bar{T} \times n) + C
\]

Equation 4.1

where: \( \bar{T} \) = Average size of each FT checking routine (bytes)

\( n \) = Number of FT checks.

The size of the NF region is then given by:

\[
NF = M - P*
\]

Equation 4.2

\[
UP = 0
\]

Equation 4.3

If NF is not implemented however, then:

\[ P* \] is also the size of the program obtained from the statistics generated when it is read into the buffer of a PROM/EPROM programmer.
Techniques intended to reduce the impact of program-flow errors on embedded systems

\[
\begin{align*}
NF & = 0 \\
UP & = M - P^*
\end{align*}
\]

As the entire NF region guarantees error detection and does not change the state of the processor (apart from the PC and timing-related registers), it is classified as SL. This is also the case for the first byte of every FT check and the PFEH. Therefore, SL may be defined as:

\[
SL = NF + n + 1
\]

Equation 4.4

UL is based on the program’s instruction and is calculated as so:

\[
UL = \sum S + \left( \sum \frac{DN}{2} \right) + \sum D2 + \left( \sum \frac{TN}{3} \right) + \\
\sum T23 + \left[ \left( \sum T3 \right) \times \frac{2}{3} \right] - n - 1
\]

Equation 4.5

where: \( \sum X \) = Total code memory location occupied by instruction-type ‘X’ (e.g. DN) of a program

As shown in Table 4.2, not all bytes of double- and triple-byte instructions are considered unsafe. Hence, DN-, TN- and T3-type instructions are multiplied by the ratio of the number of UL byte(s), to the length of the instruction. The number of FT checks and one byte for the PFEH are also subtracted from UL – they have been accounted for in SL. DL, also determined from Table 4.2, is given as:

\begin{table}[h]
\end{table}

---

14 It is assumed that the PFEH is shared between all error detection techniques, and will be able to detect errors that ‘land’ on its first byte.

15 These parameters are program dependent.
Techniques intended to reduce the impact of program-flow errors on embedded systems

\[
DL = (\sum DN/2) + (\sum TN \times 2/3) + (\sum T3/3) + UP
\]

Equation 4.6

Equation 4.6 may in turn be simplified to:

\[
DL = M - SL - UL
\]

Equation 4.7

Equation 4.4, Equation 4.5 and Equation 4.7 can be restated in percentages of the physical code memory size by dividing them by \(M\), producing the following:

\[
SL(\%) = \frac{SL}{M}
\]

Equation 4.8

\[
UL(\%) = \frac{UL}{M}
\]

Equation 4.9

\[
DL(\%) = \frac{DL}{M}
\]

Equation 4.10

Equation 4.8 to Equation 4.10 are valid irrespective of the amount of physical code memory available since memory aliasing is automatically taken into consideration. Memory aliasing occurs when not all the address lines are used to decode the physical code memory locations as the implemented code memory is usually smaller than what the 8051 is capable of.

4.4 Simulation to evaluate NF and FT

4.4.1 Description of test program

One test program, implementing a combination of NF and FT, was written in C and compiled with the Keil C51 compiler. ‘Prog’, and its variants, were simulated on dScope, the bundled 8051 simulator. A C-like simulation script – part of dScope – was used to automatically control and log results of the entire simulation.

Prog is a 41-tap finite-impulse response (FIR) filter that randomly reads code memory locations for the input values. The reading is carried out when Timer 0 overflows at a rate of
Techniques intended to reduce the impact of program-flow errors on embedded systems

2kHz, when the MCU is fed with a 12MHz clock. The 8-bit output of the filter is written to one of the ports and an I²C (inter-interconnect) memory device. The I²C protocol is emulated in software.

As there are four possible combinations of implementing NF and FT, four test programs – suffixed ‘_A’, ‘_B’, ‘_C’ and ‘_D’ – are described as follows:

- **Prog_A** – Original version without NF and FT.
- **Prog_B** – Implements NF.
- **Prog_C** – Implements FT.
- **Prog_D** – Implements NF and FT.

‘Prog’ (without suffix) would refer to all variants. A graphical representation of them (not in proportion) is shown in *Figure 4.5*. A 4kB code memory is assumed for this experiment.

Prog_C and Prog_D implemented 32 FT checks (11 more than the minimum\(^{16}\) necessary) at function entry and exit points, and just before ‘critical’ code segments. When errors are detected, the FT checking routines vector program execution to the PFEH, situated at the end of the physical code memory. This allows both FT and NF to use the same PFEH.

In the test programs, all the PFEH did was to set the error status (a reserved RAM location that is checked by the simulation script), and to put the program into an infinite loop.

\(^{16}\) The number of checks required to only monitor function calls (checks not inserted at other locations).
4.4.2 Simulation procedure

A 'Monte Carlo'-style simulation was carried out to evaluate the model (see Appendix M for script listing). Due to the complexity of the processor, the only viable way to evaluate models associated with it is to change the desired parameters (i.e. error injection) and observe its effects. Benso et al. [Benso et. al. 1999] and Delong et. al. [Delong et. al. 1996] did mention that fault injection is an effective approach for the evaluation of fault coverage properties.

During each simulation cycle, one PC error is introduced, at a random basis, between the first and 10,000,000th instruction cycle. The instruction cycle in which the error occurs (ECY), together with the erroneous PC value (EPC), is predetermined using dScope’s random number generator.

A conditional breakpoint is used to stop program execution when the current instruction cycle is equal or greater than ECY. At that breakpoint, the simulation script changes PC’s value to EPC. If EPC points to an empty code memory location or the NF region, the error is recorded as an ‘E Jump’ or ‘NF’ respectively, and a new simulation cycle commences. If EPC points to a location within the programmed region however, the simulator compares this value with a list of all LMIT locations17. A match would mean an ‘LMIT’ hit is logged and the simulation script starts a new simulation cycle.

If none of the previously mentioned categories apply, the simulator single-steps for up to a further 50,000 clock-cycles. On each step, the script checks a number of important program variables. If an FT checking routine detects an error, an FT hit is logged and a new simulation cycle commences. Upon reaching the 50,000 additional clock-cycle limit – which implies that no errors have been detected – the situation is classified as ‘Continued’ and another simulation cycle begins.

The error simulation method where register bits are flipped18 to emulate corruption has been carried out in [Asenek et. al. 1998, Velazco et. al. 2000]. Both groups concluded that the results produced by this technique correlates well with radiation experiments when the register’s duty cycle is accounted for.

---

17 This list is generated by STATS (see Appendix J).
18 Known in [Velazco et. al. 2000] as 'Code Emulating Upsets'.

46
4.5 Results

4.5.1 Static instruction-type profile

The task of generating Prog's static instruction-type profile was left to STATS, an in-house program (see Appendix J), that derived the required statistics by processing the downloadable (in Intel Hex8 format) code. The results are shown in Table 4.4.

<table>
<thead>
<tr>
<th>Program</th>
<th>Size Code</th>
<th>Single (bytes)</th>
<th>Double (bytes)</th>
<th>Triple (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>NF 0</td>
<td>DN 840</td>
<td>D2 818</td>
</tr>
<tr>
<td>Prog_A</td>
<td>2017</td>
<td>840</td>
<td>818 80</td>
<td></td>
</tr>
<tr>
<td>Prog_B</td>
<td>1931</td>
<td>2165</td>
<td>790 64</td>
<td></td>
</tr>
<tr>
<td>Prog_C</td>
<td>2635</td>
<td>941</td>
<td>1156 130</td>
<td></td>
</tr>
<tr>
<td>Prog_D</td>
<td>2520</td>
<td>1576</td>
<td>1112 110</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Instruction breakdown for Prog

The total program size shown in Table 4.4 excludes the unprogrammed code memory locations. An extra piece of code to aid simulation is added to Prog_A and Prog_C, which explain their larger sizes as compared to Prog_B and Prog_D respectively.

4.5.2 Calculated SL, UL and DL values

Since Prog_A and Prog_C do not employ NF, their SL value is 0 and 33 respectively (Prog_C has 33 due to its FT checks and PFEH). Based on the difference in program sizes, it is calculated that each FT check requires, on average, 19 bytes.

SL, UL and DL values, based on Equation 4.4 to Equation 4.6, are shown in Table 4.5. The results are given in bytes and percentage of physical code memory.

<table>
<thead>
<tr>
<th>Program</th>
<th>SL (bytes)</th>
<th>UL (%)</th>
<th>DL (bytes)</th>
<th>DL (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog_A</td>
<td>0</td>
<td>0.00</td>
<td>1427</td>
<td>34.84</td>
</tr>
<tr>
<td>Prog_B</td>
<td>2166</td>
<td>52.88</td>
<td>1375</td>
<td>33.57</td>
</tr>
<tr>
<td>Prog_C</td>
<td>33</td>
<td>0.81</td>
<td>1766</td>
<td>43.12</td>
</tr>
<tr>
<td>Prog_D</td>
<td>1609</td>
<td>38.28</td>
<td>1704</td>
<td>41.60</td>
</tr>
</tbody>
</table>

Table 4.5: Calculated SL, UL and DL values for Prog
4.5.3 Simulation results
Simulations were carried out on four Pentium 200MHz PCs running Windows NT 4.0. The entire simulation took 14 days. The results are shown in Table 4.6.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Errors</th>
<th>Undetected Error</th>
<th>Total Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FT</td>
<td>NF</td>
<td>E Jump</td>
</tr>
<tr>
<td>Prog_A</td>
<td>0</td>
<td>0</td>
<td>474</td>
</tr>
<tr>
<td>Prog_B</td>
<td>0</td>
<td>528</td>
<td>0</td>
</tr>
<tr>
<td>Prog_C</td>
<td>313</td>
<td>0</td>
<td>346</td>
</tr>
<tr>
<td>Prog_D</td>
<td>263</td>
<td>402</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.6: Results of 1000 error simulation cycles

4.5.4 Classifying results as SL, UL and DL
In order to do a comparison between the simulated and calculated results, the categories shown in Table 4.6 have to be reclassified as SL, UL or DL. Although Table 4.6 deals with the hits recorded in each category, they can be classified as SL, UL and DL since the PC is assumed to have equal probability to hold any addressable value when a corruption occurs. Therefore, when Prog_B recorded 528 NF hits, it can be stated that 52.8% of PC errors landed in Safe Locations (the NF region is considered as are SLs, for reasons discussed earlier).

'E Jump' and LMIT hits are classified as DL since they would cause a system reset in the former, and will cause misinterpretation of instructions in the latter. Although some systems may tolerate an unforced reset, others may not. 'Continued' is classified as UL since the processor still manages to continue correct program execution.

The classification of FT is not straightforward. Unlike NF, the processor will most probably execute some instructions before FT checks detect the error, as in scenario 'C' of Figure 4.6. It is unlikely that FT checks would detect the program-flow errors immediately after they occur, as in scenario 'A', which is the ideal condition. Instead, scenarios 'B', 'C' and 'D' predominate. Scenario 'B' is also particularly distressing: some 'critical' code would have been executed despite the fact that an FT check took place.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Due to the fact that processor state changes are very likely to occur even when program-flow errors are detected by FT checks, FT hits are considered as UL. Table 4.7 shows the reclassified simulation results.

<table>
<thead>
<tr>
<th>Program</th>
<th>SL</th>
<th>UL</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(hit)</td>
<td>(%)</td>
<td>(hit)</td>
</tr>
<tr>
<td>Prog_A</td>
<td>0</td>
<td>0.0</td>
<td>361</td>
</tr>
<tr>
<td>Prog_B</td>
<td>528</td>
<td>52.8</td>
<td>315</td>
</tr>
<tr>
<td>Prog_C</td>
<td>0</td>
<td>0.0</td>
<td>417</td>
</tr>
<tr>
<td>Prog_D</td>
<td>402</td>
<td>40.2</td>
<td>362</td>
</tr>
</tbody>
</table>

Table 4.7: Reclassified simulation results

4.6 Discussion

Ignoring the code overhead of Prog_A and Prog_C (which was discussed earlier), the sizes of Prog_A and Prog_B, excluding the NF region, are roughly the same, as shown in Table 4.4. This comes as no surprise: NF does not increase the program’s size and there will only be a small increase due to the PFEH. Prog_C and Prog_D are about 20% bigger due to the implementation of FT, in line with Coulson’s [Coulson 1998, Coulson 1999] estimates. On average, each FT check requires approximately 19 bytes.

The static instruction-type profile for the various programs also show an increase in percentage in the number of multi-read instructions when FT is implemented (e.g. single- and multi-read instructions increased by 12% and 34% respectively for Prog_A and Prog_C). This means that a larger proportion of the instructions making up FT checks are multi-read
Techniques intended to reduce the impact of program-flow errors on embedded systems

types. As expected, the increase of multi-read instruction also increases the percentages of UL and DL for Prog_B and Prog_D, as shown in Table 4.5.

The calculated values for SL, UL and DL (see Table 4.5) show a large increase in SL when NF is implemented. Since unprogrammed code memory locations for variants not implementing NF are classified as DL, Prog_A and Prog_C have high DL values. Prog_A's DL value is higher than Prog_C's as Prog_C is larger; hence, it has less unprogrammed code memory locations. The situation is opposite for Prog_B and Prog_D however: the FT checks reduced the amount of unprogrammed code memory locations.

The simulation results (see Table 4.6), clearly show the importance of employing at least some form of error detection technique. Without NF or FT, close to 50% of all errors could have reset the processor (the processor would execute past the last physical memory location and wrap around due to memory aliasing). Of the two techniques, NF produce a higher error detection rate than FT since error detection is almost 100% within its coverage; this is not the case for FT (discussed in the next section).

NF's error detection rate is lower in Prog_D as compared with Prog_B. A similar trend is also seen in Prog_C and Prog_A: Prog_C has less 'E jumps' (when NF is not implemented) than Prog_A. This is expected due to following reasons:

- FT's implementation increases the program size, thus decreasing NF's coverage (the unprogrammed code memory locations).
- Every byte in the NF region contributes to the error detection (and an SL). FT is only guaranteed to work if the PC erroneously 'land' before or on its first byte; hence, each FT check only contributes one SL and the rest would be UL or DL.

Table 4.6 also shows that the variants implementing FT recorded more LMIT and less 'Continued' hits. This is attributed to the large and more complex code when FT is implemented.

The overall error detection rate is the highest (nearly 67% of injected errors) when both techniques were implemented. This does not mean that a system is safer, however. As discussed in the previous section, by the time FT has detected errors, it is practically the case that the processor (excluding the PC and timing-related registers) has changed state and could have executed 'critical' code segments (see Figure 4.6).
When this is viewed together with the fact that FT checks contribute only one SL at the expense of many ULs and DLs, it can be concluded that, although FT does increase the error detection rate, it may reduce a system's safety, especially when processor state changes are intolerable. Hence, instead of implementing both techniques, it may be better to implement NF alone. This conclusion is also supported by the calculated results (see Table 4.5).

The comparison between the calculated and simulated results for SL, UL and DL is shown in Table 4.8. This table is a summary of Table 4.5 and Table 4.7. The third column ('Dif.') under SL, UL and DL, shows the difference between the calculated and simulated values.

The main observation from Table 4.8 is the good correlation between the calculated and simulated results for SL. In the case of UL and DL, the difference between simulated and calculated results is slightly larger, due to the classification method employed.

<table>
<thead>
<tr>
<th>Program</th>
<th>SL (% of code)</th>
<th>UL (% of code)</th>
<th>DL (% of code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog_A</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Prog_B</td>
<td>52.88</td>
<td>52.80</td>
<td>0.08</td>
</tr>
<tr>
<td>Prog_C</td>
<td>0.81</td>
<td>0.00</td>
<td>0.81</td>
</tr>
<tr>
<td>Prog_D</td>
<td>39.28</td>
<td>40.20</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Table 4.8: Comparison between calculated and simulated SL, UL and DL

The difference between the calculated and simulated values is largest for Prog_D. This anomaly is likely due to the classification method and the relatively small number of simulation runs.

### 4.7 Evaluation and comparison of NF and FT

The characteristics of NF and FT are very different. In this section, the techniques are compared and evaluated based on the factors discussed in Chapter 1.

#### 4.7.1 Overall cost

Both NF and FT will incur some cost due to implementation time. FT’s implementation however, is tedious and error prone as it can involve substantial program alterations (see Section 4.7.5). Hence, its implementation duration is substantially longer than NF’s and will incur a higher cost overhead. For mass-produced applications however, the cost overhead of
Techniques intended to reduce the impact of program-flow errors on embedded systems

FT may not be substantial, especially if it could reduce component count. NF’s cost overhead will be insignificant even for non-mass-produced systems.

4.7.2 Error detection rate
NF’s error detection rate within its coverage region (the NF region) is practically 100%. The only situation that will prevent the processor from executing the PFEH within the stipulated time is when interrupts occur before the PFEH is reached (this also applies to FT). Under such circumstances, the PFEH will be reached after the ISR terminates. Its overall error detection rate however, is governed by the ratio of the NF region to the size of the physically implemented code memory.

FT’s error detection region complement NF’s, i.e. FT will only detect errors branching program-flow to programmed code memory locations. Within FT’s coverage, there is no direct relationship between FT’s error detection rate and the number of FT checks implemented: the location of FT checks and how programs execute must be considered. In any case, FT’s error detection rate in its coverage is less than 100% due to some of the following factors:

- PC ‘landing’ in unmeetable conditional loops without FT checks – infinite loop.
- Forward or backward branches along sequential code blocks within the same function – function ID does not change.
- Branches to locations that would call the same function that the erroneous branch happened – function ID is the same.
- Branches to locations after FT checks but before token changes – token change overrides erroneous token value.

It is only possible to conclude that NF’s error detection rate is practically 100% within its coverage, and FT’s error detection rate, though less than 100% within its coverage, may increase with the number of FT checks employed.

4.7.3 Error detection latency
NF’s maximum error detection latency is the time taken to transverse the entire NF region, although the alternative implementation reduces it to five clock-cycles\(^9\). The maximum error detection latency for FT would be the longest duration between two successive FT checks.

\(^9\) Three clock-cycles after a MUL or DIV instruction starts plus another two to executed the ‘LJMP’ instruction. Non-branching instructions are assumed.
Its shortest is the time taken to execute a check. Hence, it can be concluded that NF’s (alternative form) latency is very short, and FT’s latency can be up to the duration between the longest FT check.

4.7.4 Processing resources
FT will incur processing time, RAM and code memory. The increase in code memory is proportional to the number of FT checks implemented since they have similar high-level structures (see Listing 4.1). This also means that the execution time overhead is closely related to the number of FT checks taking place within a specific time frame. On the other hand, NF does not require RAM, code memory or processor time: it only uses unprogrammed code memory locations. In short, NF clearly meets the third criterion presented in Chapter 1, which is not the case for FT.

4.7.5 Program alterations
NF’s implementation is trivial and, apart from the PFEH, can be carried out with most chip programmers. On the other hand, to correctly implement FT, the relationship between functions has to be known: this is a tedious and error prone procedure. Hence, implementing NF is significantly easier and requires far less program alterations than FT.

4.7.6 Development tool alterations
Both NF and FT are purely software-based techniques: they may be implemented without any modifications to the development tools.

4.8 Predicting the impact of NF and FT with different program parameters
As the model has been shown to correlate well with the simulated results in the previous section, this allows the model to be used to predict the effectiveness of NF and FT with different program parameters. In this section, the physical code memory size and number of FT checks are varied to gauge the impact on SL, UL and DL values.
Equation 4.5 shows that SL is largely governed by the size of the NF region. Thus, by increasing the NF region in relation to the program size, the percentage of SL will increase. Using Prog_D as an example, this relationship is shown in Figure 4.7. The physical code memory size is varied while other factors remain unchanged.

Figure 4.7 supports the claims by Coulson and Campbell [Campbell 1995, Campbell 1998, Coulson 1998] regarding the effectiveness of NF. Note that the percentage of SL increases dramatically between 4kB and 8kB of implemented physical code memory. This increase quickly tapers off as the implemented code memory is increased further. The opposite effect applies for UL and DL. Hence, there is a trade-off point where increasing the physical code memory size is no longer feasible.

Turning to FT, Figure 4.8, Figure 4.9 and Figure 4.10 show a family of graphs where the number of FT checks were varied for implementations on 4kB, 8kB and 32kB of physical code memory (Prog_D was again used as the example). In the examples, each FT check is assumed to require 20 bytes.
Techniques intended to reduce the impact of program-flow errors on embedded systems

For small physical code memory sizes, SL decreases dramatically when the number of FT checks increases. This is expected: each FT check introduces many ULs and DLs while contributing only one SL. The impact of FT checks on SL is less prominent when larger physical code memory sizes are used as the ratio of FT check to code memory size decreases.
Techniques intended to reduce the impact of program-flow errors on embedded systems

These figures also show what has been concluded in the previous section: the number of Safe Locations decreases as the number of FT checks increases. From another perspective, it is likely that FT may decrease the safety of a system when implemented in conjunction with NF.

4.9 Conclusion

The experiments carried out in this chapter have provided further evidence that NF and FT detect program-flow errors. A model to quantify their effectiveness – based on a program’s instruction-type profile – was shown to correlate well with the experimental results.

NF contributes a higher percentage of SL than FT. Although the error detection rate increases when NF and FT are implemented together, this does not necessarily mean an increase in a system’s safety.

NF and FT’s implementation is cost effective for mass-produced systems and they do not require modifications to development tools. NF’s error detection rate is better than FT’s (in their respective coverage) and it has a lower latency. In addition, NF’s implementation is trivial as compared with FT’s. Hence, in some cases, NF may be more desired than FT.
In Part Two, the PC corruption model developed in Part One is extended to handle all possible outcomes of program-flow errors. The resulting model is evaluated by developing a simulator that can operate at the oscillation (sub-cycle) level.
The mechanisms behind 'Late Multi-read Instruction Trap' and 'Normal' error conditions when PC corruption occurred were discussed in Part One. That discussion however, assumed PC corruption only occurred on the instruction boundary. In reality, PC corruption may also occur before the processor has read all the operands of an instruction. This situation could not be studied and simulated when the previous model was conceived due to the lack of clock-cycle (sub-cycle) level simulators.

In this chapter, the remaining outcomes when PC corruption occurs are discussed and illustrated.

5.1 Early Multi-read Instruction Trap
The 'Early Multi-read Instruction Trap' (EMIT) error condition (or EMIT error) is closely related to LMIT. EMIT errors happen when the PC corruption occurs before all the operands of an instruction are read, hence the prefix 'early'. Thus, the equivalent instruction executed by the processor would contain bytes from the actual instruction sequence, and some from the location pointed to by the corrupted PC. This phenomenon is illustrated in Listing 5.1, which is a reproduction of Listing 3.1.

| 0100 | 759850 | MOV 98H,#50H |
| 0103 | 438920 | ORL 89H,#20H |
| 0106 | 756DAB | MOV 8DH,#ABH |
| 0109 | 43FF  | ORL A,#FFH   |

Listing 5.1: Original instruction sequence

Listing 5.1 shows the original instruction sequence that would be sequentially executed during normal execution. Since three reads are needed to read a triple-byte instruction, if the PC is corrupted with the value of 0x0104 after reading the second, but before the third byte of the first instruction is read (PC corruption after address location 0x0101), the processor would thus execute the instructions equivalent to that shown in Listing 5.2.
Techniques intended to reduce the impact of program-flow errors on embedded systems

In this situation, the original instruction type would be correct, but not the operands. This is because the first byte of multi-read instructions uniquely determines the type of instruction that would be executed, although some multi-read instruction sets (e.g. 68HC08) require more than one byte to uniquely identify instructions. The severity of this form of errors can be significant, ranging from incorrect arithmetic results to ‘random’ call and branch jumps.

EMIT errors will not affect single-read instruction sets of Harvard and von Neumann architecture processor families. Both architectures with multi-read instructions however, will be affected by EMIT errors. This issue is supported by Burnham & Cowling: they mentioned in [Burnham & Cowling 1984] that three-byte instruction devices are more likely to enter into random logic execution than two-byte instruction devices.

5.2 EMIT and LMIT errors

EMIT and LMIT errors are directly related. It is possible that LMIT errors occur as the result of EMIT errors – known as eiLMIT (EMIT-induced LMIT) errors; or EMIT errors masking the effects of LMIT errors. The former can be shown by extending the processor’s execution sequence from Listing 5.2, as shown in Listing 5.3.

The ‘newly’ interpreted instruction distorts the original instruction boundary when the EMIT error occurs, thus causing an eiLMIT error.

The complex relationship when an erroneous PC branch occurs (termed the PC’s ‘take-off’ point) and where it points to (termed the PC’s ‘landing’ point) is illustrated in Figure 5.1. In this figure, ‘A’, ‘X’ and ‘Z’ are single byte instructions, ‘C’ and ‘W’ are double byte instructions and ‘B’ and ‘Y’ are triple byte instructions. The bold vertical lines (e.g. between ‘B’ and ‘C’) denote the original instruction boundary.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 5.1: EMIT, LMIT and eiLMIT scenarios

The first two scenarios are the 'Normal' and LMIT errors. The third scenario shows one or more operands will be from the byte(s) at the PC's 'landing' location, while the subsequent instructions are correctly interpreted. As mentioned earlier, EMIT errors can nullify the effects of LMIT errors, as shown in the forth scenario, or cause eiLMIT errors, as in the fifth scenario. Although both EMIT and LMIT errors can occur exclusively, as in the last scenario, they are regarded as eiLMIT errors.

Collectively, EMIT, eiLMIT and LMIT error conditions are known as Multi-read Instruction Trap (MIT) error conditions (or 'MIT errors').

5.3 Summary of EMIT and LMIT error conditions for multi-read instruction sets

EMIT and LMIT error conditions can be summarised as follows:

- EMIT errors occur before all the bytes of multi-read instructions were read.
- LMIT errors occur when the PC points to allocations between multi-read instructions.
- EMIT errors cause the processor to execute the correct instruction, but with the wrong operands.
- LMIT errors cause the processor to execute 'new' instructions.
- EMIT errors can cause LMIT errors, or nullify the effect of potential LMIT errors.
Techniques intended to reduce the impact of program-flow errors on embedded systems

When taking the different processor architectures and instruction set types into consideration, Table 5.1 lists the susceptibility of each combination.

<table>
<thead>
<tr>
<th>Processor architecture</th>
<th>Instruction sets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>multi-read</td>
</tr>
<tr>
<td>von Neumann</td>
<td>EMIT and LMIT</td>
</tr>
<tr>
<td>Harvard</td>
<td>EMIT and LMIT</td>
</tr>
</tbody>
</table>

Table 5.1: MIT susceptibility of von Neumann and Harvard processor architectures with single- and multi-read instruction sets

5.4 Conclusion

MIT errors can happen when the erroneous branch occurs before all operands of an instruction have been read. This can take place in both von Neumann and Harvard architecture processor families with multi-read instruction sets. The effects of EMIT error conditions are the execution of instructions with wrong operands and the possible skewing of the instruction boundary at the ‘landing’ locations. This could lead to LMIT error conditions (i.e. eiLMIT) or prevent it from happening, depending on the instruction-type at ‘take-off’ and ‘landing’. Both Harvard and von Neumann architecture processors with single-read instruction sets do not suffer from EMIT errors.

\(^{20}\) Harvard architecture processor families with single-read instruction sets are susceptible to LMIT error conditions when data (constants) are stored in the code memory space. In most cases however, the chances of LMIT errors occurring is slim.
Design of an 8051 simulator

To model and simulate the complete MIT phenomenon, it is necessary to either purchase, or develop, an 8051 simulator that can operate at the sub-cycle level. A sub-cycle level simulator differs from commercial simulators\(^1\) in one key aspect: its simulation resolution (granularity) is the oscillator’s clock edge. In other words, simulation can stop on every oscillation cycle, as opposed to the instruction cycle (which generally comprises of multiple oscillation cycles). Another advantage of sub-cycle simulators is they can accurately simulate custom peripherals, and determine if they are synthesizable.

Since no commercially available simulators is capable of sub-cycle level simulation, it was necessary to develop one in-house. 8051Sim – the result of this endeavour – is described in this chapter.

6.1 8051Sim

\[\text{Figure 6.1: Screen-shot of 8051Sim}\]

\(^1\) Such as Keil’s dScope, Raisonnance’s MUL-SIM51, Avocet’s ADC-51, KSC’s SIMU, Tasking’s CrossView Pro and Chiptools’s ChipView X51.
Two versions of 8051Sim were developed according to the specifications of a generic 8051 microcontroller [Atmel 1997, Siemens 1996a]. The graphical version (8051Sim), shown in Figure 6.1, was written in Visual Basic, while the non-graphical version was written in a combination of Visual C++ and Visual Basic.

The main reason for developing the non-graphical version (8051Sim-NG) is to improve simulation speed, an important factor when dealing with ‘Monte Carlo’-style simulations. Apart from the graphical interface, both versions of 8051Sim are identical\(^\text{22}\) and provide the same functionality. A detailed description of the interface between Visual C++ and Visual Basic for 8051Sim-NG is given in Section 6.3.

The initial version of 8051Sim took approximately two months to complete. Another two months were spent correcting errors in 8051Sim, developing the generic peripherals, testing the simulator in conjunction with dScope, creating the scripting interface and developing 8051Sim-NG.

### 6.1.1 8051Sim features

8051Sim is cycle-accurate when it comes to instruction execution, and accurate to the sub-cycle level for instruction reads, Program Counter increments and updates of the generic 8051 peripherals. All timing specifications available from [Atmel 1997] and [Siemens 1996a] are used to infer the sequence of events that take place in a normal 8051 processor.

The main features of 8051Sim are summarised as follows:

- Instruction accurate, sub-cycle level simulation.
- Generic peripherals: 4 ports, 2 timers and one USART (not for 8051Sim-NG) – all accurately simulated at the sub-cycle level.
- Fully controllable with scripts\(^\text{23}\) written in VBScript. Script functions can also be keyed-in from the ‘Script Input’ window.
- Extended interrupt system allowing two additional user-definable interrupt sources.
- Up to 10 simultaneous conditional breakpoints.

---

\(^{22}\) The USART module and ‘Virtual RS-232 Terminal’ is not present in 8051Sim-NG.

\(^{23}\) Certain 8051Sim functions and settings are only accessible via scripts.
Techniques intended to reduce the impact of program-flow errors on embedded systems

- Five program execution modes: ‘MicroStep’ and ‘Step’ for sub-cycle and cycle level execution, ‘Animate’ for fixed-delay continuous execution, ‘Update Run’ and ‘Full Run’\(^{24}\) for continuous execution.
- Read and displays – in 8051 assembly language – Intel Hex8 format files.
- Interactive I/O ports allowing user/scripts to control the port pins.
- Processor core, register and peripheral updates in dedicated windows on every sub-cycle (except ‘Full Run’ mode).
- ‘Virtual RS-232 Terminal’ for bi-directional interaction with the 8051’s USART in Mode 1, 2 and 3 (not for 8051Sim-NG).

Please refer to Appendix E for 8051Sim’s User Guide.

6.1.2 Core extensions

The virtual 8051 MCU is identical to the generic microcontroller, with the exception of an extended interrupt handling mechanism. Two unused bits in the ‘Interrupt Enable’ (IE) and ‘Interrupt Priority’ (IP) SFRs are used to implement two user-definable interrupt (UDI) sources. The interrupt flag and interrupt vector are individually programmable for both UDI sources via scripts (see Listing G.1).

These UDI sources are very handy when it comes to custom peripheral development as they allow the peripherals to interrupt the processor when specific conditions are met. The extension works in the same manner as other 8051-generic interrupt sources: as far as the programmer is concerned, these interrupts may be used in exactly the same way as the generic interrupts.

6.1.3 8051Sim limitations

8051Sim is not a complete simulator in its current form. Omissions from a complete 8051 MCU model are the ‘Idle’ and ‘Power-Down’ modes, and the external memory interface signals via ports P0, P2 and P3.

The main limitation with the current version of 8051Sim is that instruction executions are only guaranteed to be precise, in terms of internal data transfers, at the instruction level. This situation is due to the lack of available information regarding internal data transfers and

\(^{24}\) This is the only mode that does not update the display on every cycle or sub-cycle.
Techniques intended to reduce the impact of program-flow errors on embedded systems

operations. For example, although a double-byte 2-cycle instruction will have its instruction bytes read at the correct sub-cycle (S1P2 and S4P2, see [Atmel 1997, Siemens 1996a]), execution of that instruction will only take place at the last sub-cycle (S6P2) of the last clock-cycle of that instruction, instead of being executed in part throughout the two clock-cycles.

Another limitation is that the PC cannot hold a value larger than 8191, i.e. it cannot be aliased in the same way as the 8051. On the 8051, the PC will hold any value above the implemented physical code memory: it is the address lines above this boundary left unconnected that causes memory aliasing. On the other hand, 8051Sim’s PC will only hold the aliased value that is less than its 8kB limit (8191), even if a larger value is written to it (via the scripts). This limitation is because the data bus was not modelled: data read and writes happen directly between two memory locations.

Non-processor specific omissions – usually found in commercial simulators (e.g. dScope) – are facilities for displaying program code at higher abstraction levels, reading program code in other formats, ‘profiling’ tasks/functions, and a means of displaying program variables as symbols.

6.2 Design of the simulator

Figure 6.2: Block diagram of 8051Sim
The simulator is made up of three main components: the ‘Simulation Engine’, the ‘Simulation Control’ and the ‘Graphical Interface’ module. 8051Sim’s structure is similar to that of the ST10 simulator described in [Gauthier & Jerraya 2000], apart from the technique employed to integrate custom peripherals.

The Simulation Engine module mimics the 8051 MCU core and generic peripherals, and is user- or script-controlled via the Simulation Control module. In most aspects, the Graphical Interface module is loosely tied to the Simulation Engine and Simulation Control modules. The only direct interaction between the Graphical Interface and Simulation Engine is through the port pins and serial communication, as shown in Figure 6.2. This loose integration between the various modules allow easy porting of the Simulation Engine to other programming environments.

6.2.1 Representation of the 8051 microcontroller memory space

The MCU’s memory spaces (code memory, RAM, SFR and XRAM) are represented by byte-sized arrays and are part of the Simulation Engine module. Although SFRs are located between 0x80 and 0xFF in the directly addressable memory, the ‘SFR array’ is implemented from 0x00 to OxFF on 8051Sim (see Figure 6.3). The lower 128 bytes (0x00 to 0x7F) of this array are considered by 8051Sim as the 8051’s directly addressable RAM. This greatly simplifies programming and improves execution speed since the memory spaces are grouped according to the addressing modes.

With the help of memory access functions, the actual 8051 memory space is automatically reconstructed from 8051Sim’s internal memory layout, and correctly presented to the users, scripts and microcontroller programs. Hence, the behaviour of the simulator is not altered.

An additional 128-byte (0x80 to 0xFF) array, ‘Ppin’, is implemented as a shadow memory space to store the values of the port pins and the USART’s Transmit register. Ppin is only used within 8051Sim and is not externally visible. Figure 6.3 shows the difference between the actual 8051 MCU memory map and 8051Sim’s implementation.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Indirectly addressable | Directly addressable | Indirectly addressable | Directly addressable | Shadow SFR
0xFF | RAM | SFR | RAM | SFR | Ppin
0x00 | RAM |
0x7F | |
0x00 | |

8051 MCU memory map 8051Sim memory map

Figure 6.3: Differences between the 8051 MCU and 8051Sim’s memory map

### 6.2.2 Simulation Engine module

The core of this module is the ‘Cycle Unit’ that mimics a 12-stage state machine: equivalent to the depth of a clock-cycle in the 8051 architecture. Multiple cycle instructions traverse the cycle engine two or four times (equivalent to the number of clock-cycles required to execute each instruction). The simulator keeps track of multiple byte reads and cycles for longer instructions, preventing other instructions from being read before the execution of the current instruction finishes.

<table>
<thead>
<tr>
<th>Sub-cycle</th>
<th>Interrupt</th>
<th>Execution</th>
<th>Peripheral</th>
<th>Breakpoint</th>
<th>GUI update</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1P1</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1P2</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2P2</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3P1</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3P2</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4P1</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4P2</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5P1</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5P2</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6P1</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6P2</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.1: Sub-module execution priority

25 Will not execute during ISR vectoring cycles
During the relevant sub-cycles, the Simulation Engine executes functions in the ‘Interrupt
Control’, ‘Instruction Execution’, ‘Peripheral Update’, ‘Breakpoint Control’ and ‘GUI’26
Update’ sub-modules, as illustrated in Figure 6.2. Note that if dScope (or any other
commercially available simulators) had been used, the sub-cycle level execution shown in
Figure 6.2 cannot be obtained.

Interrupt Control takes priority over all other sub-modules, and will prevent the Instruction
Execution sub-module from running during ISR vectoring cycles. On other cycles, the
Instruction Execution sub-module executes at S1P2 (fetch), S4P2 (fetch) and S6P2 (execute).
The Peripheral Update sub-module is executed whenever any peripheral needs to be
updated27. At S6P2, the Breakpoint Control module is executed to determine if it is necessary
to halt the processor. The lowest priority sub-module is the GUI Update, which is executed
on every sub-cycle. Table 6.1 shows the sub-module update sequence for one clock-cycle.

Note: when custom peripherals are simulated with scripts, it is possible to update the
peripheral before updating the virtual 8051 MCU at each sub-cycle. Care is needed to
prevent peripheral SFRs from being accessed by the processor and peripheral during the same
sub-cycle (i.e. bus contention [Horowitz & Hill 1989]).

6.2.3 **Graphical Interface module**

Apart from the ports and the Virtual RS232 Terminal, this module reads the relevant RAM,
ROM, SFRs and simulation parameters to update visible windows and the ‘Status Bar’. The
updating is carried out only when the internal subroutine `UpdateWindows()` or
`UpdateStatusBar()` is called28. This arrangement allows flexible control over the
comparatively slow process of updating the display. For example, status updates are usually
only required when processor execution stops.

The port pins react to user inputs in the same manner as the ports on the 8051 MCU. The
Graphical Interface module therefore responds in the following manner to user inputs:

- The port pin will change state when toggled if the corresponding bit in the port latch
  is at logic ‘1’.

---

26 Graphical user interface
27 Only 8051 generic peripherals are updated by this sub-module.
28 These subroutines were exposed as scripting functions. See Appendix F for more information.
Techniques intended to reduce the impact of program-flow errors on embedded systems

- The port pin will always be held at logic ‘0’ if the corresponding bit in the port latch is at logic ‘0’.
- The corresponding port pin will, on the port update cycle (SIPI), take on the logic value of ‘0’ – even if it had previously been at ‘1’, when the processor writes a logic ‘0’ to a port latch bit.

Since port pins and port latches are separate entities, the Ppin memory space (described in Section 6.2.2) is used to store the port pin values.

The Virtual RS232 Terminal is updated on a sub-cycle basis like other generic 8051 peripherals. However, this add-on only responds to the changes on Port 3 (P3), pins 0 and 1 (the transmit and receive lines). Serial data transmitted by the 8051’s USART is only accepted correctly by this virtual device when it is set to the same operating mode as the USART. This requirement also applies to transmitting serial data from the Virtual RS-232 Terminal to the USART.

To simplify programming, this add-on is assumed to be running at the same baud rate as the MCU’s USART.

6.2.4 Simulation Control module
The Simulation Control module is responsible for controlling the execution of the virtual 8051 MCU by executing specific functions in the Simulation Engine module. Simulator commands are accessible from the pull-down menus and via scripting. Frequently used commands (e.g. ‘Step’ and ‘Full Run’) are also accessible from the ‘Toolbox’, and they have associated hot-keys.

The MCU program loading routine, scripting interface (discussed in Section 6.4) and simulator settings (e.g. oscillation frequency and ‘Animate’ delay) are also part of this module.

6.3 Increasing execution speeds
‘Monte-Carlo’-style simulations require numerous repetitions of each simulation cycle. Due to this, a fast simulator is essential for simulations to be carried out within a feasible time period (see Section 6.5 for simulation speed benchmarks).
As 8051Sim was written in Visual Basic, the greatest speed improvement was to port the Simulation Engine module and parts of the Simulation Control module to C/C++, as a dynamic-linked library (DLL). Although Visual Basic creates executables, its higher level of abstraction and run-time error checking means it generates less efficient code, and runs slower than Visual C++-compiled programs.

Visual Basic was used to develop the dialog box, and to host the scripting interface. With this, 8051Sim-NG was created, as shown in Figure 6.4.

The core of 8051Sim-NG was written as a DLL so that it can easily be encapsulated with graphical interfaces coded in various programming languages. In order to control the core, important functions were identified and exposed as DLL functions\(^{29}\). These functions were then used by wrapper programs to control every aspect of the core.

The wrapper of 8051Sim-NG is written in Visual Basic since this is the language most familiar to the author. Another reason for using Visual Basic is it is easier to integrate the ActiveX Microsoft Scripting Control (MSC – discussed in Section 6.4).

\(^{29}\) ‘DLL functions’ are not to be confused with ‘scripting functions’. The former is used to communicate with wrapper programs; the latter are functions are could be used in scripts.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 6.5: Block diagram of 8051Sim-NG

The schematic representation of 8051Sim-NG is shown in Figure 6.5. The MSC is located in the Scripting Interface module, which is responsible for interpreting the VBScripts and executing the relevant DLL function. Since 8051Sim-NG does not have a graphical interface, it is only capable of working via scripts.

The interaction between the DLL functions and scripting functions described in Appendix F.

6.4 Scripting for automation and peripheral simulation

Figure 6.6: The interactions of the program code and script with 8051Sim
Figure 6.6 schematically shows the interaction between the scripts and 8051Sim. Although Figure 6.6 is more complicated than Figure 6.5, they are almost identical when the Graphical Interface module is ignored. From the script’s perspective, both versions of 8051Sim are identical.

6.4.1 Exposing 8051Sim’s functions and variables
As with 8051Sim-NG, 8051Sim’s Scripting Control sub-module is responsible for encapsulating the simulator’s scripting functions (SF) and scripting variables (SV) that are made externally visible with the Microsoft’s Scripting Control.

The MSC is a wrapper for VBScript and JavaScript scripting engines (only VBScript enabled) in the form of an ActiveX control. This control allows specifically defined functions, subroutines and variables to be externally ‘viewable’. The MSC also forms the host and parser for the scripts.

The relevant functions/subroutines that controls 8051Sim are exposed by the MSC. Additional functions/subroutines to aid automation, written as part of the Scripting interface, are exposed in the same manner. The MSC also enables direct read/write access to named SFR and simulator variable (for example, \texttt{PO = 0xFF, CYCLES = 50}) instead of doing it through scripting functions.

6.4.2 Script execution
Scripts are parsed and sequentially executed by the MSC. Syntax and run-time errors encountered by the MSC halts the parsing/execution process and the relevant error message is displayed. As for predefined simulator functions and variables, these have to be prefixed with ‘\texttt{SF}.’ and ‘\texttt{SV}.’ respectively, when being referenced (see Listing G.1 as an example).

There are two other important points regarding 8051Sim’s script execution. Script execution will only start from the subroutine ‘\texttt{Main()}’, which must be present in all scripts. In addition, the serial nature of program execution means hardware parallelism has to be explicitly catered for, which differs from other simulation approaches (e.g. when using

\footnotesize
\begin{itemize}
\item SF and SV are classes of functions and variables respectively, that can be read, written and executed by scripts.
\item This is not to be confused with the simulator’s ‘Scripting Control’ sub-module.
\end{itemize}
Techniques intended to reduce the impact of program-flow errors on embedded systems

VHDL – VHSIC\(^{32}\) Hardware Description Language). However, coding for explicit parallelism is seldom needed as access to the same memory location by the processor and peripherals should never occur on the same sub-cycle, unless the SFRs are dual accessed (e.g. Motorola’s 68HC11 MCUs [Melear 1995]).

A summary of points regarding script execution is as follows:

- Scripts must be written in VBScript.
- Scripts written for 8051Sim will work in the same manner when executed under 8051Sim-NG (except when explicitly stated).
- All scripting functions and variables must be prefixed ‘\(SF\).’ and ‘\(SV\).’ respectively. Functions, subroutines and variables defined in the scripts must not be prefixed.
- All scripts must contain the subroutine ‘\(Main()\).’ Script execution starts from this subroutine.
- Scripts are sequentially executed; i.e. parallelism has to be explicitly catered for.

### 6.4.3 Framework for creating and simulating virtual peripherals

As the script functions are able to control every aspect of the virtual 8051 MCU, it is even possible to simulate custom peripherals below the sub-cycle level. This added bonus becomes a necessity when peripherals are clocked at multiples of the processor core frequency. The commented framework (in pseudo code) for simulating such custom peripherals is shown in Listing 6.1.

```
'Initialise global variables, 8051 MCU, custom peripheral
'Load HEX file

For CY = 0 To MAX_CYCLES 'cycles to simulate
    For SCY = 1 To 12 '12 sub-cycles
        For MCK = 1 To 4 'emulating 4x clock
            If MCK = 1 Then
                - microstep 8051 simulator
            End If
            - execute virtual peripheral
            - write execution details to log file
        Next MCK
    Next SCY
Next CY

Listing 6.1: Framework for simulating clock-multiplied custom peripherals
```

\(^{32}\) Very High Speed Integrated Circuit.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The framework shown in Listing 6.1, comprised of three ‘For...Next’ loops that emulate the clock (CY), sub-clock (SCY) and clock-multiplied (MCK) cycles. Clock-multiplication of the custom peripheral is therefore carried out by sub-cycle executing (‘Micro-stepping’) the virtual 8051 MCU only once every four MCK cycles. Execution log files can also be created to record simulation states and variables for later analysis.

To aid clarity, testing and debugging, the custom peripherals should be written as separate functions. Only two functions – one for resetting the peripheral, and another for its execution – should be called from the main routine. By coding the custom peripheral as a state machine, it is easy to execute the required action on each clock-cycle. An example framework (in pseudo code), linked to Listing 6.1, is shown in Listing 6.2.

```pseudocode
Function UpdateCustomPeripheral()
    Select Case MCK
        Case 1: 'perform action on 1st sub-sub-cycle
        Case 2: 'perform action on 2nd sub-sub-cycle
        Case 3: 'perform action on 3rd sub-sub-cycle
        Case 4: 'perform action on 4th sub-sub-cycle
    End Select
End Function
```

Listing 6.2: Example framework of a state machine-based custom peripheral

6.5 Benchmarking

Two sets of simulations to benchmark the execution speed of 8051Sim were carried out. The first benchmark involved comparing the maximum simulation speed between 8051Sim, 8051Sim-NG and dScope. The second benchmark was used to gauge the performance degradation when executing custom peripherals.

6.5.1 Simulation speed comparison between 8051Sim, 8051Sim-NG and dScope

Three test programs – ‘Alarm’, ‘Krider’ and ‘Prog’ (described in Chapter 7) – were simulated for 10,000,000 clock-cycles on each simulator. Before the simulation starts, a breakpoint was set to automatically stop execution at the required number of clock-cycles. The average duration for three runs, accurate to one second, is listed in Table 6.2. The relative speed difference (8051Sim taken as 1.00) is listed under the ‘Dif.’ column. All three simulations were carried out on the same Pentium II 350 MHz machine with 128 MB RAM running on Windows 2000.
Techniques intended to reduce the impact of program-flow errors on embedded systems

![Image](image.png)

**Table 6.2: Execution speed comparison for 10,000,000 instruction cycles**

Due to the slow simulation speed of 8051 Sim, only 100,000 clock-cycles were simulated and the duration extrapolated to 10,000,000 cycles. Window refresh for 8051 Sim and dScope was disabled for an even comparison with 8051 Sim-NG.

The main observation from the execution speed is the fact that 8051 Sim-NG outperformed its graphical sibling by up to 90 times. This unexpected huge difference is attributed to two main factors: the much faster and tighter executable produced by Visual C++, and the absence of any window refresh routines in the code. Based on the test programs, 8051 Sim-NG is also significantly faster than the commercially available dScope.

When the simulation speed of dScope was tested without any breakpoints, approximately 300,000 C/s (300,000 instruction cycles per second), 667,000 C/s and 500,000 C/s were achieved by Alarm, Krider and Prog respectively. This meant breakpoints greatly slowed dScope’s simulation. On the other hand, the same experiment carried out on 8051 Sim yield practically identical results to that in Table 6.2; hence, breakpoints have little impact on it. As 8051 Sim-NG is only script-controllable, this experiment cannot be performed.

Both versions of 8051 Sim recorded similar execution times for different programs. This is to be expected: the cyclic nature of their instruction decoding engine means that the execution time for simple and complex instructions is about the same. The situation is different for dScope due to its instruction-level nature.

---

33 dScope’s 16-bit nature will account, to some degree, for its slower execution.
34 Both simulators were allowed to run at maximum speed for 10 seconds and the number of executed cycles were recorded.

74
6.5.2 Performance degradation due to the simulation of custom peripherals

Two variants of a custom peripheral (script and program is in Appendix G) were developed to benchmark the overhead involved when custom peripherals were simulated. The original version, ‘ORIG’, is based on a flashing LED (light emitting diode) program that is controlled by a custom 16-bit up/down timer.

The first variant, ‘NOUPD’, does not run the custom timer. This version looks at the processing overhead involved in updating the custom peripheral, but not running the timer. The second variant, ‘NORUN’, uses the same program code as ORIG, but executes 8051Sim in ‘FullRun’ mode\(^{35}\). This provides the maximum possible simulation speed, which also means that the custom peripheral is disregarded.

All three programs were executed on 8051Sim for 10000 clock-cycles on the same machine that ran the previous benchmark. Their execution times are listed in Table 6.3.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time (s)</th>
<th>Cycles per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIG</td>
<td>451</td>
<td>22.17</td>
</tr>
<tr>
<td>NOUPD</td>
<td>421</td>
<td>23.75</td>
</tr>
<tr>
<td>NORUN</td>
<td>14</td>
<td>714.29</td>
</tr>
</tbody>
</table>

Table 6.3: Comparison of simulation times

Implementing the custom peripheral slows the simulation process considerably, since many ‘expensive’ (in terms of processing time) calls between the script and the simulator are necessary. By not enabling the custom peripheral (T16_RUN bit not set, see Appendix G), a large portion of custom peripheral’s code was bypassed. However, it still took over 7 minutes to complete 10000 cycles as ‘MicroStepping’ was carried out by a ‘For...Next’ loop.

The largest speed improvement (by a factor of 30 over the original implementation) came when the custom peripheral was disregarded (i.e. not tied to the processor). This is to be expected, as no calls were needed between the script and the simulator.

---

\(^{35}\) Custom peripherals cannot be simulated at ‘Full Run’, ‘Update Run’ or ‘Animate Run’ simulation modes as the script does not have sub-cycle access to the simulator under these modes.
6.6 Conclusion

8051Sim’s ability to simulate an 8051 MCU at the sub-cycle level is instrumental in accurately mimicking hardware-based custom peripheral in software. With its automation capability and the ability to respond to custom events, 8051Sim is ideal for autonomous simulations. It appears that 8051Sim-NG is also faster than dScope.

With 8051Sim, further investigation of the MIT error phenomenon – particularly EMIT and eiLMIT errors – is possible. It is also possible to simulate the outcome of program-flow errors, which can then be compared with the model to gauge the model’s accuracy.

8051Sim’s ability to integrate custom peripherals also plays a key role in the aims of this thesis: to develop chip- and peripheral-based program-flow error detection and correction techniques.
In Chapter 5, EMIT error conditions were described: this completed the picture of PC corruption on multi-read instruction sets. With the development of 8051Sim, described in the previous chapter, it was possible to carry out detailed investigations and simulation of MIT error conditions.

In this chapter, a model is presented to calculate the probability of each outcome when PC corruption occurs based on a program's instruction-type profile. This model is then evaluated with 8051Sim-NG to determine its accuracy.

### 7.1 Modelling MIT errors

Both static (based on the program code) and dynamic (based on executing the program for a predetermined number of cycles) instruction-type profiles are used to statistically determine the outcome when a PC corruption occurs.

#### 7.1.1 Assumptions

One key assumption made when modelling MIT errors is that, in response to PC corruption, the PC has the same probability of taking on any value in the range of 0 to 65535 (the 8051’s PC is 16-bits wide).

It is also assumed that no other errors occur before the first two instructions at the ‘landing’ are executed. The first two instructions at the ‘landing’ are also assumed non-branching instructions.

#### 7.1.2 Definition and description of symbols and schematic representations

The schematic representations shown in the following tables (Table 7.2, Table 7.3 and Table 7.4) are detailed in Figure 7.1.
Techniques intended to reduce the impact of program-flow errors on embedded systems

![Diagram of program branch and landing point](Image)

**Figure 7.1:** Description of schematic diagram used in Table 7.2, Table 7.3 and Table 7.4

Only the last instruction before program branches, and the first (or first two) instruction where the branch 'lands', are shown in the schematic. Instruction branch and 'land' positions are denoted numerically (0 to 2), starting from the left boundary of the first byte of an instruction. The colour-inverted bytes are those that will be interpreted as one instruction.

In the tables, $L_{POS}$ denotes the 'landing' position of the branch while $L_{1S}$ and $L_{2S}$ are the instruction sizes (in bytes) of the 'landing' and the next instruction respectively. An 'X' for $L_{2S}$ means that instruction does not play a part in determining the outcome.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Single-byte instruction</td>
</tr>
<tr>
<td>D</td>
<td>Double-byte instruction (all types)</td>
</tr>
<tr>
<td>DN</td>
<td>Double-byte instruction with non zero-valued bytes</td>
</tr>
<tr>
<td>D2</td>
<td>Double-byte instruction with 2(^{nd}) byte zero-valued</td>
</tr>
<tr>
<td>T</td>
<td>Triple-byte instruction (all types)</td>
</tr>
<tr>
<td>TN</td>
<td>Triple-byte instruction with non zero-valued bytes</td>
</tr>
<tr>
<td>T23</td>
<td>Triple-byte instruction with 2(^{nd}) and 3rd byte zero-valued</td>
</tr>
<tr>
<td>T3</td>
<td>Triple-byte instruction with 3rd byte zero-valued</td>
</tr>
<tr>
<td>UP</td>
<td>Unprogrammed byte</td>
</tr>
<tr>
<td>SUP</td>
<td>Single and Unprogrammed byte ($S + UP$)</td>
</tr>
<tr>
<td>I11</td>
<td>Single-byte one-cycle instruction</td>
</tr>
<tr>
<td>I12</td>
<td>Single-byte two-cycle instruction</td>
</tr>
<tr>
<td>I14</td>
<td>Single-byte four-cycle instruction</td>
</tr>
<tr>
<td>I21</td>
<td>Double-byte one-cycle instruction</td>
</tr>
<tr>
<td>I22</td>
<td>Double-byte two-cycle instruction</td>
</tr>
<tr>
<td>I32</td>
<td>Triple-byte two-cycle instruction</td>
</tr>
</tbody>
</table>

**Table 7.1:** Static and dynamic instruction-type symbols
Techniques intended to reduce the impact of program-flow errors on embedded systems

Table 7.1 lists all the static and dynamic instruction-types that are used to categorise program instructions. Most of the static instruction-type symbols were introduced in Table 4.1; they are shown here for completeness. These symbols are used throughout in this chapter.

### 7.1.3 EMIT error scenarios

EMIT errors occur under three situations: when the program branch happens between the first and second byte of a double-byte instruction (type \( D_{EMIT} \)), between the first and second byte of a triple-byte instruction (type \( T_{EMIT1} \)) and between the second and third byte of a triple-byte instruction (type \( T_{EMIT2} \)). Each situation has its own set of conditions for EMIT/\( eilMIT \) errors to occur; the following tables are used to show all possible combinations.

<table>
<thead>
<tr>
<th>Situation</th>
<th>( L_{POS} )</th>
<th>( L_{1S} )</th>
<th>( L_{2S} )</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>EMIT</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>X</td>
<td>( eilMIT ) exc. ( D2 )</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>X</td>
<td>EMIT</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
<td>X</td>
<td>( eilMIT ) exc. ( T23 )</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>X</td>
<td>( eilMIT ) exc. ( T23, T3 )</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>X</td>
<td>EMIT</td>
</tr>
</tbody>
</table>

Table 7.2: Outcome of 'take-offs' at position 1 for double-byte instructions (\( D_{EMIT} \))
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Situation</th>
<th>L_{POS}</th>
<th>L_{1S}</th>
<th>L_{2S}</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A B C</td>
<td>P X</td>
<td></td>
<td>EMIT</td>
</tr>
<tr>
<td>2</td>
<td>A B C</td>
<td>P X Y</td>
<td></td>
<td>eiLMIT exc. D2</td>
</tr>
<tr>
<td>3</td>
<td>A B C</td>
<td>P X Y Z</td>
<td></td>
<td>eiLMIT exc. T23</td>
</tr>
<tr>
<td>4</td>
<td>A B C</td>
<td>P Q</td>
<td></td>
<td>EMIT</td>
</tr>
<tr>
<td>5</td>
<td>A B C</td>
<td>P Q X</td>
<td></td>
<td>EMIT</td>
</tr>
<tr>
<td>6</td>
<td>A B C</td>
<td>P Q X Y</td>
<td></td>
<td>eiLMIT exc. D2</td>
</tr>
<tr>
<td>7</td>
<td>A B C</td>
<td>P Q X Y Z</td>
<td></td>
<td>eiLMIT exc. T23</td>
</tr>
<tr>
<td>8</td>
<td>A B C</td>
<td>P Q R</td>
<td></td>
<td>eiLMIT exc. T23, T3</td>
</tr>
<tr>
<td>9</td>
<td>A B C</td>
<td>P Q R</td>
<td></td>
<td>EMIT</td>
</tr>
<tr>
<td>10</td>
<td>A B C</td>
<td>P Q R X</td>
<td></td>
<td>EMIT</td>
</tr>
<tr>
<td>11</td>
<td>A B C</td>
<td>P Q R X Y</td>
<td></td>
<td>eiLMIT exc. D2</td>
</tr>
<tr>
<td>12</td>
<td>A B C</td>
<td>P Q R X Y Z</td>
<td></td>
<td>eiLMIT exc. T23</td>
</tr>
</tbody>
</table>

Table 7.3: Outcome of 'take-offs' at position 1 for triple-byte instructions ($T_{EMIT}$)
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Situation</th>
<th>L_POS</th>
<th>L_1s</th>
<th>L_2s</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>EMIT</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>eiLMIT exc. D2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>EMIT</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>eiLMIT exc. T23</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>eiLMIT exc. T23, T3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>EMIT</td>
</tr>
</tbody>
</table>

Table 7.4: Outcome of ‘take-offs’ at position 2 for triple-byte instructions ($T_{EMIT2}$)

Two important points can be made from the scenarios presented in the tables. In the case of $D_{EMIT}$ and $T_{EMIT2}$, only the ‘landing’ byte is necessary to determine if an eiLMIT error has occurred. The possibility of eiLMIT errors occurring for some $T_{EMIT1}$ scenarios however, can only be ascertained by knowing the instruction-type of the instruction following the ‘landing’ instruction. This situation arises when the ‘landing’ instruction will only provide one of the two operands needed.

Like the previous model (see Chapter 4), the fact that not all eiLMIT conditions will produce errors is taken into consideration. If the remaining bytes of the ‘landing’ instruction that do not form part of the erroneously decoded instruction (e.g. byte ‘Q’, condition 2, Table 7.4) have the value of 0x00, an eiLMIT error does not occur: this is noted in bold in the ‘outcome’ column.

7.1.4 LMIT error scenarios

LMIT errors are simpler to comprehend, as only the ‘landing’ location needs to be taken into account. This is due to the fact that all the bytes of the current instruction have been read when a PC error occurs. Table 7.5 is the schematic representation for all possible outcomes.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Table 7.5: Outcome for all LMIT conditions

<table>
<thead>
<tr>
<th>Situation</th>
<th>L_{POS}</th>
<th>L_{IS}</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>Normal</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>LMIT exc. D2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>Normal</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>LMIT exc. T23</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>LMIT exc. T23, T3</td>
</tr>
</tbody>
</table>

As with EMIT, not all LMIT conditions will produce errors. Zero-valued bytes at the 'landing' locations would prevent LMIT errors from occurring, hence the exclusions written in bold. Note that single byte instructions will never cause LMIT errors.

7.2 Mathematical representation of the MIT error model
To simplify the statistical modelling process, the model is divided into three sections, each described separately. Figure 7.2 is the schematic representation of the model. The equations in Section 7.2.3, Section 7.2.4, Section 7.2.6 and Section 7.2.7 are used to form the complete model discussed in Section 7.2.8.

7.2.1 Dissecting the model
The first section of the model, as shown in Figure 7.2, determines the probability of an EMIT error occurring, denoted by \( \alpha \). As the only other possible situation is that an EMIT error does not occur, its probability is simply \( 1-\alpha \). The other two sections split the EMIT/no EMIT probability into four categories. The probability of each split is denoted by \( \beta \) (Normal\(^{36}\) or

---

\(^{36}\) 'Normal' denotes erroneous program-flow branches will occur but will not cause EMIT, LMIT or eiLMIT errors. This will not directly cause program-flow errors, but may have serious repercussions.
Techniques intended to reduce the impact of program-flow errors on embedded systems

LMIT) and λ (EMIT or eiLMIT). As with 1-α, 1-β and 1-λ are the respective conjugates. The probability of each instruction-type is denoted in the form of P(X).

Figure 7.2: MIT error model

7.2.2 Instruction-type probability
Static instruction-type probabilities (S, DN, D2, D, T, TN, T23, T3 and UP) – measured from the program code itself – are given by the ratio:

\[ P(X) = \frac{\text{Number of bytes of instruction-type } X}{\text{Total physical code memory size (bytes)}} \]

Equation 7.1

Dynamic instruction-type probabilities (111, 112, 114, 121, 122 and 132) – determined via simulation – are given by the ratio:

\[ P(X) = \frac{\text{Number of clock-cycles executing instruction-type } X}{\text{Total number of clock-cycles simulated}} \]

Equation 7.2

Calculating the probability of β and λ only requires static instruction-type probabilities while dynamic instruction-type probabilities are needed to calculate α.

7.2.3 Probability of EMIT (α)
EMIT probability calculations depend on the dynamic instruction-type probability of multi-read instructions. As discussed in Chapter 2, the first byte of all instructions is read by the
processor during S1P2 of its first clock-cycle. Double-byte one-cycle (I21) instructions are read during S1P2 and S4P2 while triple-byte (I32) instructions during S1P2, S4P2 and S1P2 of the next clock-cycle [Atmel 1997, Siemens 1996a]. It is assumed that for double-byte two-cycle (I22) instructions (e.g. ORL), both bytes are read during S1P2 and S4P2 of the first cycle.

As a result, EMIT errors will happen between states S1P2 and S4P1 for a double-byte instruction during its first clock-cycle, and between S1P2 of the first and following clock-cycle for triple-byte instructions. In terms of the proportion of time taken to execute each instruction, I21, I22 and I32 instructions will cause EMIT errors in 6 out of 12, 6 out of 24 and 12 out of 24 sub-cycles, respectively. Hence, \( \alpha \) is determined as follows:

\[
\alpha = \frac{P(I21)}{2} + \frac{P(I22)}{4} + \frac{P(I32)}{2}
\]

Equation 7.3

The possibility of EMIT error not occurring will then be:

\[
P(\text{No EMIT}) = P(I11) + P(I12) + P(I14) + \frac{P(I21)}{2} + [\frac{3}{4} \times P(I22)] + \frac{P(I32)}{2}
\]

Since only two possible conditions exist, this equation can be simplified as follows:

\[
P(\text{No EMIT}) = 1 - \alpha
\]

Equation 7.4

which is also true in the case of \( \beta \) and \( \lambda \).

7.2.4 Probability of LMIT (\( \beta \))

The probability of LMIT occurring – described in a similar manner in Chapter 4 – is based on the probability that an instruction would ‘land’ on the first, second or third byte of an instruction. For double-byte instructions, the probability for LMIT errors occurring would be half the probability of ‘landing’ on a double-byte instruction. This is not the case however if the second byte is zero-valued, hence its probability would be \( P(DN)/2 \). Likewise, the same principle applies to triple-byte instructions.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Based on Table 7.5, the probability of LMIT errors occurring is given by:

\[
\beta = \frac{P(DN)}{2} + \frac{2P(TN)}{3} + \frac{P(T3)}{3}
\]

Equation 7.5

\[
P(\text{No LMIT}) = P(\text{SUP}) + \frac{P(DN)}{2} + P(D2) + \frac{P(TN)}{3} + \left[\frac{2}{3} \times P(T3)\right] + P(T23)
\]

\[
= 1 - \beta
\]

Equation 7.6

where \(P(\text{SUP}) = P(S) + P(\text{UP})\)

7.2.5 Sub-dividing the calculations for \(\lambda\)
Calculating the probability of eILMIT errors occurring is more complicated due to the various possible combinations. To simplify discussion and modelling, the three possible eILMIT scenarios, \(D_{EMIT}\), \(T_{EMIT_1}\) and \(T_{EMIT_2}\), are considered separately. Only two unique equations, \(\lambda_1\) and \(\lambda_2\), are derived since \(D_{EMIT}\) and \(T_{EMIT_2}\) conditions are the same. The ‘take off’ instruction is not considered here: it has already been included while calculating \(\alpha\).

7.2.6 Calculating eILMIT for \(D_{EMIT}\) and \(T_{EMIT_2}\) conditions (\(\lambda_1\))
For double-byte instructions, there will be a 50% chance of the PC ‘landing’ at position 0, which would cause an eILMIT error. However, \(D_2\)-type instructions will not cause eILMIT errors, hence only \(DN\)-type will\(^{37}\).

Triple-byte instructions will cause eILMIT errors when the PC ‘lands’ at position 0 or 1 (67% likelihood). Their eILMIT conditions are calculated in the same manner as that for double byte instructions. By summing the terms of each condition, the overall eILMIT probability for \(D_{EMIT}\) and \(T_{EMIT_2}\) conditions is as follows:

\(^{37}\) Note that \(P(D)\) would be equal to \(P(DN) + P(D2)\).
Techniques intended to reduce the impact of program-flow errors on embedded systems

\[ \lambda_1 = \frac{P(DN)}{2} + \frac{P(TN)}{3} + \frac{P(T)}{3} \]

Equation 7.7

\[ P(\text{No eiLMIT}) = P(SUP) + \frac{P(D2)}{2} + \frac{P(D)}{2} + \frac{P(T23)}{3} + \frac{P(T23) + P(T3)}{3} + \frac{P(T)}{3} \]

\[ = 1 - \lambda_1 \]

Equation 7.8

\(\lambda_1\) has the same equation as \(\beta\) since eiLMIT and LMIT errors have the same probability of occurring for both double- and triple-byte instructions, and are both bound by the same exclusions.

7.2.7 Calculating eiLMIT for TEMIT conditions (\(\lambda_2\))

TEMIT eiLMIT error probability calculations are along the same line as \(\lambda_1\), but are more complicated due to the extra conditions. Under some scenarios, an additional instruction following that of the ‘landing’ instruction is needed to determine if an eiLMIT error has occurred. This is because two bytes at the ‘landing’ location are needed to complete the operand, which affects all single-byte instructions, ‘landings’ at position 1 of double-byte instructions, and at position 2 of triple-byte instructions.

When a second instruction is needed to fulfil the eiLMIT criteria, the probability of the next instruction type occurring is multiplied by the probability of the first ‘landing’ instruction. Exclusion conditions due to zero-valued trailing bytes, are taken into account in the same manner as that for TEMIT1 and TEMIT2. Each eiLMIT condition is summed to produce \(\lambda_2\) as follows:

\[ \lambda_2 = [P(SUP) \times P(DN)] + [P(SUP) \times \{P(TN) + P(T3)\}] + [P(D)/2 \times P(DN)] + [P(D)/2 \times \{P(TN) + P(T3)\}] + P(TN)/3 + P(T)/3 \times P(DN) + [P(T)/3 \times \{P(TN) + P(T3)\}] \]
Techniques intended to reduce the impact of program-flow errors on embedded systems

\[
P(SUP) \times [P(DN) + P(TN) + P(T3)] + \\
P(D)/2 \times [P(DN) + P(TN) + P(T3)] + \\
P(T)/3 \times [P(DN) + P(TN) + P(T3)] + P(TN)/3 = \\
[P(SUP) + P(D)/2 + P(T)/3] \times \\
[P(DN) + P(TN) + P(T3)] + P(TN)/3
\]

Equation 7.9

\[
P(No \ eiLMIT) = \\
[P(SUP) \times P(SUP)] + [P(SUP) \times P(D2)] + \\
[P(SUP) \times P(T23)] + P(D)/2 + [P(D)/2 \times P(SUP)] + \\
[P(D)/2 \times P(D2)] + [P(D)/2 \times P(T23)] + \\
P(T23)/3 + P(T3)/3 + P(T)/3 + [P(T)/3 \times P(SUP)] + \\
[P(T)/3 \times P(D2)] + [P(T)/3 \times P(T23)] = \\
P(SUP) \times [P(SUP) + P(D2) + P(T23)] + P(D)/2 + P(T)/3 + \\
P(D)/2 \times [P(SUP) + P(D2) + P(T23)] + P(T23)/3 + \\
P(T)/3 \times [P(SUP) + P(D2) + P(T23)] + P(T3)/3 = \\
[P(SUP) + P(D)/2 + P(T)/3] \times [P(SUP) + P(D2) + P(T23)] + \\
P(D)/2 + [P(T23) + P(T3) + P(T)]/3 = \\
1 - \lambda_2
\]

Equation 7.10

7.2.8 Overall model for PC errors

Though \( \alpha \) has been calculated as a single value, it is necessary to work with each term separately when it comes to calculating EMIT and eiLMIT errors. This is due to the fact that both \( \lambda \) coefficients representing \( D_{EMIT} \), \( T_{EMIT} \), and \( T_{EMIT} \), have to be multiplied by each of the three terms in \( \alpha \)’s equation before being summed. Hence, the equation for eiLMIT is as follows:

\[
P(eiLMIT) = \\
([P(I21) + P(I22)] \times \lambda_1) + \\
[P(I32)/3 \times \lambda_1] + [P(I32)/3 \times \lambda_2] = \\
([P(I21) + P(I22)] \times \lambda_1) + [P(I32)/3 \times (\lambda_1 + \lambda_2)]
\]

Equation 7.11
The probability of EMIT is then:

\[
P(\text{EMIT}) = \left\{ \left[ P(I21) + P(I22) \right] \times (1 - \lambda_1) \right\} + \\
\left[ \frac{P(I32)}{3} \times (1 - \lambda_1) \right] + \left[ \frac{P(I32)}{3} \times (1 - \lambda_2) \right] \\
\left\{ \left[ P(I21) + P(I22) \right] \times (1 - \lambda_1) \right\} + \left[ \frac{P(I32)}{3} \times (2 - \lambda_1 - \lambda_2) \right]
\]

Equation 7.12

By combining the equations from previous sections, it is possible to calculate the probability of each of the four PC error classifications from the static and dynamic instruction-type profile of a program. The error classifications will be a product of these probabilities, as shown in Table 7.6.

<table>
<thead>
<tr>
<th>Error Category</th>
<th>Error Condition</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMIT</td>
<td>No</td>
<td>Normal</td>
</tr>
<tr>
<td>LMIT</td>
<td>No</td>
<td>LMIT</td>
</tr>
<tr>
<td>EMIT</td>
<td>Yes</td>
<td>EMIT</td>
</tr>
<tr>
<td>LMIT</td>
<td>Yes</td>
<td>LMIT</td>
</tr>
</tbody>
</table>

Table 7.6: PC error classification probability equations

7.3 Simulating the effects of MIT errors

To validate the statistical model, a ‘Monte Carlo’-style simulation was carried out on two real-world programs, ‘Prog’ and ‘Alarm’. Simulations scripts were written in VBScript and autonomously executed on 8051Sim.

7.3.1 Description of test programs

The test programs were written in C and compiled with the Keil C51 compiler. Although the aim of this simulation is to quantify the effects of MIT errors, NF and FT were also implemented as a side objective.

Prog is very similar to that used in Chapter 4. The main difference is the former Prog had its PFEH coded and located in assembly, whereas the version here codes the PFEH in C and locates it with linker directives. Alarm is an alarm clock program taken from [Pont 2001] that has a user interface made up of four buttons and a liquid-crystal display.
Techniques intended to reduce the impact of program-flow errors on embedded systems

As with the naming convention introduced in Chapter 4, ‘Alarm’ and ‘Prog’ (without suffix) would refer collectively to all variants of each program. ‘A programs’ would refer to ‘Alarm_A’ and ‘Prog_A’.

All programs have a high code-coverage level (> 80%) without any input stimulus. They use 8051-generic peripherals that are present in all variants of the 8051 family of microcontrollers.

The programs employ two different programming methodologies. Prog is written in a procedural manner. By contrast, Alarm is task-based and runs using a co-operative scheduler [Pont 2001]. Prog also uses code memory locations to store the coefficients for the FIR filter.

The source codes for the test scripts and programs can be found on the CD-ROM (see Appendix R for CD-ROM layout).

7.3.2 Simulation procedure

A large number of simulation cycles is required to provide a high level of confidence with the results obtained. These simulations are executed, without user intervention, under the control of the test script. An 8kB code memory device was assumed.

The simulation procedure – and effectively the entire test script – is graphically represented as a flowchart in Figure 7.3 (script listing is in Appendix N). Each program was subjected to 1000 simulation cycles with an error randomly injected between the first and 10,000,000th clock-cycle. Before each cycle commenced, the microcontroller is reset and the erroneous clock-cycle and erroneous PC value are pseudo-randomly determined. A breakpoint is set to ECY and the simulator began program execution.

When execution reaches the breakpoint, the breakpoint is cleared and an EMIT test (α-test) is carried out. This involved checking the multi-read flag38, which is set whenever a multi-read instruction is being executed and not all the operands have been read. If the multi-read flag is set, the simulation script test for EMIT or eiLIMIT errors (λ-test). On the other hand, a test for ‘Normal’ or LIMIT errors (β-test) is conducted.

38 This flag is part of the simulator and can be read externally.
Techniques intended to reduce the impact of program-flow errors on embedded systems

λ-testing involves checking for valid eiLMIT conditions: this required fewer test conditions than testing for non-eiLMIT errors. Detected eiLMIT errors are recorded as eiLMIT hits while non-eiLMIT errors are recorded as EMIT hits.

Testing for ‘Normal’ or LMIT errors (β-test) is simpler as it involves interrogating a table containing all known LMIT locations. The list of LMIT locations was generated with STATS (see Appendix J). LMIT errors are logged as LMIT hits while non-LMIT errors are recorded as ‘Normal’ hits.

If the simulation script has trouble determining the error classification (it is a complex process to determine eiLMIT/EMIT errors), an ‘Unknown’ (UNK1) error is recorded. This concludes the first phase of the experiment.

The next phase evaluates the effectiveness of NF and FT. Unlike the procedure carried out in Chapter 4, the simulation script checks to see if NF or FT had detected program-flow errors.
Techniques intended to reduce the impact of program-flow errors on embedded systems

for all errors categorised in the first phase (including those UNK1-classified). This classification happens after 10 additional clock-cycles are executed after the PC is corrupted to allow the processor to react to any potential MIT errors.

The simulation script determines if the PC has 'landed' in empty code memory locations, or within the program. If the former condition is detected, an NF hit is logged for programs employing NF, or 'E Jump' for those that do not. The simulation script then restarts another simulation cycle. If the PC 'lands' within the programmed code memory locations, up to an additional 50,000 clock-cycles (more than necessary to determine if an FT check detects the error) is step-simulated. The simulation script checks a reserved RAM location, upon every clock-cycle, that is set to a predetermined value if an FT check detects the induced error and branches to the PFEH. If such a situation occurred, an FT hit is recorded and the simulation process is restarted.

When none of the situations for the second phase is met, the simulation cycle is classified as 'Undetected Error' (UNDET). Under this classification, it cannot be assumed that the MCU would still be working as it should.

In the event that the simulator reports a simulator error, that simulation cycle is classified as 'Unknown' (UNK2). Such errors are accompanied with a returned value, which can be used to determine the type of error (see Appendix F).

### 7.4 Results and discussion

The experiments were carried out using 8051Sim-NG on two Pentium II 350MHz machines with 128MB RAM running Windows 2000. Each 1000-cycle simulation took, on average, 12 hours to complete.

The static and dynamic (for 10,000,000 clock-cycles) instruction-type profile for Alarm and Prog are shown in Table 7.7 and Table 7.8. A summary of the simulation results is shown in Table 7.9, while a comparison between the simulated and calculated results is shown in Table 7.10. Table 7.11 shows the correlation for α, β and λ.

---

39 This is about 30 times faster than the experiments carried out on dScope. Direct comparison however, is not possible since the simulations are not identical.
Techniques intended to reduce the impact of program-flow errors on embedded systems

7.4.1 Correlation between instruction-type and recorded errors

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Alarm</th>
<th>Prog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>S</td>
<td>#</td>
<td>642</td>
</tr>
<tr>
<td>P(S)</td>
<td></td>
<td>0.078</td>
</tr>
<tr>
<td>DN</td>
<td>#</td>
<td>722</td>
</tr>
<tr>
<td>P(DN)</td>
<td></td>
<td>0.176</td>
</tr>
<tr>
<td>D2</td>
<td>#</td>
<td>115</td>
</tr>
<tr>
<td>P(D2)</td>
<td></td>
<td>0.028</td>
</tr>
<tr>
<td>TN</td>
<td>#</td>
<td>288</td>
</tr>
<tr>
<td>P(TN)</td>
<td></td>
<td>0.105</td>
</tr>
<tr>
<td>T23</td>
<td>#</td>
<td>0</td>
</tr>
<tr>
<td>P(T23)</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>T3</td>
<td>#</td>
<td>33</td>
</tr>
<tr>
<td>P(T3)</td>
<td></td>
<td>0.012</td>
</tr>
<tr>
<td>UP/NF</td>
<td>#</td>
<td>4913</td>
</tr>
<tr>
<td>P(UP)</td>
<td></td>
<td>0.600</td>
</tr>
<tr>
<td>Code Size</td>
<td></td>
<td>3279</td>
</tr>
</tbody>
</table>

Table 7.7: Static instruction-type profiles for test programs

In theory, A and B (also between C and D) variants should have identical static profiles since they are the same at the higher abstraction level; the slight variation is due to optimisation by the compiler.

Alarm is a complex program and carries the overhead of a co-operative scheduler, hence its larger code size with respect to Prog. The same trend observed in Chapter 4 – the larger code sizes of C and D programs as compared with A and B – can also be seen in Table 7.7. This comes as no surprise: implementing FT requires additional code memory.

The overhead imposed by FT on Alarm is more than 100% of its original code size. This huge increase – 5 times over Coulson’s [Coulson 1998] estimate – is attributed to the numerous function calls that exist in Alarm. As a comparison, the code memory overhead for Prog remains at roughly 30% – about the same as that in Chapter 4. In short, the overhead due to FT increases when more FT checks are required, as predicted.
Based on Table 7.8, the differences in programming methodologies can be seen by comparing Prog and Alarm. In terms of instruction-types executed, Alarm spends between 55% and 58% of its time executing I32 instructions compared to just 2% to 4% for Prog. This large difference is attributed to the triple-byte instruction used for the ‘idle loop’; this loop is entered by the scheduler when no tasks are due to execute.

It cannot be concluded that time-triggered (and event-driven) programs are likely to prove more prone to EMIT and eILMIT errors because of the time the processor spends in the ‘idle loop’. In many time-triggered (and event-triggered) applications, the processor will be placed in the ‘idle’ mode when it enters the ‘idle loop’ to reduce power consumption [Pont 2001]. This may also mean that the system may have greater immunity to PC errors during this period. As 8051Sim is not able to reproduce the impact of idle mode, this issue is not addressed here.

One last issue regarding Table 7.8 is the total clock-cycles simulated is less than 10,000,000. This is because each interrupt generates a hidden LCALL instruction requiring two clock-cycles when vectoring to the ISR. The probability of each instruction-type occurring is therefore based on the number of clock-cycles in which instructions are executed, and not the total simulated.
Techniques intended to reduce the impact of program-flow errors on embedded systems

7.4.2 **Impact of programming methodology on error classification**

<table>
<thead>
<tr>
<th>Program</th>
<th>eiLMIT</th>
<th>EMIT</th>
<th>LMIT</th>
<th>Normal</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>%</td>
<td>#</td>
<td>%</td>
<td>#</td>
</tr>
<tr>
<td>Alarm_A</td>
<td>53</td>
<td>5.5</td>
<td>271</td>
<td>28.0</td>
<td>110</td>
</tr>
<tr>
<td>Alarm_B</td>
<td>57</td>
<td>6.0</td>
<td>270</td>
<td>28.3</td>
<td>117</td>
</tr>
<tr>
<td>Alarm_C</td>
<td>86</td>
<td>8.9</td>
<td>209</td>
<td>21.7</td>
<td>191</td>
</tr>
<tr>
<td>Alarm_D</td>
<td>94</td>
<td>9.7</td>
<td>215</td>
<td>22.1</td>
<td>170</td>
</tr>
<tr>
<td>Prog_A</td>
<td>10</td>
<td>1.0</td>
<td>187</td>
<td>19.3</td>
<td>49</td>
</tr>
<tr>
<td>Prog_B</td>
<td>16</td>
<td>1.7</td>
<td>161</td>
<td>16.8</td>
<td>47</td>
</tr>
<tr>
<td>Prog_C</td>
<td>11</td>
<td>1.1</td>
<td>147</td>
<td>15.1</td>
<td>74</td>
</tr>
<tr>
<td>Prog_D</td>
<td>18</td>
<td>1.9</td>
<td>134</td>
<td>13.8</td>
<td>79</td>
</tr>
</tbody>
</table>

Table 7.9: Simulation results for 1000 error cycles

The percentages shown in Table 7.9 for eiLMIT, EMIT, LMIT and ‘Normal’ errors occurring are based on the number useful simulation cycles, which is more than 95% of all simulation cycles generated.

It is evident that EMIT and eiLMIT errors occur more frequently than LMIT errors. This is expected: the probability of EMIT and eiLMIT errors occurring is related to the number of multi-read instructions. In all test programs, more than 50% of the programs’ duration is spent executing such instructions. Even so, EMIT and eiLMIT hits only account for about 32% and 17% of errors in Alarm and Prog respectively as not the entire multi-read period is susceptible to such errors.

Alarm recorded more EMIT/eiLMIT hits as compared with Prog. This is due to Alarm being a more complex program, which is also reflected in the dynamic instruction-type profile.

When FT is implemented, a slight decrease in EMIT/eiLMIT hits was detected for Alarm and Prog. The same trend is also seen in the percentage of multi-read instructions (I21-, I22- and I32-type) executed (see Table 7.8), indicating that FT could have reduced the number of multi-read instructions executed for a given time period. Although FT checks were deduced in Chapter 4 to contain a higher percentage of multi-read instructions than the program (Prog) itself, this statement is still valid. The apparent anomaly stems from the fact that there is no direct relationship between the static and dynamic profile.
Between 2.4% and 4.5% of the errors could not be classified as EMIT or eiLMIT. An examination of the log file for Alarm_A showed that 13 errors occurring at I14-type instructions and 18 for two-cycle instructions were classified as UNK1. These errors went undetected due to: 1) the conditions for I14-type instructions were left out from the script and, 2) the errors injected at sub-cycle 2 happened after the MCU was executed; hence only one other instruction needed to be executed instead of the two assumed by the script. Although rectifiable, the simulation was not repeated as it is extremely time consuming, and more than 95% of the results were usable.

### 7.4.3 Correlation between simulated and modelled results

To gauge the accuracy of the model, both the simulated and calculated results are shown together in Table 7.10. By comparing the simulated and calculated $\alpha$, $\beta$ and $\lambda$ coefficients (see Table 7.11), the accuracy of each part of the model can be determined.

<table>
<thead>
<tr>
<th>Program</th>
<th>eiLMIT</th>
<th>EMIT</th>
<th>LMIT</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sim</td>
<td>Cal</td>
<td>Dif</td>
<td>Sim</td>
</tr>
<tr>
<td>Alarm_A</td>
<td>5.47</td>
<td>7.99</td>
<td>2.52</td>
<td>27.97</td>
</tr>
<tr>
<td>Alarm_B</td>
<td>5.97</td>
<td>8.10</td>
<td>2.13</td>
<td>28.27</td>
</tr>
<tr>
<td>Alarm_C</td>
<td>8.95</td>
<td>11.83</td>
<td>2.88</td>
<td>21.75</td>
</tr>
<tr>
<td>Alarm_D</td>
<td>9.66</td>
<td>11.83</td>
<td>2.17</td>
<td>22.10</td>
</tr>
<tr>
<td>Prog_A</td>
<td>1.03</td>
<td>1.34</td>
<td>0.30</td>
<td>19.30</td>
</tr>
<tr>
<td>Prog_B</td>
<td>1.67</td>
<td>1.38</td>
<td>0.29</td>
<td>16.81</td>
</tr>
<tr>
<td>Prog_C</td>
<td>1.13</td>
<td>1.83</td>
<td>0.70</td>
<td>15.06</td>
</tr>
<tr>
<td>Prog_D</td>
<td>1.85</td>
<td>1.81</td>
<td>0.04</td>
<td>13.79</td>
</tr>
</tbody>
</table>

Table 7.10: Comparison between calculated and simulated results (%)
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Program</th>
<th>Coefficient $\alpha$</th>
<th>Coefficient $\beta$</th>
<th>Coefficient $\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm_A</td>
<td>33.44</td>
<td>39.12</td>
<td>5.68</td>
</tr>
<tr>
<td>Alarm_B</td>
<td>34.24</td>
<td>39.12</td>
<td>4.88</td>
</tr>
<tr>
<td>Alarm_C</td>
<td>30.70</td>
<td>36.23</td>
<td>5.53</td>
</tr>
<tr>
<td>Alarm_D</td>
<td>31.76</td>
<td>36.23</td>
<td>4.47</td>
</tr>
<tr>
<td>Prog_A</td>
<td>20.33</td>
<td>19.92</td>
<td>0.41</td>
</tr>
<tr>
<td>Prog_B</td>
<td>18.48</td>
<td>19.92</td>
<td>1.45</td>
</tr>
<tr>
<td>Prog_C</td>
<td>16.19</td>
<td>18.28</td>
<td>2.09</td>
</tr>
<tr>
<td>Prog_D</td>
<td>15.64</td>
<td>18.25</td>
<td>2.61</td>
</tr>
</tbody>
</table>

Table 7.11: Comparison of $\alpha$, $\beta$ and $\lambda$ coefficients

Table 7.11 compares the coefficients to give an indication of the accuracy of each part of the model. $\alpha$ is derived from summing the percentages of EMIT and eiLMIT hits. $\beta$ is derived from the ratio of eiLMIT hits to the sum of EMIT and eiLMIT hits. The same calculation method is also used to determine $\lambda$.

$\alpha$ showed the biggest difference between the simulated and calculated results in most situations, especially for Alarm. These differences show that the calculated results are generally higher than the simulated ones. The ‘unevenness’ of the instruction-types during different execution periods partially accounts for some of the differences. The $\beta$ and $\lambda$ coefficients however show less difference, testimony of the accuracy of the model.

7.4.4 The impact of NF and FT

The second phase of the experiment took place after the errors were injected to gauge the effectiveness of NF and FT in dealing with the different error scenarios. ‘Unknown’ (UNK1) errors from the first phase of the experiment were also evaluated in this phase.

The results obtained from the second phase of the experiment are shown in Table 7.12. Of the 1000 cycles simulated, over 95% of them were usable.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Program</th>
<th>E Jump</th>
<th>NF</th>
<th>FT</th>
<th>UE</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>%</td>
<td>#</td>
<td>%</td>
<td></td>
</tr>
<tr>
<td>Alarm_A</td>
<td>581</td>
<td>61.0</td>
<td>0</td>
<td>0.0</td>
<td>371</td>
</tr>
<tr>
<td>Alarm_B</td>
<td>0</td>
<td>0.0</td>
<td>580</td>
<td>60.8</td>
<td>4</td>
</tr>
<tr>
<td>Alarm_C</td>
<td>198</td>
<td>20.8</td>
<td>0</td>
<td>0.0</td>
<td>578</td>
</tr>
<tr>
<td>Alarm_D</td>
<td>0</td>
<td>0.0</td>
<td>203</td>
<td>21.4</td>
<td>604</td>
</tr>
<tr>
<td>Prog_A</td>
<td>788</td>
<td>81.1</td>
<td>0</td>
<td>0.0</td>
<td>184</td>
</tr>
<tr>
<td>Prog_B</td>
<td>0</td>
<td>0.0</td>
<td>747</td>
<td>76.7</td>
<td>3</td>
</tr>
<tr>
<td>Prog_C</td>
<td>687</td>
<td>71.2</td>
<td>0</td>
<td>0.0</td>
<td>211</td>
</tr>
<tr>
<td>Prog_D</td>
<td>0</td>
<td>0.0</td>
<td>695</td>
<td>73.1</td>
<td>186</td>
</tr>
</tbody>
</table>

Table 7.12: Effectiveness of NF and FT in detecting program-flow errors

As with the results shown in Chapter 4 (see Table 4.7), programs implementing at least one form of error detection technique are better off than those not implementing any. When both NF and FT were implemented, over 85% of program-flow errors were detected. This is higher than when either technique was implemented. However, it may be better to implement NF alone instead, for the reasons discussed in Chapter 4.

The error detection rate for NF – in many cases the more desirable technique – is highest when NF alone is implemented. This is again to be expected, based on the discussion in Chapter 4.

Another intriguing aspect from Chapter 4’s discussion – obvious in Table 7.12 – is the relationship between the NF region, physical code memory size and program size. Since the implementation of FT doubled Alarm’s program size compared to a 30% increase for Prog, Alarm’s NF size shrank by a much larger margin (60.5% to 20.3% for Alarm compared with 76.5% to 69.5% for Prog). Interestingly, this shrinkage is mirrored in the number of NF hits recorded by Alarm and Prog.

Between 26 and 49 simulation cycles resulted in errors. These errors were trapped and logged by the simulation script. Upon analysis of Alarm_A’s simulation log, 15 UNK2 categorised errors were caused by the false execution of a RETI instruction. This meant that the PC error vectored program-flow into the ISR. The remaining 33 errors were caused by trying to access non-existing XRAM locations, which was the result of executing MOVX instructions. As no MOVX instruction was used in the original program, these may have been the result of LMIT errors.
7.5 Conclusion

In this chapter, the complete error model has been shown to be effective in predicting the probability for 'Normal', LMIT, EMIT and eiLMIT error categories, based on a program's static and dynamic instruction-type profile. Different programming and design methodologies have also been shown to influence the instruction-type profiles, which in turn would affect the error classification probabilities.

In accordance with the conclusions drawn in Chapter 4, NF and FT have also been shown to be very effective in dealing with program-flow errors when EMIT and eiLMIT errors are considered. The experiments carried out in this chapter further proofs NF has a high error detection rate, low error detection latency, low requirement on processor resources, and is easy to implementation.
PART THREE

CHIP-BASED PROGRAM-FLOW ERROR DETECTION TECHNIQUES

In Part Two, Program Counter corruption was investigated. The outcome of the investigation was the conclusion that EMIT, LMIT and eiLMI error conditions can cause instruction misinterpretation.

Two software-based techniques which are intended to cope with such situations were discussed, modelled and simulated. These techniques are not the only ones capable of detecting and/or correction program-flow errors, however. In Part Three some proposed chip-based techniques are introduced and evaluated. Part Three also proposes other chip-based techniques that may be more suitable for embedded environments.
Existing chip-based program-flow error detection and correction techniques: A review

It was demonstrated in Part One and Part Two that NF and FT both detect program-flow errors. As discussed in Chapter 1 however, techniques to detect and/or correct program-flow errors are not just limited those that are software-based. Indeed, such errors can also be detected and/or corrected with chip-based techniques, as discussed in Part Three.

This chapter gives an overview of hardware-based techniques proposed by other authors that are integrated with processor core. Only techniques that can detect and/or correct program-flow errors are considered here.

8.1 Overview of chip-based techniques

Various chip-based techniques have been successfully implemented on processors used in ‘mission critical’ computing systems (e.g. space exploration, military) for many decades. Indeed, chip-based techniques have been developed for virtually every module in a processor such as RAM (Error-Correcting Codes – ECC [Grey 2000]), cache (Cache Write Sure [Kim & Somani 1999]) and memory access (Memory Access Checking [Mahmood & McCluskey 1998]).

Although chip-based techniques were mainly developed for complex and important computer systems, some of them may be tailored for use in embedded processors, or their principles used as the basis of other techniques.

The various chip-based techniques can be divided into five groups:

- **Data duplication** – duplicates the values of important registers
- **Parity-based mechanisms** – uses additional bits to help detect errors
- **Signature Monitoring** – uses additional bits/instructions to calculate program flow
- **Micro Rollback** – a recovery mechanism that restores a processor to a ‘good’ state
- **Watchdog Processor** – a co-processor that tracks the processor’s state
Due to the wide variations possible with some techniques, the discussion mainly focuses on their generic version. Most of these techniques are also dependent on the processor’s architecture [Beuscher & Toy 1970].

8.2 Data Duplication
Data Duplication (DD) [AN435 1998, Burnham & Cowling 1984, Shuette & Shen 1987] is a simple technique where the PC is replicated (see Appendix D for a general, more in-depth discussion). When the PC is written, its ‘replicated’ registers are also written with the same (or complementary) value. Upon PC reads, its ‘replicated’ registers are also read, and compared. Non-identical values would signify PC (or register) corruption, which may be corrected by other techniques (e.g. ‘Majority voting’ [Niaussat 1998, AN435 1998, Campbell 1995, Campbell 1998, Caldwell & Rennels 2000, MISRA 1994, Rennels et al. 1997]).

In general, DD’s will probably have very high (approaching 100%) error detection rates – particularly when each ‘critical register’ has many ‘replicated’ registers – since the chances of the ‘critical’ and ‘replicated’ registers having the same corrupted value is minute. The gate count overhead for DD should not be significant if implemented on the PC alone. On the other hand, if DD is implemented on many registers, each with multiple ‘replicated’ registers, the overheads involved can be substantial. DD may also increase the register access times, particularly write accesses, due to the checking circuitry (e.g. ‘Majority Voting’ involved).

Data Duplication does not require any processor resources such as RAM or clock cycles. Since this technique is transparent to the program, alterations to the program and development tools are not necessary.

8.3 Parity-based Mechanisms
Parity-based error detection Mechanisms (PM) [Beuscher & Toy 1970, Gaisler 1997, Grey 2000] employ one or more additional bit(s) to make the number of ‘1’ or ‘0’ of a register equivalent to a necessary amount. In its simplest form, known as modulo-2, an extra bit is used to make the number of ‘1’s in a register even (even-parity) or odd (odd-parity). This mechanism will only detect single bit errors, and correction is not possible [Sridhar & Thatte 1982]. When more bits are used however (multi-bit parity), multi-bit errors can be detected and corrected [Sridhar & Thatte 1982] (e.g. Hamming ECC [Grey 2000], extended parity
Techniques intended to reduce the impact of program-flow errors on embedded systems

check schemes [Beuscher & Toy 1970]). Note: PMs are at times used as the basis of other schemes such as those described in [Beuscher & Toy 1970].

8.3.1 Evaluation of Parity-based Mechanisms
Parity-based Mechanisms can be implemented on most registers and buses. The hardware overhead involved in generating and checking each parity-enabled register is small (if using modulo-2); it does however increase with the number of parity bits employed. Gray showed this relationship for Hamming ECC in [Grey 2000].

If PMs fail to detect an error, the error cannot be detected at some later stage (i.e. it does not have an error detection history). This limitation is due to the fact that PMs work on the sub-cycle basis, and there is no interaction between the cycles. On the other hand, PM is not affected by interrupts.

Although parity checking occurs concurrently with processor execution, there will be some performance degradation due to the latency of the parity generation, and the need to check logic gate circuitry. This delay will only be a problem for very fast processors: for example, Gray mentioned that a 64-bit word employing Hamming ECC increases the memory’s read and write time by 2.2 ns and 1.6 ns respectively [Grey 2000].

PMs have been shown to achieve very high error detection rates. Gaisler showed in [Gaisler 1997] that a SPARC compatible processor (the ERC32) was able to detect more than 97.5% of errors when 99% of the processor’s registers implemented parity. This high error detection rate comes with a price: the increase in implementation size. Implementation size will also increase when PMs are required to correct errors: they can only do so when multiple parity bits are used.

PMs will have very short error detection latencies as they basically operate on the clock edge. In addition, error correction takes place almost immediately, thus making it suitable for real-time applications. Programs and development tools do not need to be modified for use on processors with parity-based mechanisms. However, error-handling mechanisms may be required to handle the errors that are detected.

Before leaving this topic, it should be noted that Li et. al. [Li et. al. 1984] described a technique know as ‘Safe ROM’ where an additional bit is used to identify the instruction boundary in code memory locations. This technique, which can be considered a kind of PM,
Techniques intended to reduce the impact of program-flow errors on embedded systems

increases the code memory overhead by 12.5% (1 extra bit out of 8 bits) and may not have good error detection rates, especially when, in general, a majority of a program's instructions are single byte.

8.4 Signature Monitoring

Signature Monitoring (SM) [Wilken 1993] is a category of error detection techniques involving the use of assigned or derived program signatures to check program-flow. Assigned signatures are unique 8-, 16- or 32-bit values that are arbitrarily assigned by the programmer or compiler for each node during implementation (i.e. this is analogous to a function's ID). On the other hand, derived signatures are calculated based on a certain function applied to instructions in a node [Mahmood & McCluskey 1998, Saxena & McCluskey 1990, Shuette & Shen 1987, Sridhar & Thatte 1982]. This is similar to FT, apart from it being implemented in software at the source code level.

The assigned or derived signatures are inserted into the nodes at the start or the end of the node [Shuette & Shen 1987], and are compared with the signatures computed during processor execution. These computed signatures are generated concurrently with instruction execution by custom hardware [Mahmood & McCluskey 1998]. When specific conditions are met (e.g. when special instructions are executed [Wilken 1993] or just before a program-flow branch), the computed and assigned/derived signatures are checked. Any difference between the two signatures signifies program-flow errors [Shuette & Shen 1987, Sridhar & Thatte 1982].

The generation of assigned and derived signatures (e.g. path signature analysis, branch address hashing) is itself an area of research. A description of the various schemes can be found in [Mahmood & McCluskey 1998, Shuette & Shen 1987]. The placement of signatures, an issue that will have an impact on processor overhead, is discussed in [Wilken 1993].

40 Also known as Control-Flow Checking [Saxena & McCluskey 1990], Signatured Instruction Stream [Shuette & Shen 1987] or Parallel Signature Analyser [Velazco et. al. 2000].

41 A node is a segment of instructions without branches to or from it [Mahmood & McCluskey 1998].
8.4.1 Evaluation of Signature Monitoring

Schuette & Shen [Shuette & Shen 1987] reported an overall program-flow error detection rate of 82% for signatured instruction streams. This level of detection (80%) was also reported by Mahmood & McCluskey [Mahmood & McCluskey 1998]. Since SM can only detect errors, other techniques are necessary to correct them. One approach to implement SM with error correction techniques on a Watchdog Processor, as discussed by [Mahmood & McCluskey 1998, Saxena & McCluskey 1990] (see in Section 8.6).

The program size and processing overhead when SM is implemented depends on each program's characteristics (e.g. number and frequency of program-flow branches). As a rough estimate, Schuette & Shen [Shuette & Shen 1987] reported a program size and processor overhead of 26% and 10% respectively when implemented on the Motorola MC68000 processor. He also mentioned that the implementation increased the processor's gate count by approximately 17% (3947 gates – the MC68000 has approximately 23000 gates [Shuette & Shen 1987]). This is within Mahmood & McCluskey's [Mahmood & McCluskey 1998] estimate of between 10% and 20%.

The average error detection latency is dependent on the average number of instructions executed between two successive checks [Sridhar & Thatte 1982], which is the same as FT’s. It is possible however, to detect errors within one clock cycle: this occurs when the PC 'lands' right on an SM check. On the other hand, SM's maximum latency will be the time taken to transverse the longest node.

Since the assigned or derived signatures are implemented as part of the code, the processor has to be tailored to differentiate program-flow checking instructions from others [Saxena & McCluskey 1990, Shuette & Shen 1987]. This is achieved either by adding additional bits to an instruction (i.e. widening the instruction width) or to create instructions just for program-flow checking [Saxena & McCluskey 1990, Sridhar & Thatte 1982]. In both cases, the instruction set and programming tools must be altered [Shuette & Shen 1987, Wilken 1993], unless the approach described in the next paragraph is taken. This is not commercially viable for most companies as new tools may have to be acquired. The programs however, do not need modifications, as long as they are not written in assembly42.

42 The signatures can be inserted by specialised tools.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Mahmood & McCluskey [Mahmood & McCluskey 1998] describe a scheme for the MC68000 processor where a jump instruction that displaces execution by two words is inserted before every signature. Since the MC68000 prefetches the next executable instruction, the SM hardware can access the signature, but the processor does not execute it. This scheme was implemented by Schuette & Shen [Shuette & Shen 1987]. Note: such a scheme may not be applied on the 8051 MCUs as most, if not all, versions do not prefetch instructions, and the 8051 has only one unused instruction mnemonic.

SM monitors program-flow over many instruction cycles (to be exact, over the length of each node) as compared with parity-based mechanisms. This allows it to detect program-flow errors that could have occurred at some earlier stage of execution (e.g. corrupted data memory location used to compute branching location). Its drawback is signatures have to be saved when interrupts occur [Shuette & Shen 1987], which will require additional hardware and possibly processor time.

8.5 Micro Rollback

Micro Rollback (MR) is an error recovery technique which involves returning the processor to a previous known good state [Fujimoto et. al. 1992, Tamir et. al. 1988, Tamir & Tremblay 1990] in the event of an error. In order to do this, snapshots of the processor’s state (values of registers) are taken at every clock-cycle, a process known as checkpointing [Tamir et. al. 1988, Tamir & Tremblay 1990]. The number of ‘snapshots’ determines the maximum rollback distance (recoverable cycles) that can take place, in order to guarantee correction. This latency should be greater than the worst-case latency of the slowest error detection and/or correction technique [Tamir & Tremblay 1990].

The ‘snapshots’ are saved in first-in-first-out (FIFO) register or RAM arrays [Tamir et. al. 1988, Tamir & Tremblay 1990]. Each register will require its own FIFO/RAM array that is equivalent to the width of the register and depth of the rollback distance. Each bus and non-addressable registers that are responsible for the processor’s state will also require a FIFO or RAM array [Tamir et. al. 1988].

When an error is detected by some other techniques (e.g. Signature Monitoring), MR will be activated to correct the processor. The saved ‘snapshot’ just before the error occurred will be read by the MR circuitry and the respective registers overwritten with the snapshot’s value.
Techniques intended to reduce the impact of program-flow errors on embedded systems

This process is carried out for all FIFO/RAM arrays; hence, correction is done in parallel in a single cycle [Tamir et. al. 1988, Tamir & Tremblay 1990].

MR can be implemented as part of the processor, or it can be implemented as a co-processor. The latter approach (using a Rollback chip) is described by Fujimoto et. al. [Fujimoto et. al. 1992]. Please refer to [Tamir et. al. 1988, Tamir & Tremblay 1990] for a detailed description of Micro Rollback.

Note: Hwu & Patt [Hwu & Patt 1987] describe a similar checkpoint and rollback technique that is used to repair out-of-order execution processors. A software-based analogy of this technique is the System Restore functionality built into Windows ME/XP.

8.5.1 Evaluation of Micro Rollback

Micro Rollback is only an error recovery technique [Tamir et. al. 1988]. The rollback latency is short: it can be performed within one-clock-cycle [Tamir & Tremblay 1990].

MR's implementation overhead can be significant especially for processors with many registers: each register requires a FIFO/RAM array that is the depth of the rollback distance [Tamir & Tremblay 1990]. Storage units that are solely connected to a bus do not require FIFO/RAM arrays. Instead, one FIFO/RAM array is implemented for each bus (assuming only one memory unit can be connected to the bus at any given time). Even so, if a processor with a (say) 16-bit internal bus and (say) ten 8-bit registers implement MR with a rollback distance of 20, the implementation overhead for the FIFO/ROM alone will be 960 flip-flops/RAM bits. When the support circuitries are included, MR's implementation will be significant, as predicted by Tamir & Trembley [Tamir et. al. 1988, Tamir & Tremblay 1990].

Although the 8051 family has relatively few registers requiring FIFO/RAM units for MR (e.g. PSW, ACC, TMP1, TMP2 – see Figure 2.2), its high clock cycle to instruction cycle ratio (i.e. 1, 2 or 4 cycles per instruction cycle [Atmel 1997, Siemens 1996a]) means the rollback distance may have to be big. For example, if the maximum latency for an error detection technique is two instruction cycles, the rollback distance has to be at least 8. In short, MR is unsuitable for the 8051 family of MCUs; it is more suited for simple processors with few registers (that needs FIFO/RAM) and low oscillation to instruction cycle ratio. Note: if timing related registers (e.g. TL0, TH1) have to be rollback, MR's implementation overhead will increase further.
One potential problem related to MR’s implementation overhead is that it may present a large cross-sectional area (die area) for corruption from EMI or high-energy particles. If MR is corrupted, it may not work correctly, or the processor’s state may be incorrect after a rollback if the FIFO/RAM is corrupted [Tamir et. al. 1988].

Because MR reverses the state of the processor, there is a potential problem with interrupts: if they happen between the error and rollback occurring, the rollback will overwrite the interrupt, causing the processor to miss it, or part of an ISR may be executed twice. This may not be tolerable for real-time systems: possible workarounds are described in [Smith & Pleskun 1998, Torng & Day 1993].

MR uses the same mechanism to save and restore registers and memories for all processors; it is only the size and width of the FIFO/RAM array that changes. Hence, this technique is independent of the processor’s architecture (it is proxy-based just like SecurePorts – see Chapter 17) [Tamir & Tremblay 1990]. On the other hand, full knowledge of the processor’s architecture will be necessary to implement MR correctly.

MR is transparent to the program [Tamir & Tremblay 1990]. This means that program modification is unnecessary, and that the current development tools can also be used without changes. It can also be independently implemented for each module [Tamir & Tremblay 1990].

8.6 Watchdog Processor

The Watchdog Processor (WP) is a scaled down processor (i.e. a co-processor) that performs concurrent system-level error detection (and possibly correction) by monitoring the behaviour of the main processor. Parameters that could be monitored include the internal buses, the external buses and the state of the processor [Mahmood & McCluskey 1998, Saxena & McCluskey 1990]. Although the WP is described as a separate standalone device [Mahmood & McCluskey 1998], it could potentially be integrated with the processor.

43 This is not the same as the Watchdog Timer described in Chapter 14, but it can incorporate the Watchdog Timer.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Error detection takes place in two stages: 1) the WP is provided with information relating to the processor or the process to be checked, and 2) it monitors the processor and/or error detection circuitry to collect the relevant information concurrently [Mahmood & McCluskey 1998]. The information provided to the WP could be in the form of memory access behaviour, state of control signals, or the reasonableness of results [Mahmood & McCluskey 1998]. If the watchdog processor detects any abnormal system behaviour, it can stop the processor and possibly execute some form of error correction (e.g. Micro Rollback [Fujimoto et. al. 1992, Tamir et. al. 1988, Tamir & Tremblay 1990]).

The WP is not only limited to detecting and/or correcting errors by tracking the internal buses and the state of the processor (e.g. sequencer), it is also a means of controlling other error detection and/or correction techniques indirectly (e.g. Signature Monitoring [Mahmood & McCluskey 1998]).

8.6.1 Evaluation of the Watchdog Processor

The Watchdog Processor circuitry is independent of the processor’s: this provides protection against common or related errors due to design diversity [Mahmood & McCluskey 1998]. The monitoring process is also done concurrently and is unobtrusive to the processor’s execution.

Since the watchdog processor is a stripped down version of the processor that it monitors, the implementation overhead is dependent on the complexity of the processor’s architecture and the checks that are required44. Mahmood & McCluskey [Mahmood & McCluskey 1998] mentioned that the WP’s implementation overhead (without other error detection and/or correction techniques) could be as low as 20% of the complexity of the MC68000 processor (i.e. 4600 gates based on the estimated gate count of 23000 [Shuette & Shen 1987]).

The WP’s error detection latency could be as short as one clock-cycle since it constantly monitors the processor’s buses and states. When other techniques are controlled by it (e.g. Signature Monitoring), the error detection latency then depends on the technique’s latency. The error correction latency will also depend on the technique implemented.

44 Implementation overhead of error detection and/or correction techniques are considered separately.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The WP can be controlled by the program, or a separate ‘monitoring’ program can be loaded into the WP’s memory [Mahmood & McCluskey 1998]. The former approach treats the WP like a peripheral, which means program modification will be necessary to insert the required checking instructions. These checking instructions should be inserted at the assembly language level for the best coverage; hence, the development tools should be modified. Note: there will be processing overheads here since the program has to configure the WP when necessary. This is not the case with the latter approach.

The latter approach treats the WP as another processor. In this situation, programs do not require modifications. The ‘monitoring’ programs have to be written for the WP however [Mahmood & McCluskey 1998]: this can be done using available development tools. On the other hand, if some WPs have specific instructions that differ from a processor’s instruction set, new development tools may be required.

Since the WP is specifically tied to one processor family, they cannot be ported to other families without non-trivial changes. To tailor the WP for other processor families, intimate knowledge of the processor core is necessary.

8.7 Evaluation of chip-based techniques

Each of the four chip-based techniques discussed in the previous sections have their own advantages and disadvantages. One issue in common with them is they were primarily designed for large computing systems where cost, error detection latency, and processing overhead (except processing time) are not mitigating factors. On the other hand, these factors are important when it comes to mass-produced embedded systems.

The production cost overhead increases with implementation size due to larger die area of the processor. Of the four techniques, parity-based mechanisms should have the lowest overhead when it is only implemented on registers that are related to program-flow. Signature Monitoring has been reported to increase implementation overhead by 17% [Shuette & Shen 1987]. This may be less than Micro Rollback and the Watchdog Processor (quoted by [Mahmood & McCluskey 1998] at a minimum of 20%).

PM, SM and WP are techniques that detect program-flow errors. Although their effectiveness has been discussed in the literature [Gaisler 1997, Mahmood & McCluskey...
Techniques intended to reduce the impact of program-flow errors on embedded systems

1998, Shuette & Shen 1987], direct comparisons is not possible due to the different ways authors classify their results, implement the techniques and also carry out the experiments. In any case, PM and SM have been reported to detect over 80% of injected errors.

If PM does not detect errors when they occur, it will not detect them at some later stage. SM and WP may, since they work at a higher level of abstraction. The drawback is SM (and probably WP) will not handle interrupts correctly unless additional hardware is used [Shuette & Shen 1987]. This will further increase their implementation size.

The error detection latency is almost zero for PM and WP. Both techniques can detect errors as soon as they occur. This makes them suitable for real-time systems. SM’s error detection latency varies between the minimum and maximum between signature checks, just like FT (see Chapter 4).

PM and WP have the potential to correct errors. This comes at a high price however; PM requires multi-bit parity (e.g. Hamming ECC) for correction [Sridhar & Thatte 1982] while WP has to incorporate other techniques (e.g. MR). Hence, the implementation overhead for PM and WP will increase when error correction is involved. Problems could also arise when MR’s rollback causes loss of data, re-execution of ‘critical instructions’ or missing/re-executing interrupts.

PM and MR’s implementation is transparent to the program, which means program and development tool modifications are unnecessary. Moreover, the processor core itself should remain relatively intact: PM and MR are only auxiliary components. Although SM is also transparent to the program, it does require processor core/instruction set and development tool modifications to derive and insert the signatures into the program code. This potentially makes SM costly. Code portability is not an issue for SM, code generation is. The WP require changes to the program when it is controlled by the program, or changes to the development tools when it requires its own ‘monitoring’ program. In both cases, some changes to the program/development tools are necessary.
8.8 Conclusion

The chip-based techniques presented in this chapter have various advantages and disadvantages from the perspective of their feasibility for embedded systems. Some techniques such as Parity-based mechanisms and Data Duplication can be feasibly implemented in such systems. Although other techniques may be less suitable, nevertheless their principles could form the basis of more feasible techniques.
It was concluded in the previous chapter that not all existing chip-based techniques to detect and/or correct program-flow errors are suitable for embedded systems, although some of their principles could be tailored to do so. In this chapter, a novel chip-based technique that is potentially suitable for embedded systems is discussed and evaluated.

9.1 Overview of State Tracking
State Tracking (ST) is a technique that detects and corrects register corruption. Although this technique could be implemented on most registers, ST is implemented on the Program Counter and the discussion in this chapter centres on the PC.

9.1.1 Definition of ST’s processes and terms
Before ST is described in detail, it is easier to understand the description if its operations are first described. These are as follows:

- **Store** – This operation involves loading the SPC (see next sub-section) with the PC’s value.
- **Check** – This operation involves comparing the PC with the SPC.
- **Restore** – This operation involves loading the PC with the SPC’s value.

The terms ‘PC read’ and ‘PC write’ are used to denote the read and write operations that take place during S1P2 and S2P1 (and also S4P2 and S5P1) respectively when PC increments occur. Both operations are collectively termed ‘PC increments’. ‘PC write’ also denotes the write operation that takes place during S6P2 due to legitimate program-flow branches.

9.1.2 ST’s cycle level operation
State Tracking is based on the way instructions are executed on an 8051 processor. By implementing another register (known as the Shadow Program Counter – SPC) that acts as a

---

45 PC increments are actually PC reads followed by writes. The PC is read into an incrementer, incremented, and written back during the next sub-cycle.
Techniques intended to reduce the impact of program-flow errors on embedded systems

back up to the PC, it is possible to detect corruption of the PC. Error correction may then be carried out by overriding the PC with SPC's value. ST's technique can be similar to the Watchdog Processor [Mahmood & McCluskey 1998] when it is used to monitor register accesses, and its error correction technique is similar to Data Duplication (see Chapter 8).

ST executes at the clock-cycle level and repeats the same procedure indefinitely. The ST processes for one clock-cycle are illustrated in Figure 9.1. PC reads and writes are denoted at the start of their respective sub-cycle.

Referring to Figure 9.1, the idea behind ST is to perform a 'store' at every sub-cycle that the PC's value may change. This applies to sub-cycles S2P1, S5P1 and S6P2. One sub-cycle before the PC may be read however (S1P1 and S4P1), a 'check' is performed to ensure the PC and the SPC have the same value. Any difference will signify either the PC or the SPC (or both) have been corrupted sometime between a 'store' and a 'check'. When such a situation occurs, a 'restore' is performed. Note: since the SPC is not directly connected to the address bus (see Figure 9.2), it is more likely that the PC is corrupted when its value and SPC's differ.

'Checks' are also performed at sub-cycles S1P2, S4P2 and S6P1 before the sub-cycle where the PC may change. These 'checks' are designed to detect PC corruption before 'store' occurs for sub-cycles that the PC are not overwritten. Without these 'checks', the 'store'

---

46 The PC will not always be read on S1P2 and S4P2, or written to on S2P1, S5P1 and S6P2. The reading and writing solely depend on each instruction.

47 If the PC is incremented or otherwise changed, a write occurs that could correct the corruption.
Techniques intended to reduce the impact of program-flow errors on embedded systems

operation could have loaded the SPC with the corrupted PC value. In short, a 'check' precedes each 'store' and PC read.

Since ST only tracks the sub-cycle states and not the instructions, ST has to perform the operations mentioned above for all instruction-types. For example, 114-type instructions will perform 20 'checks' and 12 'stores' although only two 'checks' and one 'store' are necessary.

9.2 Functional description of ST

Figure 9.2 shows the block diagram for ST. The blocks in inverted colours are those present in the 8051 core, the rest are part of ST. The solid lines represent data lines/buses while dashed lines are the control signals.

![Block diagram for State Tracking](image)

The heart of ST is the SPC, a 16-bit register that is loaded with the value of the PC whenever the PC-S signal performs a low-to-high transition. In a similar vein, the PC will - on a low-to-high transition of the S-PC signal - be loaded with the value of SPC. Note that the PC has to be modified to accommodate the reading from two sources: the SPC and the internal address bus.

The 'Cycle Register' (CREG) acts like the 8051 sequencer. The main difference is that the CREG does not have any knowledge of the instructions that are executed. This duplication of the processor's own sequencer makes ST almost self-contained and, as far as possible, independent of the processor.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The CREG sets the level of its two outputs, \texttt{CHK} and \texttt{STR}, at the start of the appropriate sub-cycles. These outputs will remain high for the duration of the ‘check’ and ‘store’ operations respectively. The CREG is also responsible for producing the states pertaining to the ‘quan-cycles’, which are discussed in the next section.

The ‘Comparator’ (COMP) is a simple device that goes high whenever the values of the PC and the SPC are different. When both inputs are the same, \texttt{CMP} is low.

The control of data flow between the PC and the SPC is taken care of by the ‘Control’ (CTRL) module. This module acts upon signals from the CREG according to the following rules:

- When \texttt{CHK} is high, a ‘restore’ will take place if, and only if, \texttt{CMP} is high. A ‘restore’ is carried out by pulsing the \texttt{S-PC} line. The \texttt{ERROR} signal will also be taken high in this situation. This signal can be used to trigger an interrupt.
- When \texttt{STR} is high, the \texttt{PC-S} line will be pulsed.

Although the SPC is not directly connected to the internal buses of the processor, there is still a small chance that it (or other components in ST) gets corrupted, causing ST to reload the PC with the corrupted value. Hence, to prevent such situations from going unnoticed, the \texttt{ERROR} signal is set whenever the PC and the SPC differ during ‘check’ operations.

9.2.1 Sub-cycle timing

There are certain periods in each clock-cycle where ST will not be able to detect PC corruption (discussed in the next section). To reduce this period, the ‘check’ and ‘store’ operations have to be performed as fast and late (just before the PC is read or just after it is written) as possible. Hence, it was decided that ST should be executed at four times the processor’s frequency (clock-quadrupled). In practice, clock-doubling is sufficient to carry out the required operations within one sub-cycle. Clock-quadrupling is instead implemented as this gives room for potential expansion, and decreases the likelihood of undetected errors.

A new term, ‘quan-cycle’ (quantum-cycles) is used to describe the clock-quadrupled cycles. Four ‘quan-cycles’ – labelled from 0 to 3 (ascending time order) – will take place during each sub-cycle.
For the PC to be able to load on a ‘quan-cycle’ boundary, it either must be asynchronously loadable (i.e. the flip-flop states are set with the ‘set’ and ‘reset’ inputs), or clocked at the ‘quan-cycle’ level (‘normal’ flip-flop clocking). Since clocking the PC at a higher rate may not be possible (the PC is part of a complex synchronous machine), asynchronous loading was chosen. This method can safely be applied since the PC is only read/written on specific sub-cycles. Therefore, as long as the PC is loaded by ST when it is not connected to the address bus, bus contention will not occur.

To decrease the period where the PC and the SPC’s value can change during asynchronous loads, the PC and the SPC are designed to only load on the edge transition of the $s_{PC}$ and $p_{c}$ signals respectively.

### 9.2.2 ‘Check’ and ‘Restore’ operations

‘Check’ and ‘restore’ operations are carried out together since $\text{CMP}$ is derived from combinational logic, and the $s_{PC}$ level can be determined by ANDing $\text{CMP}$ and $\text{CHK}$ with the correct ‘quan-cycle’. The ‘check’ and ‘restore’ operation on the ‘quan-cycle’ basis is shown in Figure 9.3.

![Figure 9.3: 'Check' and 'Restore' operations at the 'quan-cycle' level](image)

Note: the ‘check’/‘restore’ operation happens just before the sub-cycle when the PC is read, or a ‘store’ takes place. This allows ST to detect errors up to ‘quan-cycle’ 3; hence reducing ST’s period where errors are undetectable.

### 9.2.3 ‘Store’ operations

The ‘store’ operation is simple: between ‘quan-cycles’ 1 and 2, the $p_{c}-S$ line is pulsed, which means that the SPC is loaded with the value from the PC at ‘quan-cycle’ 1. The SPC is loaded at ‘quan-cycles’ 1 instead of ‘quan-cycles’ 0, as this gives the PC time to settle at its...
Techniques intended to reduce the impact of program-flow errors on embedded systems

level when it is written at 'quan-cycle' 0. The timing diagram for a 'store' operation is shown in Figure 9.4.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{figure9_4}
\caption{'Store' operation at the 'quan-cycle' level}
\end{figure}

9.3 Error detection and correction
Before the estimation on ST’s error detection and correction rates is carried out, it is necessary to define some of the terms and classifications used.

9.3.1 Classification of 8051 instructions
The 8051 instructions are categorised in three types: 'Non-branch', 'Conditional branch' and 'Unconditional branch'. The following list describes each classification:

- **Non-branch** – Instructions that will not cause program-flow branches. They will only increment the PC to point to the next instruction.
- **Conditional branch** – Instructions that would cause program-flow branches when certain conditions are met.
- **Unconditional branch** – Instruction that will cause program-flow branches under any circumstances.

Examples of the instruction types are ‘DEC @R1’, ‘JNB 0xB5, 0x50’ and ‘LJMP 0x042A’ respectively. The term ‘branching instructions’ collectively refers to conditional and unconditional branching instructions. Note: all branching instructions are of type I21, I22 or I32.

9.3.2 Categorising injected errors
The categorisation of errors injected into the PC is vital in differentiating the actions taken by ST, and their overall effects to the system. A two-tier classification approach is taken: the first shows how ST handles the error (DETCOR, DET, UNDET and UNK) and the second
Techniques intended to reduce the impact of program-flow errors on embedded systems
classification is based on the overall effect of the injected errors on program-flow (COR and
UNCOR), as shown in Table 9.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Error type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DETCOR</td>
<td>Detected &amp; Correctable</td>
<td>Detected and correctable PC errors</td>
</tr>
<tr>
<td>DET</td>
<td>Detected</td>
<td>Detected PC errors</td>
</tr>
<tr>
<td>UNDET</td>
<td>Undetected</td>
<td>Undetected PC errors</td>
</tr>
<tr>
<td>UNK</td>
<td>Unknown</td>
<td>Simulator error occurred</td>
</tr>
<tr>
<td>COR</td>
<td>Corrected</td>
<td>Corrected PC errors</td>
</tr>
<tr>
<td>UNCOR</td>
<td>Uncorrected</td>
<td>Uncorrected PC errors</td>
</tr>
</tbody>
</table>

Table 9.1: Summary of simulation error classification

Injected PC errors that are not detected by error detection and correction techniques are
classified as UNDET. If the injected errors are detected, they are classified as DET – or
DETCOR when they are also corrected. Simulator errors are classified as ‘Unknown’ (UNK)
and takes precedence over DETCOR, DET and UNDET. This completes the first
classification.

The COR and UNCOR error classifications are carried out from a different perspective. An
error is only classified as COR when program-flow is unaffected as the result of ST
correcting the error, or PC writes overriding them. On the other hand, errors that are not
COR-classified are UNCOR-classified, which means program-flow has been affected. The
independence of the second classification from the error detection and correction techniques
makes COR and UNCOR the main guide to the effectiveness of ST.

The two-tier classification approach means eight combinations are possible (e.g.
UNDET/COR, DETCOR/UNCOR). They are however considered separately, which means
DETCOR, DET and UNDET only refer to the ST's ability to detect and correct errors, COR
and UNCOR only refer to the overall outcome of the error injection.

9.3.3 Estimation of error detection and correction rates
Depending on the conditions, PC writes are capable of overwriting errors, or ST may detect
and correct errors that are subsequently 'uncorrected' by PC writes. Therefore, the effect of
PC writes must be considered when estimating the error detection and correction rates.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Three sets of 'check' and 'store' operations take place during each clock-cycle (see Figure 9.1). Since two sets – these taking place between S1P1 and S2P1, and S4P1 and S5P1 – are the same, only the former is described.

Depending on the instruction-type being executed, the PC may or may not be read during PC increment sub-cycles, as shown in Figure 9.5. The periods where errors are uncorrectable (UNCOR) are highlighted in black. Errors occurring at the remaining periods will be detected and corrected by ST, or overridden by PC writes; hence these are classified as COR.

When the PC is read at S1P2, PC errors that happen after the CMP flag is read by ST, but before the PC is read by the processor, will go undetected. Under such circumstances, the processor will read, increment and write back the wrong PC value. Therefore, even though the ‘check’ operation at S1P2 would detect and correct the error, the PC would overwrite the corrected value. Such errors are classified as DETCOR since detection and correction occurred, but not as COR (program-flow errors occur).
Techniques intended to reduce the impact of program-flow errors on embedded systems

Errors that occur after ‘quan-cycle’ 3 of S1P2, but before PC writes, will not be detected. Such errors are therefore classified as UNDET. Due to PC writes however, these errors would be overwritten, preventing program-flow errors from occurring; hence, classified as COR. On the other hand, Errors that occur during S2P1 ‘quan-cycle’ 0 will be undetected and cause program-flow errors; hence, classified as UNDET and UNCOR.

If a PC read does not occur, a write will not occur either. Therefore, errors that occur after the CMP flag is read at S1P2, but before a ‘store’ takes place at S2P1, will be undetected and classified as UNDET and UNCOR.

Another situation illustrated in Figure 9.6, involves the PC write that may occur at S6P2 due to program-flow branches. If a PC write occurs at S6P2, errors that happen after the CMP flag is read by ST, but before the PC is written – would be overwritten. Such errors are undetectable by ST, but will not cause program-flow errors; hence, classified as UNDET and COR. Errors that happen after the PC write, but before a ‘store’, will cause program-flow errors and classified as UNDET and UNCOR.

On the other hand, if PC writes do not take place during S6P2, errors that occur after CMP flag reads will not be detected by ST, or overwritten by PC writes. These errors are classified as UNDET and UNCOR.
9.3.4 Quantifying the error detection and correction rates

Estimating ST’s effectiveness in combination with PC writes gives a more accurate picture of the overall error correction rate of ST. Therefore, only the estimation of COR-classified error rates is discussed.

For PC increment sub-cycles, PC errors are uncorrectable for two ‘quan-cycles’. Since two PC increments take place during each clock-cycle, four ‘quan-cycles’ would be UNCOR-classified. Turning to PC writes at S6P2, either one or two ‘quan-cycles’ are UNCOR-classified depending on whether PC write occurs. By assuming that program-flow branches happen less frequent than sequential execution (a ratio of 1:10 is a reasonable estimate), it is possible to say that each PC write contributes two UNCOR-classified ‘quan-cycle’.

The total UNCOR-classified ‘quan-cycles’ per clock-cycle would then be six, which is equivalent to 12.5% of the period of one clock-cycle. PC errors occurring at the remaining 87.5% of each clock-cycle would then be COR-classified. Note that this does not mean ST’s detection rate is 87.5%. Rather, it is the combined correction rate of ST and the PC write overriding errors.

9.4 Simulation of ST

‘Monte Carlo’-style simulations similar to those mentioned in Part Two were carried out on three test programs: ‘Alarm’, ‘Krider’ and ‘Prog’. Alarm and Prog were the same as Alarm_A and Prog_A that were used in Part Two. Krider, a ‘running LED’ display with a trail, was added into the list of test programs as it has a code coverage of 100%, without the need for any stimulus (c.f. Alarm requires external stimulus to achieve 100% code coverage). For each test program, 1,000 simulation cycles were carried out, each randomly injecting a PC error anywhere between the 1st and 1,000th clock-cycle (the ‘error window’). The simulation cycle range was much less than the previous simulations due to the time overhead incurred when implementing ST.

This method of emulating ‘real world’ MCU errors has been shown in [Asenek et. al. 1998, Velazco et. al. 2000] to correlate well with irradiation experiments. Although those experiments were carried out by bit-flipping the SFRs and internal memory, not just the PC, they gave credence to this technique of emulating processor upsets.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Unlike the other simulations carried out so far, the entire execution profile for each test program has to be inserted into the simulation script, which is necessary to determine if the processor still works correctly after error injection. These execution profiles are ‘clean runs’ (simulation runs without error injections) of each test program with the PC value listed for every clock-cycle. The execution profiles are generated with another script and pasted into the simulation script. Due to the ‘clean run’ execution profile, the simulation script size grows proportionately with the size of the ‘error window’.

Figure 9.7: ST error simulation flow-chart

Figure 9.7 shows the flow chart of each simulation run. Note: UNK-classified errors are not shown in this diagram.

The entire simulation is executed on the ‘quan-cycle’ basis, which is four times faster than the sub-cycle. To emulate clock quadrupling, the MCU is only ‘microstep’ executed during
Techniques intended to reduce the impact of program-flow errors on embedded systems

'quand-cycle' 0. Errors however, can be injected at any 'quand-cycle'. ST is also executed on the 'quand-cycle' basis; hence, it is perceived as running four times faster than the MCU.

As 8051Sim/8051Sim-NG do not cater for parallelism, the comparator is programmed to compare its inputs at 'quand-cycle' 2 of the 'check' operation cycle since processor execution and error injection (where applicable) takes place before ST. If the compared inputs are latched at 'quand-cycle' 3, it will be able to detect errors injected at 'quand-cycle' 3: this is not the case with the actual hardware. The modification to the comparator does not affect ST's operation other than the fact that errors occurring on or after 'quand-cycle' 2 during a 'check' operation will not be detected. The UNCOR-classified 'quand-cycles' would then increase by three, reducing the statistical COR-classified error rate to 81.25%.

The erroneous cycle, sub-cycle, 'quand-cycle' (EQC) and PC value are randomly generated before each simulation cycle starts. When execution reaches the error injection cycle, the PC is reverted to EPC, hence injecting the error. Three additional cycles are executed before the simulation script checks the ERROR flag to determine if the error has been detected. Errors that are detected are classified as DETCOR, while those undetected are classified as UNDET.

When the simulator is micro-stepping the processor, the error status of the simulator is also read to make sure no simulator errors occur. If an error does occur, the simulation cycle is classified as UNK. As previously mentioned, this classification takes precedence over DETCOR and UNDET errors.

After the UNK, UNDET and DETCOR classifications, the second stage is carried out to determine the overall effect of the injected errors. This provides a measure of ST's overall effectiveness.

The simulation is step-executed for another 10 clock-cycles. After each step-execution, the PC is compared with the value generated during the 'clean run'. Only when the comparison of all subsequent clock-cycles are the same, is the error classified as COR. Note: even when

---

48 This applies to S1P1, S4P1 and S6P1.
49 These errors are the result of 8051Sim/8051Sim-NG reaching unspecified conditions (e.g. execution of RETI instruction when not servicing an ISR). Trapping such errors prevents the simulator from halting.
Techniques intended to reduce the impact of program-flow errors on embedded systems

the additional 10 clock-cycles are the same as the “clean run’s”, this does not guarantee – but it is highly likely – that the injected error did not cause other problems.

Once these operations complete, another simulation cycle starts until the required 1000 simulation cycles for each program completes.

9.5 Results and discussion

Simulations are carried out on 8051Sim-NG with ST itself being part of the script. The framework of the simulation script was similar to the custom peripheral example discussed in Chapter 7. The script is located in Appendix O.

9.5.1 Simulation results

Simulation of each test program took approximately 200 minutes on a Pentium II 350MHz machine with 128MB RAM running Windows 2000. The simulation results are shown in Table 9.2. The difference between the total simulated errors and COR-classified ones are UNCOR-classified.

<table>
<thead>
<tr>
<th>Program</th>
<th>DETCOR</th>
<th>UNDET</th>
<th>UNK</th>
<th>COR</th>
<th>UNCOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>707</td>
<td>293</td>
<td>0</td>
<td>831</td>
<td>169</td>
</tr>
<tr>
<td>Krider</td>
<td>741</td>
<td>259</td>
<td>0</td>
<td>829</td>
<td>171</td>
</tr>
<tr>
<td>Prog</td>
<td>714</td>
<td>286</td>
<td>0</td>
<td>818</td>
<td>182</td>
</tr>
</tbody>
</table>

Table 9.2: Results of ST simulation

The results show that between 70.7% and 74.1% of the injected errors are detected and corrected (DETCOR) by ST. The remaining injected errors were not detected. Therefore, ST's error detection and correction rate is roughly 72%. No UNK-classified errors are detected throughout the simulation, which meant catastrophic simulator errors did not occur during any error-injection cycle.

What is more important though is the fact that on average, 82.6% of all errors are corrected when PC writes were taken into account. This proves that ST has a good overall error correction rate, and that the error correction estimation is accurate in predicting ST's overall effectiveness.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The minor difference between the COR-classified errors for the test programs is due to the relatively small number of simulations cycles carried out. The larger difference between DETCOR-classified errors is due its dependence on each program's dynamic instruction-type profile.

With these results, the combined detection rates for ST and PC writes is very promising, considering the fact that ST is a relatively simple technique. It is however, probable that other registers associated with program-flow may have been corrupted before the injected errors are corrected. At present, this situation cannot be simulated due to the lack of information regarding internal data transfers other than those related to the PC. Hence, it is disregarded in this experiment.

ST could also be implemented on registers other than the PC. If more than one register implements ST, it is unnecessary to duplicate the CREG for every implementation: this reduces the implementation overhead. It would be necessary to increase the number of signals from the CREG if the access times for other registers vary, however. The error flags from each implementation can also be ORed to form a bit that is read by the interrupt (or similar) system. If the important registers such as the Stack Pointer, Program Status Word and Accumulator implement ST, the number of detectable errors may increase, as well as the error detection rates.

9.5.2 Implementation

To estimate the implementation overhead, it is necessary to know the approximate gate count of the macro blocks that could be inferred during synthesis. The gate count for various macro blocks, when synthesized with Xilinx's XST, is shown in Table 9.3.

<table>
<thead>
<tr>
<th>Macro block</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>D flip-flop</td>
<td>8</td>
</tr>
<tr>
<td>16-bit 2-1 multiplexor</td>
<td>96</td>
</tr>
<tr>
<td>2-bit up counter</td>
<td>22</td>
</tr>
<tr>
<td>4-bit up counter</td>
<td>56</td>
</tr>
<tr>
<td>2-bit down counter</td>
<td>22</td>
</tr>
<tr>
<td>6-bit down counter</td>
<td>159</td>
</tr>
<tr>
<td>16-bit adder</td>
<td>96</td>
</tr>
<tr>
<td>16-bit comparator</td>
<td>72</td>
</tr>
<tr>
<td>16-bit comparator (+1 output)</td>
<td>240</td>
</tr>
</tbody>
</table>

Table 9.3: Approximate gate count for macro blocks synthesized with Xilinx's XST
Although ST is not implemented in hardware, it is reasonable to assume certain points about its implementation. Concentrating on the PC, a 2-to-1 16-bit multiplexer would be needed to handle two input sources: the internal address bus and the SPC. The SPC would comprise of 16 D-type flip-flops that hold PC’s value. The CREG would require a 4-bit up counter, a 2-bit up counter (QCY), two D-flip-flops for the chk and str signals, and some combinational logic. The control module itself is largely combinational, apart from three D-flip-flop for the ERROR, S-PC and PC-S signals. The comparator will be made from combinational logic.

<table>
<thead>
<tr>
<th>Module</th>
<th>Macro blocks</th>
<th>Quantity</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>16-bit 2-1 multiplexor</td>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>SPC</td>
<td>D flip-flop</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>CREG</td>
<td>D flip-flop</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4-bit up counter</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>2-bit up counter</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>COMP</td>
<td>16-bit comparator</td>
<td>1</td>
<td>72</td>
</tr>
<tr>
<td>CTRL</td>
<td>D flip-flop</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>Misc.</td>
<td>logic gates</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>514</td>
</tr>
</tbody>
</table>

Table 9.4: ST’s gate count estimate

With the approximate gate count shown in Table 9.3, ST’s estimated implementation is shown in Table 9.4. Note: 100 gates have been added to account for the combinational logic circuits such as CREG’s reset and CTRL’s logic.

Compared with TE-51’s – a generic FPGA (field programmable gate array) synthesizable 8051 MCU (described in Part Four) – 29,069 gate count, ST’s overhead is approximately 1.76%.

9.5.3 Evaluation of ST

ST is a chip-based designed with minimal impact to the processor die: this in turn equates to negligible increase in component cost. Its implementation overhead is also over an order of magnitude less than SM, MR and PM; and it detects and corrects errors.

ST is capable of detecting and correcting 83% of PC corruption with very short latencies (detected errors are corrected within 4 sub-cycles), and does so without requiring any processor resources, unlike SM and WP. Since ST is transparent to the processor core (and even to the PC), this means programs written on a device without ST, can be ported to one
Techniques intended to reduce the impact of program-flow errors on embedded systems

with ST without any code changes, or even recompilation. Hence, no modifications are also necessary to programming tools, unlike SM and WP.

ST does have a few drawbacks. For a start, ST only tracks the PC and is unable to detect program-flow errors that were caused by the corruption of other registers (e.g. Stack Pointer) or the address bus. In addition, since ST does not work across the clock-cycle boundary: if an error is not detected by ST, it will never be. On the other hand, this means it can handle interrupts without any modifications.

9.6 Conclusion
When combined with the effects of PC writes that overwrite corrupted values, ST has been shown in simulations to detect and correct over 83% of PC corruption. ST also detects and corrects errors in real-time, does not require program code modifications, and does not use any MCU resources. ST's implementation overhead has also been shown to be minimal. On the other hand, ST has a few drawbacks such as the fact that only PC corruption is detectable. Nevertheless, ST serves as the foundation for other improved versions, as discussed in the following chapters.
State Tracking 2: Improving detection and correction rates

State Tracking is capable of achieving an overall error correction rate (PC writes taken into account) of 83%. Although this is a good achievement for such a simple mechanism, it still means one in six errors will never be detected. Moreover, only errors directly corrupting the PC are detectable. State Tracking 2 (ST2) seeks to improve on ST's error detection and correct rates.

Since ST2 is in many ways similar to ST, some of its description will refer back to the relevant sections of the previous chapter. This chapter starts with the functional description of ST2, followed by the estimation of its error correction rate, simulations, discussion and evaluation.

10.1 Description of State Tracking 2

ST's relatively low error detection rate is due to its inability to detect PC corruption occurring in the two 'quan-cycles' during which each PC increment may occur. This is due to its 'store' operation that does not check the value of the PC when the SPC is updated. ST2 rectifies this major drawback by implementing a more complex 'Compare' module and introducing another operation.

In ST2, the 'store' operations are replaced by 'checkstore' operations at PC increment sub-cycles (S2P1 and S5P1). This is actually a 'check' operation that is possibly followed by a 'store' operation. 'Checkstore' operations will perform the 'store' if the PC and the SPC do not differ by more than one. On the other hand, an error will be flagged if the PC and the SPC differ by more than one.

A 'store' operation is still carried out at the PC write sub-cycle of S6P2 since it is not possible to know the PC's value for branching instructions without monitoring other registers (e.g. PSW). Hence, the undetectable 'quan-cycles' at S6P2 remains the same as ST's. ST2's block diagram is shown in Figure 10.1.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 10.1: Block diagram for State Tracking 2

Three of ST's modules have been modified to accommodate the changes. The CREG now outputs an additional flag, **CHKSTR**, that goes high during 'checkstore' cycles, as shown in Figure 10.1. The 'Comparator' also outputs two signals, **CMP0** and **CMP1**, that will be respectively set when the PC and the SPC are the same, and when the PC is one (in value) more than the SPC. If the SPC's value is two or more than the PC's, both outputs will be low. Hence, as long as **CMP0** and **CMP1** have equal states, it signifies an error. Accordingly, the 'Control' module has been modified to accommodate both inputs.

Figure 10.2: ST2 operation cycles

ST2's operation cycles for each clock-cycle are shown in Figure 10.2. The difference between this and ST's operation cycles is the lack of 'checks' during S1P2 and S4P2, and two 'checkstores' replacing the 'stores' at S2P1 and S5P1. Figure 10.3 shows the 'checkstore' operation at the 'quan-cycle' level.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 10.3: 'Checkstore' operation at 'quan-cycle' level

Either the PC-S or ERROR signal goes high at 'quan-cycle' 1 during a 'checkstore' operation. If COMP0 and COMP1 are at the same level, the ERROR signal goes high. On the other hand, PC-S will instead be held high for one 'quan-cycle' if COMP0 and COMP1 are different. The PC-S signal can thus be generated by ANDing the XORed COMP's outputs with the 'quan-cycle' and the CHKSTR signal. The ERROR signal will be clocked and held high on the rising edge of 'quan-cycle' 1. As with ST, this signal can be used to trigger an appropriate interrupt.

'Checkstore' operations cannot correct errors as ST2 has no knowledge of the instruction-type being executed. Without this knowledge, it is impossible to know if the PC should be corrected with the original value in SPC, or an incremented value.

10.2 Estimation of error detection and correction rates

As with ST, the combined ST2 and PC write error correction (COR) rate is more important than its error detection and/or correction rate. A new category, DET, is needed for ST2 since 'checkstore' operations can only detect errors.

Since the situation for PC writes is the same as ST's, only PC increment sub-cycles are discussed here. ST2's uncorrectable period during PC increment and non-increment sub-cycles are shown in Figure 10.4. The bars highlighted in black are the periods where errors will not be successfully corrected (UNCOR-classified). All other periods are COR-classified.
Techniques intended to reduce the impact of program-flow errors on embedded systems

During PC increment sub-cycles, errors that happen between PC-SPC compares and PC reads will be detectable by the ‘checkstore’ operation at S2P1. This however, will have allowed the processor to read the corrupted value, increment, and write it back at the start of S2P1.

Although the ‘checkstore’ is almost certain to detect the error, its inability to correct it may lead to program-flow errors. Such errors will be DET- and UNCOR-classified.

Errors occurring between the PC read and PC write will automatically be overridden by the PC write, hence such errors are UNDET- and COR-classified.

When PC increments do not take place, errors occurring between CMP and CHKSTR will not cause program-flow errors. At the ‘checkstore’ cycle, these errors will be detected, preventing the ‘store’ from taking place. The error will then be rectified at the subsequent

---

50 The following check will make PC = SPC, which means the same byte may be read twice.
Techniques intended to reduce the impact of program-flow errors on embedded systems

'check' cycle. Therefore, such errors are classified as DETCOR (due to the recovery at the following 'check' cycle) and COR.

Note: only one situation exist where PC errors is undetected by 'checkstore' operations: the corrupted PC value is equivalent to, or one more than SPC's. Since the probability of this situation occurring is minute, it is not considered here.

The combined ST2 and PC write error detection rate will be almost 91.7% (44 out of 48 'quan-cycles'), assuming two PC increments per clock-cycle and program-flow branches do not occur. Table 10.1 shows the theoretical correction rates for all non-branching instruction-types. The theoretical correction rates for 112, 122 and 132 instruction-types when branching occur are 95.8%, 94.8% and 93.8% respectively.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Sub-cycle</th>
<th>Instruction-type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>S1P2</td>
<td>Inc.</td>
</tr>
<tr>
<td></td>
<td>S4P2</td>
<td>No Inc.</td>
</tr>
<tr>
<td>2</td>
<td>S1P2</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>S4P2</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>S1P2</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>S4P2</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>S1P2</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>S4P2</td>
<td>X</td>
</tr>
<tr>
<td>Instruction length</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>‘Quan-cycles’ per instruction</td>
<td>48</td>
<td>96</td>
</tr>
<tr>
<td>UNCOR 'quan-cycles'</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>COR 'quan-cycles' (%)</td>
<td>93.8</td>
<td>94.8</td>
</tr>
</tbody>
</table>

Table 10.1: ST2 correction rates of non-branching instruction-types

From the table, 'Inc.' and 'No Inc.' stand for 'increment' and 'no increment' respectively during the PC increment sub-cycles. The former contributes one UNCOR 'quan-cycles'. An 'X' means the particular increment cycle does not apply to the referred instruction-type.
Techniques intended to reduce the impact of program-flow errors on embedded systems

10.3 Simulation of ST2
The simulation procedure for ST2 is the same as ST's (see Chapter 9). ST2’s simulation script is almost identical to ST’s. Another variable is added to the simulation script to signify when non-correctable errors are detected. If this variable (and the error flag) is set, the injected error is classified as DET. On the other hand, if only the error flag is set, the injected error will be detected and correctable; hence classified as DETCOR.

10.4 Results and discussion
On average, each simulation took approximately 4 hours to complete on a Pentium II 350MHz PC with 128MB RAM running Windows 2000. The longer simulation time (when compared to ST’s) is due to the more complicated structure of ST2. The simulation script is listed in Appendix O.

The results obtained from the simulation are shown in Table 10.2.

<table>
<thead>
<tr>
<th>Program</th>
<th>DETCOR</th>
<th>DET</th>
<th>UNDET</th>
<th>UNK</th>
<th>COR</th>
<th>UNCOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>739</td>
<td>167</td>
<td>94</td>
<td>0</td>
<td>940</td>
<td>60</td>
</tr>
<tr>
<td>Krider</td>
<td>738</td>
<td>160</td>
<td>102</td>
<td>0</td>
<td>926</td>
<td>74</td>
</tr>
<tr>
<td>Prog</td>
<td>739</td>
<td>151</td>
<td>110</td>
<td>0</td>
<td>924</td>
<td>76</td>
</tr>
</tbody>
</table>

Table 10.2: Results of ST2 simulation

From the simulated results, close to 74% of injected errors were detectable and correctable by ST2. This made ST2’s average recorded DETCOR slightly higher than ST’s. ST2’s overall error detection rate (DET + DETCOR) is about 17% greater than ST’s, however. This means ST2 is capable of detecting 17% more errors than ST, and can correct roughly as many.

As with ST, the important parameter to compare is COR. ST2 – in combination with PC writes – is able to correct more than 92% of injected errors. This is a significant increase in COR rate over ST’s. ST2’s COR results are also within the ranges estimated in the previous section. The differences between test programs are attributed to the relatively small number of samples and the differences in their dynamic instruction-type profile.

Due to the nature of ‘checkstore’ operations, PC values that differ by two or more from SPC’s will be detected.
Techniques intended to reduce the impact of program-flow errors on embedded systems

10.4.1 Implementation
ST2’s implementation overhead would be higher than ST’s mainly due to the COMP module that requires two outputs. Its estimated gate count, based on the approximate gate count for each macro block (see Table 9.3), is shown in Table 10.3.

<table>
<thead>
<tr>
<th>Module</th>
<th>Macro blocks</th>
<th>Quantity</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>16-bit 2-1 multiplexor</td>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>SPC</td>
<td>D flip-flop</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>CREG</td>
<td>D flip-flop</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>4-bit up counter</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>2-bit up counter</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>COMP</td>
<td>16-bit comparator (+1)</td>
<td>1</td>
<td>240</td>
</tr>
<tr>
<td>CTRL</td>
<td>D flip-flop</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>Misc.</td>
<td>logic gates</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>740</td>
</tr>
</tbody>
</table>

Table 10.3: ST2’s gate count estimate

The additional logic gates allocated for ST2 is higher that ST’s due to the additional ‘checkstore’ operation. This operation will increase CREG and CTRL’s gate count. Based on ST2’s estimated gate count, it is approximately 44% higher than ST’s, and is about 2.55% of TE-51’s implementation (see Chapter 14).

10.4.2 Evaluating ST2
Since ST2 and ST are similar in many ways, they share many advantages and disadvantages as discussed in the previous chapter. The main issues that differentiate ST2 from ST are its higher error detection rates (DET) and COR rates, and its marginally higher implementation overhead (relative to TE-51). ST2’s larger implementation size may also mean it is more likely to be corruptible due to its larger cross sectional area [Asenek et. al. 1998]51. Its higher detection rate however, should offset this drawback.

---

51 ST and ST2 have similar duty cycle (period where registers hold active data).
10.5 Conclusion

ST2, which is principally based on ST, has been shown – through calculations and simulations – to have a higher overall error detection rate (83% and 93% for ST and ST2 respectively) than ST. Its minor drawback, from ST’s point of view, is the marginally higher implementation overhead. In any case, this drawback is masked by ST2’s higher error detection rate, which generally makes it a better choice than ST.
State Tracking, which was described in Chapter 9, was shown to be relatively effective in detecting and correcting PC errors. ST had a few shortcomings, however: one of which – due to its ‘store’ operations – was addressed by ST2. With ‘checkstore’ replacing ‘store’ operations, PC errors during PC increment sub-cycles were detectable by ST2.

However, like ST, ST2 has another major disadvantage: the same operations are performed regardless of the instruction-type. This leads to unnecessary operations in many instruction-types, which has an impact on ST and ST2’s error detection rates. The root of this problem lies in the fact that ST and ST2 do not have any knowledge of the instruction-type being executed; as a result, they cannot tailor the operations to different instructions types. An improvement to ST2 – known as Byte Cycle Tracking (BCT) – aims to address this issue by making it aware of the various instruction-types.

This chapter starts with a functional description of BCT, followed by the estimation of the error detection and correction rates, simulations, discussion of results and conclusion.

11.1 Functional description of BCT

BCT implements additional registers to keep track of the instruction-type that is being executed, and tailors the operations to suit them. To allow errors to be correctable during PC increments, an additional operation: ‘SPCinc’, is introduced. This operation increments the SPC. Three additional modules – the Byte Register (BREG), the SPC incrementer (INC) and the Instruction Decoder (IDEC) – are also necessary to derive the byte size and execution cycle information of each instruction. The CREG, SPC and ‘Control’ module have also been updated to handle the extra conditions, as shown in Figure 11.1.

The main obstacle when dealing with the 8051 instruction set is that there are seven different byte-cycle combinations. Instructions are classified by the IDEC, which decodes each instruction’s size (bytes) and execution time (cycles) based on the first operand (the first byte
Techniques intended to reduce the impact of program-flow errors on embedded systems

uniquely identifies all 8051 instructions). An instruction's size and execution time are stored in the BREG and the CREG respectively.

Unlike ST and ST2, BCT's CREG and BREG decreases its value. Decrementing allows 'check', 'SPCinc', 'checkstore' and 'store' operations to take place at the same CREG and BREG value for all instruction types.

**CREG\_val** and **BREG\_val** – determined by the IDEC – are based on the first byte of an instruction. This value can be derived from the Instruction Register, or the instruction buffer\(^{52}\). For example, if the current instruction is of type 132, **CREG\_val** and **BREG\_val** will be 24 and 3 respectively. If, and only if, the **INT\_LCALL** flag is set\(^{53}\), will the IDEC set **BREG\_val** and **CREG\_val** to 0 and 24.

The BREG, a 2-bit register, decrements during 'checkstore' cycles until it reaches zero. Its associated flag, **BREG\_en**, is set whenever the BREG is not zero. If a 'store' operation is detected (**STR** is high), the BREG will be reloaded with **BREG\_val**.

The CREG is still the main register that outputs the signals necessary to synchronise BCT.

At the heart of the CREG is a 6-bit counter. This counter decrements on every sub-cycle until

---

\(^{52}\) TE-51 – discussed in Part Four – buffers the entire instruction internally.

\(^{53}\) This flag is set when the processor is generating an LCALL instruction to vector to the ISR,
Techniques intended to reduce the impact of program-flow errors on embedded systems

it reaches zero, where it will be reloaded with the value of \texttt{CREG\_val} during 'store' operations. On specific CREG values, the \texttt{CHK}, \texttt{CHKSTR}, \texttt{STR} and \texttt{INC} signals are set. These signals are only set when \texttt{BREG\_en} is set since PC increment only takes place as long as there are instruction bytes left to read. When CREG's value is one however, \texttt{CHK} will unconditionally be taken high as a 'store' takes place at S6P2. This part of the operation is the same as that in ST and ST2: BCT cannot distinguish between branching and non-branching instructions. Hence, it has to perform a 'store', just in case a PC write occurs.

Two examples of the operations carried out for \texttt{I11} and \texttt{I22}-type instructions are shown in Figure 11.2 and Figure 11.3. These operations are governed by the signals controlled by CREG.

![Figure 11.2: BCT's operations during I11-type instructions](image)

![Figure 11.3: BCT's operations during I22-type instructions](image)
Techniques intended to reduce the impact of program-flow errors on embedded systems

The "Comparator's" design is the same as that used in ST: only an equal comparison between the PC and SPC is required.

The SPC has an additional control input, inc, which goes high whenever it has to be incremented. The SPC's incrementer, a 16-bit adder, presents a 16-bit incremented value of its input, at its output. When inc goes high, SPC will be loaded with the output of the INC, hence incrementing the SPC.

The 'Control' module's functionality remains almost the same as that in ST2. The only difference is that spc will be taken high for one 'quau-cycle' if PC corruption is detected during PC increment sub-cycles. When such situations occur, the PC will be loaded with SPC's value; thereby correcting the error. This was not the case for ST2, where the lack of instruction-type information meant that it was unclear whether an increment should or should not occur. Therefore, ST2 could only detect errors in this situation.

Although branching instructions are only of type I12, I22, and I32, the 'check' and 'store' operations have to be carried out for all instruction-types even when PC writes at S6P2 are guaranteed not to occur (I11-, I14-, I21-type instructions). If these operations are not included for non-branching instructions, errors that happen after the last 'checkstore' of an instruction will go undetected until the first 'check' of the next instruction. However, since the PC is used to derive the BREG and the CREG's values at S6P2, corruption of the PC at that stage may cause the wrong values to be loaded into BREG and CREG - leading to wrong error diagnosis and correction. Hence, the 'check' operation at S6P1 is vital for all instruction-types. Note: if the 8051 processor is capable of prefetching instructions, it should be possible to derive the BREG and the CREG's value from the prefetch buffer.

The ERROR flag is set whenever errors are detected since there is a chance that the SPC (or other parts of BCT) may be corrupted.

11.2 Estimation of error detection and correction rates

BCT's COR-classified rates will be the same as ST2's during PC increment sub-cycles (see Figure 10.4). When PC increments occur, only errors that happen after the PC-SPC comparison is carried out (but before the PC read) will cause the processor to read the erroneous value. This value will then be incremented and written back at the PC write sub-
Techniques intended to reduce the impact of program-flow errors on embedded systems

cycle. Although BCT is able to correct the error during ‘checkstore’ operations (assuming the erroneously incremented PC differs by two or more from SPC), correction only occurs after the erroneous value is read by the processor.

If the PC increments were not carried out during the relevant sub-cycles, ‘check’ and ‘checkstore’ operations would not be carried out either. Hence, the corrupted PC would not be read by the processor, but would be detected by the ‘check’ operation located before the PC write at S6P1 of the last cycle of the instruction.

Unlike ST and ST2, ‘check’ and ‘store’ operations for PC writes at S6P2 only take place when the processor is executing the last clock-cycle of the current instruction (see Figure 11.2). When ‘check’ and ‘store’ operations do take place, their contribution to uncorrectable ‘quan-cycles’ is the same as that in ST and ST2. Therefore, two ‘quan-cycles’ per instruction are uncorrectable when program-flow branches do not occur, one when they do.

As BREG_val and CREG_val are dependent on the PC, corruption of the PC may alter their values and prevent BCT from working correctly: this factor has to be taken into account when estimating BCT’s error correction rate. BREG_val and CREG_val however, are read at ‘quan-cycle’ 1 of the ‘store’ operation. Hence, their ‘vulnerable’ period overlaps with the ‘store’ operation.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Quan-cycles</th>
<th>Branching</th>
<th>Non-Branching</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNCOR</td>
<td>COR</td>
<td>UNCOR</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>%</td>
<td>#</td>
</tr>
<tr>
<td>I11</td>
<td>48</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I12</td>
<td>96</td>
<td>2</td>
<td>2.1</td>
</tr>
<tr>
<td>I14</td>
<td>192</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I21</td>
<td>48</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I22</td>
<td>96</td>
<td>3</td>
<td>3.1</td>
</tr>
<tr>
<td>I32</td>
<td>96</td>
<td>4</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Table 11.1: COR-classified ‘quan-cycles’ for various instruction-types

Note that it is per instruction, not per clock-cycle.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The combined error correction rate for BCT and PC writes (COR rates) for all instruction-types are summarised in Table 11.1. As expected, I21-type instructions have the lowest COR rates since two reads occur in a single clock-cycle. On the other hand, I14-type instructions have the highest: only one PC increment takes place during four clock-cycles.

11.3 Simulation of BCT
The simulation procedure for BCT is the same as for ST2 apart from replacing ST2’s definition with BCT’s. As before, 8051Sim-NG was used in the simulation.

11.4 Results and discussion
On average, each simulation took approximately 4 hours to complete on a Pentium II 350MHz PC with 128MB RAM running Windows 2000. The simulation script is located in Appendix O.

The results obtained from the simulation are shown in Table 11.2.

<table>
<thead>
<tr>
<th>Program</th>
<th>DETCOR</th>
<th>UNDET</th>
<th>UNK</th>
<th>COR</th>
<th>UNCOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>811</td>
<td>189</td>
<td>0</td>
<td>985</td>
<td>15</td>
</tr>
<tr>
<td>Krider</td>
<td>773</td>
<td>227</td>
<td>0</td>
<td>979</td>
<td>21</td>
</tr>
<tr>
<td>Prog</td>
<td>806</td>
<td>194</td>
<td>0</td>
<td>983</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 11.2: Results of BCT simulation

The simulation results showed that over 77% of errors were detected and corrected (DETCOR-classified) by BCT. Its COR-classified error rate – the real gauge of its effectiveness – of 98.2% means its overall error detection rate is approximately 5% higher than ST2, and a great deal more than ST’s. Differences between the estimated and simulated results are mainly attributed to the small number of simulation cycles.

The remaining injected errors were undetectable, while no simulator errors were encountered. UNDET-classified errors were more than those recorded for ST2. However, it cannot be concluded that BCT had a higher UNDET rate than ST2. BCT is able to correct errors even if PC writes will not overwrite them during PC increment sub-cycles, although this situation cannot occur. This is not the case for ST2.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Like ST and ST2, no UNK-classified errors were recorded, which means the simulator did not encounter any exceptions.

11.4.1 Implementation

BCT is the most complex of the three hardware-based error detection and correction techniques presented in Part Three. As with others, the implementation size is estimated based on Table 9.3.

The CREG’s main register would involve a 6-bit down counter. It will also require 4 D-flip-flops for the operation signals, and a 2-bit up counter for the QCY signal. The BREG will be a 2-bit down counter. The SPC’s incrementer will be a 16-bit adder. The ‘Instruction Decoder’ can be implemented entirely with combinational logic. Based on a test synthesis for the IDEC only, 413 gates were required to classify all the 8051 instruction-types.

BCT’s implementation estimate is shown in Table 11.3.

<table>
<thead>
<tr>
<th>Module</th>
<th>Macro blocks</th>
<th>Quantity</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>16-bit 2-1 multiplexor</td>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>SPC</td>
<td>D flip-flop</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>INC</td>
<td>16-bit adder</td>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>CREG</td>
<td>D flip-flop</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>6-bit down counter</td>
<td>1</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>2-bit up counter</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>BREG</td>
<td>2-bit down counter</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>COMP</td>
<td>16-bit comparator</td>
<td>1</td>
<td>72</td>
</tr>
<tr>
<td>CTRL</td>
<td>D flip-flop</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>IDEC</td>
<td>logic gates</td>
<td>413</td>
<td>413</td>
</tr>
<tr>
<td>Misc.</td>
<td>logic gates</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>1206</td>
</tr>
</tbody>
</table>

Table 11.3: BCT’s gate count estimate

BCT’s gate count estimated is approximately 63% larger than ST2, and is about 4.1% of TE-51’s size (see Part Four).
11.4.2 Evaluating BCT
Apart from its higher implementation overhead (which equates to a higher production cost) and overall error correction rates, BCT has similar characteristics with ST2. BCT’s higher implementation overhead will mean the possibility of it being corrupted is also higher than ST or ST2. However, its higher error correction rates should overcome this drawback.

11.5 Conclusion
BCT has been shown to detect and correct over 98% of errors (PC writes considered), which is about 5% higher than ST2. BCT is also capable of correcting all errors during PC increments, unlike ST and ST2.

BCT’s complexity mean that it requires approximately 63% more logic gates than ST2, which will increase its implementation size. Coupled with the fact that it does require tracking of more processor flags and registers, BCT can be less robust than ST2. On the other hand, since more processor states are monitored, BCT may detect other errors, such as sequencing errors. This can be seen as an indirect benefit of BCT.
In the previous chapters, ST, ST2 and BCT have been shown to detect and correct between 72% and 79.7% (test program average) of PC errors. When the effect of PC writes overwriting PC errors were taken into account, their average error correction rate is between 82.6%, and 98.2%.

In this chapter, ST, ST2 and BCT are evaluated against each other, the software-based techniques described in Part One, and the chip-based techniques described in Chapter 8.

12.1 Comparison of ST, ST2 and BCT

When the phenomenon of PC writes overriding errors are taken into account, all errors – except those occurring between S5P1 ‘quan-cycle’ 3 and S6P1 ‘quan-cycle’ 1 – are detectable. A majority of these errors are also correctable. In short, ST2 has a higher error detection rate than ST, as shown in Table 12.1.

<table>
<thead>
<tr>
<th>Technique</th>
<th>DETCOR (%)</th>
<th>DET (%)</th>
<th>UNDET (%)</th>
<th>COR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>72.1</td>
<td>0.00</td>
<td>27.9</td>
<td>82.6</td>
</tr>
<tr>
<td>ST2</td>
<td>73.9</td>
<td>10.2</td>
<td>15.9</td>
<td>93.0</td>
</tr>
<tr>
<td>BCT</td>
<td>79.7</td>
<td>0.00</td>
<td>20.3</td>
<td>98.2</td>
</tr>
</tbody>
</table>

Table 12.1: Summary of ST’s, ST2’s and BCT’s error classification (average)

BCT’s error correction rate is the highest amongst the three techniques as it has the knowledge of the instruction-type that is being executed. This allows BCT to tailor its operations to take place only when PC increments occur, which reduces the period where errors are undetectable. BCT can also correct all the errors that are (only) detected by ST2.

Because it has instruction-type information, BCT has the potential to correct PC write errors during sub-cycle S6P2, if the program-flow vectoring address and conditions are known. This can further increase BCT’s error correction rate and broaden the type of errors that it can
Techniques intended to reduce the impact of program-flow errors on embedded systems

detect. Unfortunately, the variety of branching instructions in the 8051 instruction set make such endeavours impractical\(^5\).\(^5\)

The error correction rates increase with the estimated number of logic gates necessary to implement ST, ST2 and BCT, as shown in Table 12.2. This is not surprising: the higher error correction rates are due to increases in the complexity of the techniques. Also shown in Table 12.2 is each implementation’s size relative (in %) to each other, normalised against ST, and to the 29,069 gate count for TE-51 (see Part Four).

<table>
<thead>
<tr>
<th>Technique</th>
<th>DETCOR (%)</th>
<th>COR (%)</th>
<th>Gate count</th>
<th>Rel. Overhead</th>
<th>% of TE-51</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>72.07</td>
<td>82.60</td>
<td>514</td>
<td>1.00</td>
<td>1.77</td>
</tr>
<tr>
<td>ST2</td>
<td>73.87</td>
<td>93.00</td>
<td>740</td>
<td>1.44</td>
<td>2.55</td>
</tr>
<tr>
<td>BCT</td>
<td>79.67</td>
<td>98.23</td>
<td>1206</td>
<td>2.35</td>
<td>4.15</td>
</tr>
</tbody>
</table>

Table 12.2: Summary of ST, ST2 and BCT

Based on Table 12.2, ST2’s COR rate increased by 12.6% over ST’s with an implementation size increase of 44%. BCT’s COR rate increased by 5.6% over ST2’s while its implementation size increased by 63%. Hence, ST2 seems to be the best compromise between the three techniques in terms of COR rate and implementation size.

A synthesizable version of ST2, written in VHDL, is described in the next section.

12.2 Implementation of State Tracking 2

ST2 was written in VHDL in order to demonstrate that it was readily synthesizable (i.e. could be implemented in hardware), and – assuming it was – to determine the implementation overhead. Aldec’s Active-HDL 4.2 and Xilinx’s XST were the design entry and synthesis tool used respectively for the implementation.

The Spartan-II family of field-programmable gate arrays from Xilinx was selected as the target implementation device (see Appendix K for an overview of FPGAs). This FPGA family was chosen since it has features such as block RAM and Delay-Locked Loops, which

\(^5\) Two SFRs and all bit-addressable RAM locations would need to be monitored.
Techniques intended to reduce the impact of program-flow errors on embedded systems

allow greater design flexibility, especially for processor-like implementations. Powerful development tools (WebPack ISE) for this family are also available free of charge.

### 12.2.1 VHDL description of ST2

The source for the important modules – PC, CREG and CTRL – is listed here. All the source files are listed in Appendix Q while the entire project (Active-HDL format) can be found on the CD-ROM (see Appendix R). Please refer to Figure 10.1 for a better understanding of the following code listings.

```vhdl
entity PC is
  port (    rst: in STD_LOGIC; -- reset
            clk: in STD_LOGIC; -- clock
            PCi: in STD_LOGIC_VECTOR (15 downto 0); -- PC input value (from bus)
            PCo: out STD_LOGIC_VECTOR (15 downto 0); -- PC output value (to bus)
            PCWRi: in STD_LOGIC; -- signal to write PC
            SPCo: out STD_LOGIC_VECTOR (15 downto 0); -- to SPC
            SPCI: in STD_LOGIC_VECTOR (15 downto 0); -- from SPC
            S_PCi: in STD_LOGIC; -- signal to load PC with SPC
            INJi: in STD_LOGIC; -- signal to inject error
            INJvi: in STD_LOGIC_VECTOR (15 downto 0) -- error injection value
        );
end PC;

architecture bhv of PC is
  signal PCreg: STD_LOGIC_VECTOR(15 downto 0)
  signal PCval: STD_LOGIC_VECTOR(15 downto 0)
  signal reset: STD_LOGIC;
begin
  process(reset, clk, PCWRi)
  begin
    if reset = '1' then
      PCreg <= PCval;
        elsif rising_edge(clk) then
          if PCWRi = '1' then
            PCreg <= PCi;
          end if;
    end if;
  end process;

  reset <= rst or S_PCi or INJI;
  PCval <= x"0000" when rst = '1' else
              SPCI when S_PCi = '1' else
              INJvi when INJi = '1' else
                (others => '-');

  PCo <= PCreg;
  SPCo <= PCreg;
end bhv;
```

Listing 12.1: VHDL source code for modified PC implementing ST2

To allow asynchronous loading of the PC and error injection, the PC – based on TE-51’s (see Part Four) implementation – has to be modified. The system reset (rst), load PC with SPC (S-PC) and corrupt PC (INJI) signals are ORed to form the PC’s reset signal. These signals also determine the source for the PC: 0x0000 when a system reset occurs, the value of SPC
Techniques intended to reduce the impact of program-flow errors on embedded systems

when $s$-PC is set, or $INJ_i$, the value to corrupt the PC. The synchronous portion of the PC description remains unaltered.

CREG's VHDL code is shown in Listing 12.2. Its main counter (CREGreg) decrements on each sub-cycle. QCYreg – the ‘quan-cycle’ counter – decrements on each ‘quan-cycle’. At the relevant ‘quan-cycle’/sub-cycle, the appropriate control signals are generated.

```vhdl
entity CREG is
port (
    rst: in STD_LOGIC; -- reset
    clk4: in STD_LOGIC; -- clock (four times MCU clock)
    CHKo: out STD_LOGIC; -- CHK output signal
    STRO: out STD_LOGIC; -- STR output signal
    CHKSTRo: out STD_LOGIC; -- CHKSTR output signal
    QCYo: out STD_LOGIC_VECTOR(1 downto 0) -- QCY output
);
end CREG;
architecture bhv of CREG is
begin
    process(rst, clk4)
    begin
        if rst = '1' then
            CREGreg <= "0000";
        elsif rising_edge(clk4) then
            if QCYreg = 3 then
                QCYreg <= "00";
            elsif CREGreg = 11 then -- checking CREG for overflow
                CREGreg <= "0000"; -- CREG overflow, reset
            else
                CREGreg <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(CREGreg) + 1), 4);
            end if;
        end if;
    end if;
    end process;
    CHK <= '1' when CREGreg = "0000" else '0';
    STR <= '1' when CREGreg = "1011" else '0';
    CHKSTR <= '1' when CREGreg = "0010" else '0';
    QCY123 <= '0' when QCYreg = "00" else '1';
    QCY012 <= '0' when QCYreg = "01" else '1';
    CHKo <= CHK and QCY123;
    CHKSTRo <= CHKSTR and QCY012;
    STRO <= STR and QCY012;
    QCYo <= QCYreg;
end bhv;
```

Listing 12.2: VHDL source code for ST2's CREG module

146
Techniques intended to reduce the impact of program-flow errors on embedded systems

The VHDL source code for the Control module is shown in Listing 12.3. As shown in Figure 10.1, CTRL’s output signals (S-PC or PC-S) are based on CREG’s signals (CHK, STR, CHKSTR). If an error is detected, CTRL sets the ERROR signal.

```vhdl
entity CTRL is
  port (
    rst: in STD_LOGIC; -- reset
    clk4: in STD_LOGIC; -- clock (four times MCU clock)
    CHKi: in STD_LOGIC; -- control signal inputs
    STRi: in STD_LOGIC;
    QCYi: in STD_LOGIC;
    CMP0i: in STD_LOGIC; -- COMP inputs
    CMP1i: in STD_LOGIC;
    S_PCo: out STD_LOGIC; -- PC and SPC signal outputs
    PC_So: out STD_LOGIC;
    ERRORo: out STD_LOGIC -- error signal
  );
end CTRL;

architecture bhv of CTRL is
  signal QCYO: STD_LOGIC;
  signal QCY1: STD_LOGIC;
  signal QCY3: STD_LOGIC;
  signal ERRO: STD_LOGIC;
  signal ERR1: STD_LOGIC;
  signal ERRCHK: STD_LOGIC;
  signal ERROR: STD_LOGIC := '0';
begin
  process (rst, clk4)
  begin
    if rst = '1' then -- resetting error flag
      ERRORo <= '0';
    elsif rising_edge(clk4) then -- setting error flag
      ERRORo <= ERROR;
    end if;
  end process;
  QCY0 <= '1' when QCYi = "00" else '0';
  QCY1 <= '1' when QCYi = "01" else '0';
  QCY3 <= '1' when QCYi = "11" else '0';
  ERRO <= (not CMP0i) or CMP1i; -- if not equal
  ERR1 <= CMP0i xor CMP1i; -- if not equal and diff. by 1
  ERRCHK <= CHKi and QCY3 and ERRO;
  S_PCo <= ERRCHK;
  PC_So <= ((STRi and QCY1) or (CHKSTRi and QCY1 and (not ERR1)));
  ERROR <= '1' when ERRCHK = '1' else '0' when ((CHKSTRi = '1') and (QCY1 = '1') and (ERR1 = '1')) else ERROR;
end bhv;
```

Listing 12.3: VHDL source for ST2's CTRL module

12.2.2 Behavioural simulation of ST2

ST2 was simulated with Active-HDL with the generic timing constraints of the Xilinx’s XC2S200 Spartan-II FPGA. The relevant portions of the waveforms of various conditions
Techniques intended to reduce the impact of program-flow errors on embedded systems

are shown in the following timing diagrams (Figure 12.1, Figure 12.2, Figure 12.3 and Figure 12.4). The signal states and register values are shown on the left.

In Figure 12.1, the PC error (value of 0x1234) injected during S4P2 (CREG = 7) happened when a PC increment did not occur. This error was detected by the 'checkstore' operating at S5P1 and the ERROR flag was set. As mentioned earlier, 'checkstore' was not capable of correcting the error; as a result, the PC was not corrected. However, the PC is corrected after 'quan-cycle' 3 at S6P1, before being overwritten by a PC write (value 0x223E) at S6P2 ('quan-cycle' 0).

Note: the S-PC signal is shown as a glitch in Figure 12.1 since it is derived from the CMP0 and CMP1 signals, which are derived from the PC and the SPC's value. Hence, when the PC is corrected, the S-PC signal will be reset, causing the glitch. When propagation delays are accounted for, the duration that S-PC is set would be as long as the propagation delays necessary to change CMP0, CMP1, the PC and the SPC. Therefore, the PC will be correctly loaded with the SPC's value on the actual hardware.
In Figure 12.2, the error (value of 0xA5A5) is injected between ‘quan-cycle’ 3 of a ‘check’ operation and the PC read. Since the SPC now differs from the PC by more than one, CMP0 and CMP1 would be low, setting the S-PC signal. This causes the PC to be corrected between the PC read and PC write; hence, the PC takes the value of 0x0001. At the same time, the ERROR flag is raised.

Since the PC has read the erroneous value, this value is incremented by the PC’s incrementor and written back at the next sub-cycle, as shown by the write of 0xA5A6. This erroneously incremented value is not stored during the ‘checkstore’ operation, but correction does not take place. Therefore, such errors are detected, but cannot be corrected (DET).

‘Correction’ only occurs at the following ‘check’ cycle but the non-increment SPC’s value (0x0001) is loaded instead (it should be 0x0002). Hence, the ERROR flag is vital in this situation: it would be up to the program to correct such errors.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 12.3: Timing diagram of ST2 detecting and correcting errors

*Figure 12.3* shows ST2 detecting and correcting the errors that were injected. The first injected error was overwritten by PC increments, hence undetected. The second error was injected before a ‘check’ operation and the PC was restored during that operation. The *ERROR* flag was set since there is a slight possibility that ST2 itself could have been corrupted. The third error was injected (‘quan-cycle’ 2) just before the ‘check’ operation completed (‘quan-cycle’ 3); hence, it too was detected. Note: the *ERROR* flag remained set.

Referring to *Figure 12.4*, the errors injected during the ‘store’ operations are undetectable. Hence the PC – originally at 0x0003 – takes on the corrupted value of 0x1234. The *ERROR* flag was set during an earlier detected error that is not shown in *Figure 12.4*.

Based on these timing diagrams, the behaviour of ST2 has been shown to work as expected.
Techniques intended to reduce the impact of program-flow errors on embedded systems

![Timing Diagram for Undetected PC Errors]

**12.2.3 Implementation size**

A gate count breakdown of ST2's modules is shown in Table 12.3. The final implementation size – based on XST's report – is 817\(^{56}\) logic gates. The 28 gate difference (845 – 817) is due to optimisations carried out when all modules were synthesized under a top-level design.

<table>
<thead>
<tr>
<th>Module</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>288</td>
</tr>
<tr>
<td>SPC</td>
<td>128</td>
</tr>
<tr>
<td>COMP</td>
<td>240</td>
</tr>
<tr>
<td>CREG</td>
<td>157</td>
</tr>
<tr>
<td>CTRL</td>
<td>32</td>
</tr>
<tr>
<td>Total</td>
<td>845</td>
</tr>
</tbody>
</table>

Table 12.3: Summary of ST2's implementation gate count

\(^{56}\) The actual reported gate count is 1073 gates. The PC however, was duplicated in the IOB flip-flops – (see Appendix K) since ST2 was synthesized as a standalone unit. Hence the actual gate count, when ST2 is implemented on an MCU, will be 817 gates (1073 – 128 – 128; 8 gates per flip-flop).
Techniques intended to reduce the impact of program-flow errors on embedded systems

By regarding ST2's implementation size at 817 gates (without the PC), this means ST and BCT's gate count should be increased by approximately 10.4% ((817-740)/740 gates). The revised implementation overhead (excluding PC) estimates are shown in Table 12.4.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Previous</th>
<th>% of TE-51</th>
<th>New</th>
<th>% of TE-51</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>514</td>
<td>1.77</td>
<td>567</td>
<td>1.95</td>
</tr>
<tr>
<td>ST2</td>
<td>740</td>
<td>2.55</td>
<td>817 (actual)</td>
<td>2.81</td>
</tr>
<tr>
<td>BCT</td>
<td>1206</td>
<td>4.15</td>
<td>1331</td>
<td>4.58</td>
</tr>
</tbody>
</table>

Table 12.4: Revised estimate gate count for ST, ST2 and BCT

Note: apart from ST2, ST and BCT's estimate is assumed to increase in the same proportion. ST and BCT's gate count estimates however, have never been verified.

12.3 Comparison between NF/FT and ST/ST2/BCT

NF and FT complement each other in their error detection coverage: both techniques are necessary to ensure erroneous PC 'landing' in all code memory location can be detected. This is not the case with ST, ST2 and BCT: each technique monitors the entire addressable code memory. Moreover, the tracking of the entire PC's width (16-bits) means memory aliasing does not influence their error detection coverage, unlike NF and FT. Above all, NF and FT only detect errors; ST, ST2 and BCT correct them as well.

Although NF/FT detect more PC corruption (85% – average of Alarm_D and Prog_D, see Chapter 7) than ST/ST2/BCT (between 72% and 80%), ST2 and BCT have a higher overall error correction rates (93% and 98.23% respectively). The only significant drawback of ST, ST2 and BCT is the fact that they cannot detect program-flow errors caused by the corruption of other program-flow registers (e.g. Stack Pointer).

ST, ST2 and BCT's error detection and correction of PC corruption occur in real-time, which means the processor state should not change between error injection and correction. This cannot be achieved by NF or FT. Indeed, FT's error detection latency could mean 'critical' instructions may be executed before detection occurs, as illustrated in Figure 4.6.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Since ST, ST2 and BCT’s operations work on the clock-cycle level, they can handle interrupt without any modifications. This is not the case for FT, however. On the other hand, the former techniques are not able to detect errors if they are not detected at the upcoming ‘check’ or ‘checkstore’ operations.

NF and FT require processor resources in the form of RAM, code memory and processing cycles. On the other hand, ST, ST2 and BCT do not incur any. The latter techniques would increase the implementation size (die area), but as Table 12.2 shows, this increase is relatively insignificant as compared to the 8051’s die size.

Being software-based, NF and FT require modifications to existing programs. NF’s modification is trivial but FT’s can be very time consuming and error prone. Since ST, ST2 and BCT are transparent to the processor, programs and development tools do not need to be modified in any way.

### 12.4 Evaluating chip-based techniques

A direct comparison between ST, ST2 and BCT and other chip-based techniques is not possible, due to the different error detection levels and coverage of each approach. For example, Signature Monitoring and the Watchdog Processor may detect program-flow errors at the program level. ST, ST2, BCT and Parity-based Mechanisms detect program-flow errors at the circuit level. Micro Rollback is not even an error detection technique. SM and WP may detect program-flow errors that occurred a few cycles earlier, and those caused by other program-flow related registers, not only the PC: ST, ST2, BCT and PM cannot do this, unless they are implemented on all respective registers.

To form a reasonable basis for comparison, a hypothetical set up is assumed. Some techniques are combined while others are duplicated, as described in the following list:

- ST, ST2, BCT and PM are implemented for the Stack Pointer, PC incrementor, Data Pointer, Instruction Register, Accumulator and Program Status Word and the Program Counter (7 registers).
- PM’s implementation uses Hamming ECC for single error correction and double error detection.
- SM is combined with MR.
- WP is combined with MR.
Techniques intended to reduce the impact of program-flow errors on embedded systems

When ST, ST2 and BCT are implemented on seven registers, only one CREG/BREG/IDEC is necessary. As a result, the second to seventh implemented register will require less D-flip-flops, up/down counters and combinational logic. Based on the estimated gate count of ST, ST2 and BCT's modules (see Table 9.4, Table 10.3 and Table 11.3), and multiplied by a factor of 1.104 (difference between ST2's estimated and actual gate count), their estimated gate count (excluding CRAG, BREG and IDEC), for subsequent implementations, would be approximately 463 (567 – 104), 704 (817 – 113) and 624 (1331 – 707) gates respectively, as shown in Table 12.5.

PM’s overhead for each register is 6 D-flip-flops [Grey 2000, Hall 1989] to store the parity bits, and approximately 100 gates for the parity generation, checking and correction circuitry. This equates to approximately 150 gates for each implementation. The approximate gate count for 7 registers implementing ST, ST2, BCT and PM is shown in Table 12.5. The implementation overhead for SM/MR and WP/MR combinations are considered as the summation of both techniques.

<table>
<thead>
<tr>
<th>Technique</th>
<th>First</th>
<th>Subsequent</th>
<th>Total</th>
<th>% of TE-51</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>567</td>
<td>463</td>
<td>3345</td>
<td>11.51</td>
</tr>
<tr>
<td>ST2</td>
<td>817</td>
<td>704</td>
<td>5041</td>
<td>17.34</td>
</tr>
<tr>
<td>BCT</td>
<td>1331</td>
<td>624</td>
<td>5075</td>
<td>17.46</td>
</tr>
<tr>
<td>PM</td>
<td>150</td>
<td>150</td>
<td>1050</td>
<td>3.61</td>
</tr>
</tbody>
</table>

Table 12.5: Estimate gate count for ST, ST2, BCT and PM's implementation on seven registers

The combined techniques (SM/MR and WP/MR) have never been designed, tested nor implemented. They are assumed to work, however, and are solely used as a basis of comparison. The implementation of ST, ST2 and BCT on multiple program-flow related registers was not investigated in this thesis. ST, ST2 and BCT's performance under such conditions are only estimates, and confined to this section. Unless explicitly stated, ST, ST2 and BCT refers to the implementation on the PC alone.

12.4.1 Production cost overhead
Since production cost is directly related to the implementation overhead, it is necessary to evaluate the techniques based on their estimated overhead.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Technique</th>
<th>Estimate implementation overhead (% of TE-51 die size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>11.5</td>
</tr>
<tr>
<td>ST2</td>
<td>17.3</td>
</tr>
<tr>
<td>BCT</td>
<td>17.5</td>
</tr>
<tr>
<td>PM</td>
<td>3.6</td>
</tr>
<tr>
<td>SM/MR</td>
<td>37.0</td>
</tr>
<tr>
<td>WP/MR</td>
<td>40.0</td>
</tr>
</tbody>
</table>

Table 12.6: Estimated implementation overhead (% of TE-51) for combined techniques

Table 12.6 shows that the estimated overheads for ST, ST2 and BCT are less than those for SM/MR or WP/MR’s, but at least two and a half times those for PM. However, it must be stressed that PM can only detect double bit errors, and correct single bit errors. Therefore, although PM is suitable in detecting high-energy particle-induced errors (and virtually all single-event upsets), it is probably unsuitable for EMI-induced errors (where multiple bit errors could occur concurrently).

The implementation overheads for SM/MR and WP/MR are very high, since each technique alone already has a high overhead.

One interesting observation is ST2’s and BCT’s overheads are almost identical when implemented on many registers. Indeed, if 8 registers were implemented, ST2’s estimated overhead would be higher than BCT’s. This anomaly happens because 1) ST2’s comparator module is approximately four times the size of BCT’s, and 2) BCT’s IDEC module, which accounted for the bulk of its increase over ST2 for the first implementation, is not replicated.

Based on the scheme described, PM’s production cost overhead is negligible. ST, ST2 and BCT will incur some cost overhead, but significantly less than that for SM/MR or WP/MR.

12.4.2 Error detection rates

It is not possible to estimate the error detection rates for WP/MR since no results for it are available. The error detection rate for PM is hard to estimate: Gaisler’s [Gaisler 1997] figure

---

57 This module may have to be replicated if concurrent access to more than one BCT-implemented register is necessary.
Techniques intended to reduce the impact of program-flow errors on embedded systems

of 97.5% was based on PM being implemented on almost all registers. When PM is implemented on 7 registers, its program-flow errors detection and correction rate will almost certainly be less. In addition, multi-bit errors that occur as a result of (say) EMI-induced noise may be undetectable.

ST, ST2 and BCT’s overall error detection rate (all causes of program-flow errors) is very likely to increase (as compared to implementing it on the PC alone) when they are implemented on multiple registers (according to the scheme proposed in Section 12.4). SM/MR’s error detection rate is quoted from [Shuette & Shen 1987]: this is somewhat dependent on the program. Hence, PM and WP/MR’s error detection rates are not estimated in Table 12.7.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Estimate error detection rates (% of injected errors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>82.1</td>
</tr>
<tr>
<td>ST2</td>
<td>83.9</td>
</tr>
<tr>
<td>BCT</td>
<td>89.7</td>
</tr>
<tr>
<td>SM/MR</td>
<td>82.0</td>
</tr>
</tbody>
</table>

Table 12.7: Estimate error detection rates for combined techniques

If ST, ST2 and BCT’s program-flow error detection rate is increased by 10%, their estimated error detection rates, as shown in Table 12.7, ranges between 80% and 90%. This is within the same range as SM/MR.

It can only be concluded that PM is only suitable for situations where multi-bit errors do not (or rarely) occur. In addition, ST, ST2 and BCT’s overall program-flow error detection rate, when implemented on multiple program-flow related registers, is likely to increase.

12.4.3 Error detection latency
PM, ST, ST2 and BCT’s error detection and correction takes place in real-time. As SM’s signatures are periodically checked, the maximum latency between error occurrence and detection is equal to the maximum duration between two checks. WP/MR’s error detection latency may be very short since WP monitors the bus and processor states concurrently. In this aspect, PM, ST, ST2, BCT and possibly WP/MR have a clear advantage over SM/MR.
Techniques intended to reduce the impact of program-flow errors on embedded systems

12.4.4 Processor resources
PM, ST, ST2 and BCT do not require processor resources. SM/MR will require code memory to store the signatures and instructions to check them, which also incurs processing time. This is also the case for WP/MR when the monitoring instructions are stored in the program. If a separate monitoring program is used, there may be no processing overhead.

12.4.5 Program alterations
Since PM, ST, ST2 and BCT are transparent to the processor, program alterations are unnecessary. Program modifications should not be needed for SM/MR: a specialised tool could examine the assembly code to insert the signatures. This is also the case for WP/MR, if WP’s checking instructions are encoded as part of the program. On the other hand, if separate ‘monitoring’ programs are used, no modifications are necessary to the original program.

12.4.6 Development tools
ST, ST2, BCT and PM are transparent to the program and processor; hence, development tool modifications are unnecessary. A special development tool to insert the signatures is necessary for SM/MR. A similar tool may also be developed to implement WP’s checking instructions if it is to be inserted as part of the program. If a separate monitoring program is used however, the development tools may need alterations, especially when WP’s instructions are different from the processors’.

12.4.7 Interrupt handling
PM, ST, ST2 and BCT work on the cycle level and would treat interrupt vectoring like an ordinary LCALL instruction. On the other hand, SM has to save the calculated program signatures when interrupts occur [Shuette & Shen 1987]. The situation associated with the re-execution or missing interrupts with MR – as described in Chapter 8 – must also be considered. WP may be able to detect interrupts and adapt itself to such situations.

12.5 Porting ST, ST2 and BCT to other processor families
ST, ST2 and BCT must be modified to work with different processor architectures. The main change would be to the CREG and the BREG: they must be equivalent to the processor’s execution time and instruction set.

---

58 SM’s signatures could be inserted manually. However, this will be a tedious and error prone process.
Techniques intended to reduce the impact of program-flow errors on embedded systems

For BCT, the IDEC would also need to be tailored for different instruction-types. Their internal operations (e.g. ‘check’, ‘checkstore’) however, should remain the same: processors tend to perform specific operations (e.g. loading accumulator, reading memory) at specific cycles for all instructions.

Processor families with few instruction-types (e.g. PIC16X), would have simpler CREG, BREG and IDEC modules.

Any changes to the instruction execution time and size will have an effect on the error detection rates of ST, ST2 and BCT. When implemented on processors with shorter instruction execution times (i.e. instructions require fewer oscillation cycles to execute), ST, ST2 and BCT’s error detection rate will decrease proportionately, due to the relative increase in ‘undetectable’ ‘quan-cycles’.

12.6 Conclusion

ST, ST2 and BCT are effective in detecting and correcting PC corruption. Of the three techniques, ST2 gives the best compromise between error detection rate and implementation overhead. BCT however, may be more suitable, if these techniques are implemented on multiple registers, to improve the overall error detection rate.
Techniques intended to reduce the impact of program-flow errors on embedded systems

PART FOUR
PERIPHERAL-BASED PROGRAM-FLOW ERROR DETECTION TECHNIQUES

The studies described in Part Four aims to develop peripheral-based program-flow error detection and/or correction techniques, which are chip-based techniques that can be controlled through software.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The Watchdog Timer

The Watchdog Timer is the only peripheral-based technique specifically designed to deal with errors on commercially-available processors.

In this chapter, the WDT is described and its advantages and disadvantages discussed.

13.1 The Watchdog Timer


The WDT can be used to detect program-flow errors. Before a specific execution path is taken, the program will load the WDT’s timing register with a value that is sufficient to allow complete execution of the path (sequential code segments or nodes) to the point of the next WDT refresh. Under normal conditions, the WDT will always be refreshed before it underflows/overflows. If a program-flow error occurs (for example; one that branches from the end of a short execution path to the start of a long execution path), the WDT’s timing register may not be refreshed before it underflows/overflows. Hence, such program-flow errors can cause the WDT to reset the processor.

The WDT’s error detection rate is based on the length of the execution path, and its execution frequency. If a processor spends a significant time executing short paths, and most paths are longer, the chances of the WDT detecting program-flow errors will be higher than it would be on a processor that spends a significant duration executing long paths while most other paths
Techniques intended to reduce the impact of program-flow errors on embedded systems

are shorter. In short, the WDT's error detection rate is very implementation dependent and will certainly be less than 100% [Oberg et. al.].

13.2 Advantages of the WDT
The WDT's error detection coverage is independent of the code memory, unlike NF and FT. Since it is not directly tied to the program, the WDT can detect program-flow errors 'landing' in unmeetable conditional loops59 [Banyai & Gerke 1996], which may not be the case with FT.

The WDT is unlikely to be enabled or disabled by program-flow errors. Indeed, some WDTs can only be enabled/disabled via configuration bits during device programming (e.g. PIC16F84), or when multiple bits are set to the correct state (e.g. C515).

13.3 Disadvantages of the WDT
The main disadvantage of the WDT is that the duration of every execution path must be known at compile time [Campbell 1995, Campbell 1998]. Obtaining the necessary information can be extremely time consuming.

The combination of program execution that is based on external events, and the unpredictability of them, mean it may be impossible to accurately time every path. With variable-length execution paths, the WDT's duration may have to be made much longer than their typical execution time. This may 1) decrease the error detection rate when erroneous program-flow branches occur within such paths [Campbell 1995], and 2) cause the WDT's timing register to overflow/underflow when unforeseen conditions occur.

The problems associated with variable-length execution paths can partially be overcome with WDTs that can be disabled. In this case, the WDT is disabled for such paths, which eliminates the chances of the second condition (see previous paragraph) occurring. On the other hand, it also means erroneous program-flow branches that occur within these paths will never be detected, and instructions to enable the WDT must be executed after its timing register is loaded.

59 This assumes that the WDT's timing register is not refreshed in the loop.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Although the WDT’s error detection coverage is independent of the amount of code memory, its effectiveness is affected by memory aliasing. Erroneous program-flow branches to locations that are the memory aliased locations of the programmed code memory may cause the WDT to be refreshed before it overflows/underflows. On the other hand, erroneous branch to unprogrammed code memory locations may cause execution to wrap around the physical code memory boundary, and the WDT may not be refreshed in time. In both cases, such errors may go undetected. The chances of them occurring depend on the WDT’s value when the error occurs and the erroneous branch’s ‘landing’ location.

The WDT’s error detection latency is equivalent to the number of clock-cycles that are executed before the timing register overflows/underflows, from the time the error occurs. Its latency is hard to estimate – even via simulation – when the execution path taken is influenced by external stimulus. The only fact that is certain is that it can be as short as one clock-cycle (i.e. an erroneous branch occur just before the WDT’s register overflow/underflow), or as long as the longest duration between two WDT refreshes.

Interrupts can also be problematic. They can occur towards the end of an execution path, but before the WDT is refreshed, which may cause the WDT to overflow/underflow. The ISR may also need to save the WDT’s current value: this will increase the delay before the WDT is refreshed in the ISR.

For some systems such as those working in environments that do not have fail-safe states, a reset may be an unsuitable ‘recovery behaviour’, since the data space and integrity will be lost [Niaussat 1998]. In such situations, the pseudo-WDTs [Campbell 1995, Coulson 1998] (i.e. a general purpose timer used like a WDT) may be more appropriate: they generate interrupts instead. However, pseudo WDT’s are likely to prove more susceptible to corruption due to their more complex nature.

The WDT requires processing resources. Some code memory locations are necessary to implement the instructions that will refresh the timing register, and processing cycles are required to execute them. This overhead is proportional to the number of WDT refreshes. If, instead, a pseudo-WDT is implemented, a timing unit and some additional code memory for the ISR will also be required.
Techniques intended to reduce the impact of program-flow errors on embedded systems

13.4 Other peripheral-based techniques
To the best of the author’s knowledge, only one other peripheral-based technique to detect processor errors has been mentioned in the literature. Ker & Sung [Ker & Sung 2001] described an electrostatic discharge sensor that was integrated with an 8051 MCU. Multiple sensors were integrated at various locations throughout the processor’s die, and would automatically reset the processor when any sensor detected a transient. The sensors would also set an SFR-mapped bit when a transient was detected. Hence, the program could check this specific SFR location to determine if a reset was the result of ESD transients.

Although such sensors may detect errors induced by EMI and high-energy particles, this avenue cannot be explored in this project as it involves fabrication.

13.5 Conclusion
The Watchdog Timer is the only integrated error detection mechanism found on most commercially available processors, especially microcontrollers. Although the WDT can detect program-flow errors, its error detection rate is unpredictable and (indirectly) affected by memory aliasing. The main disadvantage of the WDT’s implementation is the duration of all execution paths has to be known at compile time, which is not possible for some systems especially those driven by external stimulus (i.e. embedded environments). The possibility of correct, but unforeseen situations occurring that can lengthen a path’s duration beyond the WDT’s refresh value, is also present. Hence, the WDT may not be suitable for some embedded systems.

Due to the potential unsuitability of the WDT, other peripheral-based techniques to detect program-flow errors are proposed in Part Four. Before these techniques are discussed however, an overview of the tools that were developed to assess other peripheral-based techniques is presented in the next chapter.
Support tools for peripheral-based program-flow error detection techniques

In Part Three, the chip-based techniques to detect and correct program-flow errors were simulated with 8051Sim-NG. Although 8051Sim/8051Sim-NG proved to be a flexible environment to design and simulate chip- and peripheral-based techniques, it is a simulator and – like all simulators – the results obtained cannot be guaranteed to match experimental results based on the actual hardware. On top of this, low-level simulations – like those carried out for ST, ST2 and BCT – are computationally intensive [Delong et. al. 1996]. This does not, in any way, mean ST, ST2 and BCT’s simulations are inaccurate: their description (in VBScript) is written to match the hardware implementation as close as possible (see Appendix O). It is a fact, however, that to conclusively prove a technique’s effectiveness, it is necessary implement them in hardware and to execute ‘real-world’ programs under ‘real-world’ conditions.

To evaluate chip- and peripheral-based techniques in this manner, it is necessary to develop – at least in part – the target processor in hardware. There are a few options available to do so: they are discussed in the first part of this chapter. The next part discusses the development of a generic 8051 MCU that acts as the base for implementing chip- and peripheral-based techniques. This is followed by the description of a method used to control the target processor when carrying out simulations.

Note: although the techniques proposed in Part Four are also chip-based, they have been termed peripheral-based since they are software-controllable. This is the only criterion that distinguishes chip- and peripheral-based techniques.

14.1 Options available to develop the 8051

It is not possible to evaluate any proposed chip- or peripheral-based techniques in hardware without an 8051 MCU that can be modified. To develop a modifiable 8051 MCU, four different approaches were considered:

- Create the MCU with an ASIC (application specific integrate circuit) process.
Techniques intended to reduce the impact of program-flow errors on embedded systems

- Create the MCU from discrete components (i.e. using 7400 or 4000 series ICs).
- Use MCUs with accessible internal buses (bond-out or BDM/JTAG\textsuperscript{60}-implemented).
- Create the MCU with FPGAs (see Appendix K).

The first approach is well beyond the budget and time constraints of this project. Moreover, the turnaround time (i.e. time taken to create each new revision of the 8051 MCU) due to fabrication may be longer than desired.

The second approach, though less expensive, would be very tedious and error prone. Although the 8051 MCU is not a complex processor by current standards, a significant number of logic ICs (e.g. 7400 series ICs) would still be required (c.f. each register would require at least one IC).

The third approach is not feasible: bond-out MCUs do not allow modifications of the processor core: they only allow access to the internal buses [Cruttenden 1987]\textsuperscript{61}. This includes Triscend's E5 - an 8051 MCU core with programmable logic – and devices with integrated diagnostics support (e.g. Infineon C167 [Siemens 1996a], Motorola MC68332 [Melear 1997b, Rebaudengo & Reorda 1999] and Microchip PIC16C87x [Microchip 2001]), as they do not have access to the processor’s core.

The fourth approach is feasible: it is relatively cheap to purchase the hardware and development tools. The time taken to implement a design is also short. Hence, this approach was taken in this project.

### 14.2 TE-51: An FPGA-based 8051 microcontroller

With the FPGA route chosen to implement the 8051 MCU, it was necessary to either design the MCU from scratch, or purchase a design from an IP provider. The latter approach requires an 8051 MCU that can be re-engineered, i.e. the entire design – not just a ‘black box’ version of the 8051 MCU – was required.

The following is a list of the companies that were approached, with their product in question:

\textsuperscript{60} Background Debugging Mode/Joint Technical Group

\textsuperscript{61} ‘Bond-out’ devices are also only supplied to the manufactures of emulators [Cruttenden 1987].
Techniques intended to reduce the impact of program-flow errors on embedded systems

- **Digital Core Design** – DR8051
- **Dolphin Integration** – Flip8051-PR
- **CAST Inc.** – C8051
- **Trenz Electronik GmbH** – TE-51

Digital Core Design and CAST did not return our inquiry, while Dolphin Integration quoted a price of 50,000 Euros: this was beyond the budget of this project.

Trenz Electronik GmbH agreed upon the free use of their 8051 core in exchange for the implementation of the generic 8051 peripherals, and the rights to market – within the terms of the agreement – the program-flow error detection and/or correction techniques described in this thesis.

### 14.2.1 Overview of TE-51

The synthesizable 8051 core developed by Mr. Felix Bertram of Trenz Electronic – TE-51 – is cycle-accurate to the generic 8051 MCU. Apart from the multiplication and division instructions (MUL and DIV – these instructions take two cycles instead of four), all other instructions require the same number of bytes and cycles to execute as the 8051. Power down and idle modes however, are not supported.

The original 8051 core obtained from Trenz Electronik was complete, but without the generic 8051 peripherals (except the ports). The interrupt system\(^2\), external interrupts and timers were designed and implemented by the author to the specifications of the generic 8051. This process took approximately four months, which included the time necessary to understand how the core worked. From here onwards, the term 'TE-51' will refer to the Trenz Electronik’s 8051 core integrated with the generic peripherals developed by the author of this thesis.

---

\(^2\) TE-51’s interrupt system was implemented with two extra sources (by using all the bits in the IP and IE register) like 8051Sim/8051Sim-NG. These could be used by custom peripherals.
Techniques intended to reduce the impact of program-flow errors on embedded systems

**Figure 14.1: Block diagram of TE-51’s connections**

TE-51’s connections are shown in Figure 14.1. Since it is meant for system-on-chip (SoC) environments, the address, data and control buses/lines are not multiplexed with P0, P2 and P3, as with the generic 8051 MCU. Hence, the ports on TE-51 will work simultaneously with external RAM/ROM configurations.

When external RAM is not used, only the *Addr* and *Din* buses need to be connected to the code memory. The code memory can be created from the FPGA’s look-up-tables (LUTs), the FPGA’s block RAM (see Appendix K), or from external EPROMs. When XRAM is used, the control lines differentiate between memory reads and writes.

### 14.2.2 Testing

Every 8051 instruction was simulated and tested by Trenz Electronik using Active-HDL's simulator. In addition, Trenz used the 8051 core in some of their products. In both cases, the 8051 core worked as expected. Hence, although the 8051 core was never mathematically verified (that is, it was not formally verified), there is little chance the core has significant hardware errors.

The timers, interrupt system and external interrupts were simulated with Active-HDL, and the design was tested with programs written in Keil’s C51 compiler. Initial tests were carried out on each peripheral individually.

The timers were tested in timer and counter mode for all operating modes (each timer has four modes – see Chapter 2). In particular, both timing units were tested together when one (or both) is in mode 3 (this is the only mode that affects both timing units simultaneously).

The interrupt system was tested with many combinations of priorities and interrupt sources. The tests also included executing the RETI instruction ('Return from interrupt'). Timer
overflows and the external interrupt sources were used as triggers. The test waveforms are located together with the project files (see Appendix R).

The final tests were carried out with programs requiring interrupts and timing units on TE-51. Since these tests – including scheduler-based programs – ran successfully, TE-51 is deemed working correctly, and in the same manner as the 'standard' 8051.

14.2.3 Implementation

TE-51 was implemented on a Xilinx’s XC2S200 Spartan-II FPGA. This family of FPGAs was chosen because:

- It is suitable for processor-like systems (see Appendix K).
- It will comfortably fit TE-51 and other ancillary/peripheral modules (see below).
- It can be developed on Xilinx’s free WebPack ISE development environment.

The FPGA is on an evaluation board – the TE-XC2S Spartan-II Development Board – developed by Trenz Electronik GmbH. Programming and functional simulation was carried out with Aldec’s Active-HDL 4.2, while Xilinx’s XST was the synthesis engine. This workflow applies to all FPGA-based development carried out in this thesis.

TE-51’s implementation size – including the generic peripherals and a module to observe its registers (see next section) – is 1372 configurable logic blocks (CLBs)\(^6\). The equivalent gate count, based on Xilinx XST’s report, is 29069 gates.

14.3 Overview of the MCU Testbench

To evaluate chip- and peripheral-based techniques integrated with TE-51, a module to execute TE-51 in various modes (e.g. step-execute) is necessary. This module should also have facilities to set/execute breakpoints, view the processor’s SFRs and – crucial for this project – the ability to change SFR values (i.e. injecting errors). All of these should be done in real-time and must be unobtrusive to the processor. Although there are commercially available hardware fault injectors, these are expensive devices [Li et al. 1984, Rebaudengo & Reorda 1999].

\(^6\) 58.3% of the 2352 CLBs on the XC2S200 [Xilinx 2001].
Techniques intended to reduce the impact of program-flow errors on embedded systems

The MCU Testbench was developed with these specifications in mind. Since the 8051 MCU only requires the clock and reset signals for normal operation, the MCU Testbench controls these signals to execute it in the various modes mentioned (run, step and micro-step). Other modules in the MCU Testbench allow execution breakpoints to be set, allow the user to observe the SFRs and the processor’s status, and – most important – allow the user to change register values.

The MCU Testbench communicates bi-directionally with the MCU Host, a program running on a desktop computer. This program is responsible for controlling all aspects of the MCU Testbench and is controlled with scripts, to allow autonomous, long-running, simulations. This is an important aspect for ‘Monte-Carlo’-style simulations.

The MCU Testbench and MCU Host form a real-time testing environment. It is similar in many ways to the ‘Fault Injection Support Board’ (FISB), described in [Benso et. al. 1999]: for example, it has the ability to inject errors unobtrusively in real-time. However, unlike the FISB, the MCU Testbench is integrated with the processor (TE-51). As a result, it can monitor all registers and buses, not just a selected few.

14.4 Summary of the MCU Testbench’s capabilities

The following is a summary of the MCU Testbench’s capabilities:

- The MCU can be executed in run, step and micro-step mode.
- The MCU is resetable.
- The MCU can be clocked between 1/2 and 1/255th of the MCU Testbench’s clock.
- Errors can be injected at the sub-cycle level.
- One execution breakpoint can be set.
- All SFRs and processor status (e.g. cycles executed since reset) can be viewed.
- The MCU execution can be controlled through scripts.
- Programs can be loaded into block RAM, which is treated as code memory by the MCU.

---

64 Minor modifications are necessary to allow TE-51’s registers to be accessed by the MCU Testbench. These modifications are discussed in the appropriate section.
14.5 Description of the MCU Testbench modules

Although TE-51 and the MCU Testbench are integrated on a single IC, it is easier to understand the system by considering each module separately. A block diagram of the system is shown in Figure 14.2. Note: the non-colour inverted blocks within the FPGA form the MCU Testbench.

Apart from TE-51, the other modules are associated with controlling the MCU (CTRL), injecting PC errors (ERR), reading the MCU register values (OBS), the MCU code memory (ROM) and communication with the MCU Host (COMINT and UART). Communication between the MCU Testbench and the MCU Host is done via RS232. The modular design of MCU Testbench allows the use of other communication protocols.

14.5.1 Control module

The main task of the control module is to act as TE-51’s clock.

The module has four registers: two of which hold the number of executed cycles and the cycle states (e.g. S1P2, S4P1). These registers – CY (32-bit)\(^{65}\) and SC (4-bit) respectively – are readable by the ERR and OBS modules (see Section 14.5.2 and Section 14.5.3).

---

\(^{65}\) Only 4,294,967,295 clock-cycles can be registered before aliasing occurs.
Techniques intended to reduce the impact of program-flow errors on embedded systems

TE-51’s clock frequency can be set between $1/2$ and $1/255$ of the MCU Testbench’s clock. Such a facility allows TE-51 to operate at a slower rate, or for slower processors to work correctly with the MCU Testbench. This frequency division task is the responsibility of PRES, the third register (8-bit), which is the reload value of a counter that decrements on each clock pulse. When the counter’s value reaches half of PRES’s value, TE-51’s clock input signal goes low. This signal goes high when the counter reaches zero, hence producing the approximately 50% duty cycle clock pulses. Upon reaching zero, the counter is reloaded with PRES’s value and the next cycle commences.

CTRL uses the fourth register – RUNTO – as an execution breakpoint. If the relevant command (see Appendix I) is issued, the testbench executes TE-51 until CY’s value is equivalent to RUNTO’s. When this occurs, TE-51’s execution is stopped.

The control module is also responsible for resetting TE-51, step execution and microstep execution. In total, eight different commands are available to control TE-51’s execution. The status of TE-51 (running or not running) can also be observed by sending the appropriate command. When TE-51’s execution is stopped (i.e. when the breakpoint is reached), a command is sent to the MCU Host to inform it.

14.5.2 Error Injection module

The error injection module contains three sets of registers that are used to hold the value of the error injection cycle and sub-cycles, and the erroneous PC value. These registers are loaded when the appropriate command is received.

When these registers have been set and error injection is enabled, the module will change the PC’s value to that of EPC when the CY and SC values are equivalent to ECY and ESC respectively. This constitutes the error injection process. For another error to be injected, this module has to be reset by issuing the appropriate command.

The ERR module can be programmed either to stop TE-51’s execution immediately after the error is injected, or to allow TE-51 to continue running.

---

66 The MCU Testbench permits other FPGA-based processor cores to be controlled with minor modifications.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Note: the register in which errors are injected has to be modified. This modification – carried out on the Program Counter – is discussed in Section 14.7.

14.5.3 Observer module
When the observer module receives the appropriate command from COMINT, it will read the RAM, SFR or PC content from TE-51, or the values of the testbench registers (CY, SC, PRES, RUNTO, ECY, ESC and EPC). Once the relevant information has been read, it sends the data via the COMINT and UART module to the MCU Host.

14.5.4 ROM module
The ROM module is responsible for loading the MCU’s firmware into the block RAM, and allowing TE-51 to use it as code memory. Since Xilinx’s Spartan-II family of FPGA have block-RAM\(^7\), this is used instead of the LUTs (which would have made the implementation very big). Another major advantage of using the block RAM as TE-51’s code memory is test programs can be downloaded without re-synthesizing TE-51 and the MCU testbench. This saves a lot of time (each synthesizing process requires approximately half an hour compared with a few seconds to fill the block RAM).

As only 7kB of block RAM is available in the XC2S200, a 4kB code memory device was implemented.

14.5.5 Command Interpreter module
The UART only accepts data that are to be transmitted in byte-sized chunks, and all commands are six bytes wide. The command interpreter module is responsible for sending each byte of the command sequentially, and to generate the command header and checksum.

When MCU Host to MCU Testbench commands are received by the UART, they are also sent one byte at a time to this module, which then reconstructs the command. Once all six bytes have been received, the COMINT module validates the command, and sends it to the appropriate module (CTRL, ERR, OBS or ROM).

---

\(^7\) 7kB for the XC2S200.
Techniques intended to reduce the impact of program-flow errors on embedded systems

14.5.6 Universal Asynchronous Receiver Transmitter module
The UART module has been fixed to transmit and receive in the following format:

- 1 start bit
- 8 data bits
- 1 stop bit (no parity)

The baud rate is set at 56,000 bits per second. This can easily be altered by changing the UART’s clock frequency, which is derived from a 48MHz clock. Note: the UART has to be clocked at a frequency 16 times the baud rate.

Although the MCU Testbench is not capable of communicating above (approximately) 950 commands per second due to the slow interface, it can emulate the processor and inject errors in real-time and unobtrusively. These factors are arguably the most important to accurately evaluate processors. Other techniques (e.g. using BDM [Melear 1997b, Rebaudengo & Reorda 1999] and the Fault Injection Support Board [Benso et. al. 1999]) are intrusive to some degree.

14.5.7 Hardware implementation
The MCU Testbench’s implementation is equivalent to approximately 9000 logic gates requiring 559 CLBs (23.8% of the XC2S200). Due to time constraints, the MCU Testbench was not optimised (i.e. its description in VHDL was not refined for CLB structures). This accounts for its relatively large footprint.

14.6 MCU Host
The MCU Host – schematically illustrated in Figure 14.3 – was written in Visual Basic. The graphical interface, shown in Figure 14.4, allows users to control the MCU Testbench and configure the internal registers (e.g. CY, EPC). Once certain buttons are pressed, or certain functions are executed within the user script, the ‘Command Transmitter’ would encode the command and send it one byte at a time to the ‘RS232 controller’. The ‘RS232 controller’ using Microsoft’s MSCOMM control, sends the commands via the serial cable to the MCU Testbench.
On receiving commands from the MCU Testbench (e.g. when TE-51’s execution stopped), the RS232 Controller sends the command, one byte at a time, to be validated by the ‘Command Receiver’. Valid commands are then passed on to the ‘Graphical Interface’ and ‘Scripting Interface’ modules.

As with the 8051Sim/8051Sim-NG’s scripting interface (see Chapter 6), Microsoft’s Scripting Control was used as the wrapper for the Scripting Interface. All variables and functions exposed by the MCU Host that are called within the user scripts must be prefixed with ‘tv.’ and ‘tc.’ respectively.

14.6.1 The user interface

TE-51’s execution is controlled by the ‘Run’, ‘Runto’, ‘Step’, ‘Microstep’, ‘Stop’ and ‘Reset MCU’ buttons. The registers – PRES, RUNTO, ECY, ESC and EPC – are set in the dialogue box that is launched when the ‘Setup register’ button is clicked. User scripts – written in VBScript – are loaded and executed from the MCU Host by clicking on the ‘Run Script’ button.

The MCU Testbench’s error injection module is enabled by clicking on the ‘ERR enable’ button. If the ‘stop when error injected’ checkbox is checked, the MCU Testbench will stop executing TE-51 once the error is injected. Once the error injection module is enabled, it can only be disabled when the ‘Reset ERR’ button is pressed. This is also the case after an error has been injected.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Figure 14.4: MCU Testbench interface

The combo boxes in the ‘Observation’ section allow the parameter type (e.g. SFR, PC, CY) and value (SFRs only: e.g. TLO, ACC) to observe to be selected. When the ‘Observe’ button is pressed, the value is shown above that button. Note: only the generic 8051 SFRs (excluding PCON, SBUF and SCON) can be observed.

Eight parameters are observed in the ‘Observer’ section. The upper parameters are TE-51’s (CY, SC, PC), the lower parameters are configured by the user (ECY, ESC, EPC, PRES, RUNTO). Clicking on the ‘Update values’ button updates all parameters. ECY, ESC, EPC, PRES and RUNTO are read from the MCU Testbench when the ‘Update values’ button is pressed, not from the MCU Host itself. This ensures that the actual values programmed into the MCU Testbench are read.

Three other parameters, TE-51’s running status, error injection status and the equivalent clock frequency are listed at the bottom of the program’s window (see Figure 14.4). Pressing the ‘Exit’ button terminates the program.

14.6.2 Scripting
A very important aspect of the MCU Host is its scripting ability. The scripting interface encapsulates the MCU Testbench commands into more meaningful functions and exposes other supporting functions (e.g. file writing). The scripting commands and variables are described in Appendix I. A sample is shown in Listing 14.1.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Listing 14.1: Sample script to control the MCU Testbench

Listing 14.1 starts with resetting TE-51 and setting prescaler (PRES) to three, which means TE-51 is clocked at one quarter the MCU Testbench’s clock. This is then followed by the loading of the program into the code memory (in Intel Hex8 format) and step executing for 10,000 clock-cycles. After each step, the clock-cycle and PC’s value are read. Once 10,000 clock-cycles have been step-executed, the log file is closed and the script terminates.

14.7 TE-51 modifications to incorporate the MCU Testbench

TE-51 requires minor modifications to allow reading of the SFRs, and so that the PC can be accessed by the MCU Testbench.

For register reads, the only modification is each SFR’s output is connected to the MCU Testbench’s OBS module. The selection of the desired SFR is carried out by the 8-bit multiplexer in the OBS module.

Modifications to the PC are more complicated as register writes are also necessary (for error injection). Since the PC may be corrupted at any time – not just on the rising edge of the input clock – the reset input of the PC’s flip-flops are used instead. By ORing the PC’s original reset line with the error injection signal, the PC is resetable by both inputs. If the reset line is activated, the PC’s value is reset. On the other hand, the PC will be set to a predefined value when the error injection signal is taken low. The code snippet in Listing 14.2 shows how this is achieved.
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
process(clk, reset)
begin
  if reset = '1' then
    if rst = '1' then
      PC <= x"0000";
    elsif ERRinj = '1' then
      PC <= ERR;
    end if;
  elsif rising_edge(clk) then
    reset <= rst or ERRinj;
  end if;
end process;
```

Listing 14.2: PC error injection mechanism

14.8 Conclusion

TE-51 was implemented and successfully tested with real-world programs. The peripherals created work in the same manner as the 8051’s generic peripherals.

TE-51 was instrumental in the development of chip- and peripheral-based error detection and/or correction techniques as it allowed them to be evaluated as part of the MCU.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Code Boundary Guard

ST, ST2 and BCT, described in Part Three, were chip-based error detection and correction techniques that were based on the 8051 instruction read/fetch/decode cycle. Although those techniques were effective in detecting PC corruption, they cannot be used on other processor families without substantial modifications. On the other hand, a few other techniques that are only loosely tied to the processor's core are described in this part, starting with Code Boundary Guard (CBG).

In this chapter, CBG – a technique similar to NOP Fills – is described, simulated and implemented.\(^6\)

### 15.1 Overview of Code Boundary Guard

In Part One, NOP Fills was introduced as a technique to prevent program-flow errors. This software-based technique had many ideal characteristics such as zero processing overheads, low error detection latency, portable and practically 100% error detection rate. Its only significant drawbacks are that, to achieve effective error detection coverage, NF requires a large number of unprogrammed code memory locations. For example, to achieve error detection coverage of 75%, the NF region has to be three times the size of program. This means a 8kB program will require 32kB of physical code memory. In addition, NF's error detection coverage is affected by memory aliasing.

Code Boundary Guard – which may be similar to ‘Unused Memory’ [Mahmood & McCluskey 1998, Li et. al. 1984] – is a peripheral-based technique that seeks to overcome NF’s error detection coverage reliance on the physical code memory size. To do so, CBG monitors the PC to determine if its value is within a predetermined threshold.

\(^6\) Please refer to Appendix L for a discussion on peripheral-processor communication.

\(^6\) This technique claimed it was able to detect over 50% of errors, but its mechanism was not fully disclosed.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Since monitoring is carried out over the entire width of the PC (16-bits), CBG’s error detection coverage is not affected by memory aliasing. Instead, its coverage is the ratio of the program size to the addressable code memory, i.e. 64kB.

15.2 Description of CBG

The core of CBG is a 16-bit comparator that is connected to the PC and a preset value. This preset value (CBG’s threshold), which represents the address location two more than the last programmed byte\(^7\), is stored in two SFRs, CBGL and CBGH.

The block diagram of CBG is shown in Figure 15.1. The colour-inverted blocks belong to the 8051 processor core and SFR module.

\[\text{CBGen0} \quad \text{CBGen1} \]

\[\text{ECON} \quad \text{CMP} \quad \text{Comparator} \quad \text{CBGH/}\]

\[\text{CBGL} \quad \text{En} \quad \text{CBGen0} \quad \text{CBGen1} \quad \text{EDC} \]

\[\text{PC} \quad \text{Internal Address BUS} \]

Figure 15.1: Schematic of Code Boundary Guard

The enable (EN) module enables or disables CBG when both flags (bits) – \textbf{CBGen0} and \textbf{CBGen1} – are set to the correct state. Two bits are used to improve CBG’s immunity to corruption: both flags – located in the ‘Error Detection Control’ (EDC) SFR – have to be set to 0 and 1 respectively to enable CBG (\textbf{CBGen} high), and set to 1 and 0 to disable it (\textbf{CBGen} low). Other combinations will not change CBG’s status.

Whenever the PC’s value is more than the 16-bit equivalent of CBGH and CBGL (above CBG’s threshold), COMP’s output (\textbf{cmp}) goes high. When CBG is enabled, \textbf{cmp} will be read

\[7\] The PC will momentarily (after the PC increment but before the PC is written with a new value) point to the code memory location after the last programmed code memory location when executing the last instruction/operand.
at S1P1 by the ‘Error Control’ (ECON) module. If \texttt{cmp} is high, the \texttt{CBGerr} flag will be set to signal the error. When CBG is not enabled, the state of \texttt{cmp} is irrelevant.

\texttt{CBGerr} is a signal sent to the SFR module to update the CBGerr bit in the ‘Error Detection Interrupt Flag’ (EDINT) SFR. EDINT holds CBG’s error flag: it is polled by the interrupt system\textsuperscript{71} to determine if an error had been detected. CBGerr is also the bit that can be read and written by the processor. Hence, it is important to realise that — from the processor and programmer’s point of view — CBGerr is the only accessible (and visible) flag.

### 15.3 Memory mapping

| Special Function Register EDC (0xC0) | Reset Value: 0x01 |
| Special Function Register CBGL (0xC1) | Reset Value: 0x00 |
| Special Function Register CBGH (0xC2) | Reset Value: 0x00 |
| Special Function Register EDINT (0xC8) | Reset Value: 0x00 |

![Figure 15.2: CBG's control registers](image)

CBG is controlled through specific registers. These registers are located in the SFR memory space, as shown in Figure 15.2. Although the SFR addresses used here are 0xC0, 0xC1, 0xC2 and 0xC8, they may be varied (as required) in other implementations. These SFR locations were chosen for this project since they are not used on the generic 8051 MCU. For processors with wider data bus widths (e.g. 16-bits), some registers can be concatenated (e.g. CBGL and CBGH) to conserve SFR addresses.

\textsuperscript{71} Although the interrupt system is employed as the exception handling mechanism, other methods can be implemented. The advantages and disadvantages of other methods are discussed in Section 15.4.
Techniques intended to reduce the impact of program-flow errors on embedded systems

CBG can be controlled from within any program by including the definitions of its registers. This is best done in a header file: a sample definition – using Keil C51’s language extensions – is as follows:

```c
sfr EDC = OxCO;
sfr CBGL = OxCl;
sfr CBGH = 0xC2;
sbit CBGen0 = OxCO;
sbit CBGenl = 0xC4;
sbit CBGerr = 0xC8;
```

Listing 15.1: Keil C51 definitions for CBG

Once the above definitions have been created, CBG can be controlled simply by writing to its registers. A C example is shown in Listing 15.2.

```c
CBGL = 0xF6;
CBGH = 0x02;
EDC = 0x10;
```

Listing 15.2: Example setup for CBG in C

15.4 Error detection process

An error detection cycle is shown in Figure 15.3. The ‘F’, ‘R’ and ‘C’ operations are the CBGerro signal being read by the SFR module to update CBGerr at S5P2, the CBGerro signal being reset at S6P1 by CBG, and the CBGerro signal being set low or high by CBG at S1P1, respectively. The interrupt system also polls the CBGerr bit at S5P2 (‘F’ operation).

Based on Figure 15.3, when the error occurs at cycle 0, the error is only detected by CBG at sub-cycle S1P1 of cycle 1, and CBGerro will then be set high. Upon S5P2 of cycle 1, the SFR module will update CBG’s CBGerr bit (in EDINT) with the value of CBGerro, in this case it means CBGerr would be set high. CBGerro is taken low by CBG at the next sub-cycle (S6P1). Since CBGerr is already set (it can only be reset by the processor, not by reading a low from the peripheral), this bit – polled by the interrupt system at S5P2 – will cause a vector to the ISR after four clock-cycles\(^2\), assuming that no interrupt blocking conditions are met.

\(^2\) Two polling cycles plus two LCALL ISR vector cycles.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The implementation discussed in this chapter causes an interrupt whenever CBG detects an error. Another approach immediately vectors program execution to a programmed code memory location. This is equivalent to non-maskable interrupts (NMI) on certain processor families. The advantages and disadvantages of these exception-handling mechanisms are summarised in Table 15.1.

<table>
<thead>
<tr>
<th></th>
<th>Interrupt</th>
<th>Immediate vectoring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>depends on priority and interrupt blocking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>conditions</td>
<td></td>
</tr>
<tr>
<td><strong>Resources</strong></td>
<td>Using unused interrupts (based on generic 8051</td>
<td>Two 8-bit SFRs for vector address</td>
</tr>
<tr>
<td></td>
<td>MCU)</td>
<td></td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>Easily implemented between various 8051</td>
<td>Will need changes to the instruction fetch</td>
</tr>
<tr>
<td></td>
<td>variants and other processor families</td>
<td>module and instruction register</td>
</tr>
</tbody>
</table>

Table 15.1: Comparison between the interrupt and the immediate vectoring exception handling mechanisms
Techniques intended to reduce the impact of program-flow errors on embedded systems

15.5 Simulation

Simulations were carried out on 8051Sim/8051Sim-NG to test the initial idea, evaluate different configurations and to gauge their effectiveness.73

To accurately simulate CBG on 8051Sim, CBG’s VBScript description was written to reflect the hardware as far as possible. There were some minor differences between CBG’s description as compared with the VHDL design however (see next section), these were done to increase simulation speed and did not alter CBG’s behaviour.

The scripts are located in Appendix P and on the CD-ROM (see Appendix R).

15.5.1 Test programs

Two test programs: ‘Krider’ and ‘Prog’ were used in the simulations. These were the same programs used in previous simulations apart from the additional code necessary to configure CBG.

Each program had three variants with different CBG thresholds. Suffixes ‘_A’, ‘_B’ and ‘_C’, their thresholds were set immediately after the last programmed location, half the physical code memory size and half the addressable code memory respectively. This is shown in Table 15.2.

15.5.2 Simulation procedure

<table>
<thead>
<tr>
<th>Program</th>
<th>Last code location</th>
<th>Error injection window</th>
<th>CBG value (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_A</td>
<td>2F5</td>
<td>810 1810</td>
<td>2F7</td>
</tr>
<tr>
<td>Krider_B</td>
<td>2F5</td>
<td>810 1810</td>
<td>7FF</td>
</tr>
<tr>
<td>Krider_C</td>
<td>2F5</td>
<td>810 1810</td>
<td>7FFF</td>
</tr>
<tr>
<td>Prog_A</td>
<td>769</td>
<td>720 1720</td>
<td>76B</td>
</tr>
<tr>
<td>Prog_B</td>
<td>769</td>
<td>720 1720</td>
<td>7FF</td>
</tr>
<tr>
<td>Prog_C</td>
<td>769</td>
<td>720 1720</td>
<td>7FFF</td>
</tr>
</tbody>
</table>

Table 15.2: Statistics of test programs

---

73 Although CBG was eventually synthesized and integrated with TE-51, TE-51 and the MCU Testbench had not been completed at this stage.
Techniques intended to reduce the impact of program-flow errors on embedded systems

An error is injected by changing the PC to a random value any time within the error injection window. Like ST, ST2 and BCT's simulation, the errors injection window was 1000 clock-cycles. Unlike those simulations, there is a non-zero minimum instruction cycle limit for error injection. This limit is necessary as CBG only becomes active after the 712th and 808th instruction cycles for Krider and Prog respectively. Table 15.2 shows the statistics of the test programs.

Note: CBG is activated on the same cycle for all variants of each program. All variants of each program also have exactly the same code size. This is expected: the only changes amongst them are the values written to CBGL and CBGH.

8051Sim-NG had to be modified for this simulation. As 8051Sim-NG’s (and 8051Sim’s) PC cannot be aliased (in the same manner as the 8051), the effects of memory aliasing cannot be observed\(^\text{74}\). Hence, 8051Sim-NG’s ROM size was increased from 8kB to 64kB, for this particular simulation.

The flow chart of CBG’s simulation is shown in Figure 15.4. EPC’s value is generated to point to any code memory location. The erroneous cycle and erroneous sub-cycle where the errors are injected are also randomly generated in advance. Each injected error constitutes an error cycle.

An error cycle starts with the resetting of the MCU and CBG, followed by the generation of ECY, ESC and EPC. Once the set up is completed, the simulation script step-executes the MCU and CBG. Upon reaching the ECY\(^{\text{th}}\) and ESC\(^{\text{th}}\) clock-cycle and sub-cycle respectively, the PC’s value is changed to EPC’s (error injection process). Note: although the PC is not the only register where corruption would affect program-flow, the corruption of other program-flow related registers would ultimately cause the wrong value to be written into the PC. Since the effect is the same as PC corruption, it is only necessary to corrupt the PC.

Once the error has been injected, up to 10 additional clock-cycles are step-simulated. Upon each step-execution, the simulation script checks to determine if CBG has detected the error

\(^{74}\) In the 8051, the PC will store any values above the implemented code memory; it is the upper unconnected address lines that causes memory aliasing. 8051Sim and 8051Sim-NG’s PC only stores the value that is modulo-8192.
Techniques intended to reduce the impact of program-flow errors on embedded systems

by checking its error signal, \texttt{CBGerro} (it should be \texttt{CBGerr} but it does not matter in this situation – the focus is on CBG's error detection, not its exception handling mechanism). If an error is detected, it is classified as detectable (DET) and a new error cycle starts. It is assumed that DET-classified errors will be corrected.

If CBG did not detect an error by the end of the additional clock-cycles, the error is classified as undetected (UNDET) if the PC points to unprogrammed code memory locations, or else the error is classified as 'others’ (OTH). UNDET-classified errors would mean that CBG was not working as it should. OTH-classified errors would mean that the execution is still within the programmed code memory locations, but whether the injected error has any effect on the integrity of the program is unknown.

Simulator errors that are detected at any stage are classified as unknown (UNK): this takes precedence over DET, UNDET and OTH and will start a new error cycle. In total, 1000 error cycles were simulated for each variant.

![Flow chart of CBG's simulation](image)

Figure 15.4: Flow chart of CBG's simulation
15.6 Results and discussion

Simulation was carried with 8051Sim-NG on a Pentium 350 MHz class PC with 128MB RAM running Windows 2000. Each simulation cycle took between 14 and 16 hours to complete. The results obtained from the simulations are shown in Table 15.3.

<table>
<thead>
<tr>
<th>Program</th>
<th>DET</th>
<th>OTH</th>
<th>UNDET</th>
<th>UNK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_A</td>
<td>845</td>
<td>155</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Krider_B</td>
<td>851</td>
<td>138</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Krider_C</td>
<td>469</td>
<td>160</td>
<td>371</td>
<td>0</td>
</tr>
<tr>
<td>Prog_A</td>
<td>857</td>
<td>143</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prog_B</td>
<td>843</td>
<td>157</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prog_C</td>
<td>446</td>
<td>159</td>
<td>395</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 15.3: Results of CBG’s simulation

From Table 15.3 it is clear that not all errors are detected by CBG, even for A variants. This is not a surprise: the following is a list of cases where CBG will not detect errors:

- Injected errors that point the PC below CBG’s threshold (set by CBGL/CBGH).
- Injected errors that are overridden by PC writes. Such errors do not affect program-flow.
- The vectoring to ISRs may mask some errors. If the corrupted PC’s value pushed onto the stack is in CBG’s detection region, CBG should detect the error once the stack is popped\(^7\).

Errors conforming to the list, and program execution that remains within the programmed code memory locations, are be OTH-classified. Such errors are undetectable by CBG.

It cannot be concluded however, that program-flow will never erroneously branch into the unprogrammed code memory locations after the additional 10 clock-cycles. If some program-flow related registers have been affected by the injected errors, program-flow errors may happen after a certain interval when specific conditions are met. When this occurs, CBG will still detect these errors.

---

\(^7\) This phenomenon cannot be observed here as the number of additional cycles simulated after error inject is shorter than the ISR. However, it is reasonable to assume that this situation rarely occur with both test programs (only 3 interrupts occur within the ‘error window’).
Techniques intended to reduce the impact of program-flow errors on embedded systems

Referring to Table 15.3, all the undetected errors for A variants were OTH-classified, none UNDET-classified, which means CBG detected all the errors it was supposed to (injected errors that branches program-flow beyond CBG’s threshold). Due to CBG’s detection technique, it is not possible for the PC’s value to be above its threshold for more than one instruction cycle, unless it is not enabled or corrupted. Therefore, CBG achieves a 100% error detection rate for detectable errors.

For B and C variants, some of the errors are UNDET-classified, which means the PC’s value is between the last programmed code memory location and CBG’s threshold. Such errors will not be detected by CBG and the processor would be executing ‘empty’ code memory locations. If program execution was allowed to continue past the CBG’s threshold however, CBG will detect the error. Note: Prog_B did not receive any UNDET-classified errors since its last programmed location was close to CBG’s limit.

Table 15.3 also shows that CBG’s error detection coverage is the ratio of the contiguous unprogrammed addressable code memory to the entire addressable code memory. In short, CBG is unaffected by memory aliasing. Hence, if a 4kB program is programmed onto a 8kB device, this would statistically present a 50% error detection coverage for NF. CBG’s detection coverage would be 93.75% (4kB / 64kB), in this situation. When CBG’s threshold is increased, as with B and C variants, the number of detected errors decreases accordingly. C variants detect less than half the injected errors: this is due to some errors being OTH-classified.

CBG does not completely solve NF’s memory aliasing problem. If an MCU implements 64kB of code memory, NF’s coverage will be the same as CBG’s, but this is rarely the case.

CBG will not detect any errors when it is not enabled. Unlike NF, CBG is not a passive method. The process to enable CBG only requires nine bytes and six clock-cycles, however, which is significantly less tedious than WDT’s implementation. CBG’s registers can also be pre-programmed with the desired values during device manufacture (masked ROM) or device

---

76 Although WDT’s error coverage is not affected by memory, its effectiveness is.
77 From another perspective, if the corrupted PC points to a location at 0x27FF, NF would not have detected this error since it would be equivalent to 0x7FF on the 8kB device. CBG will.
Techniques intended to reduce the impact of program-flow errors on embedded systems

programming (e.g. with configuration bits): this allows CBG to start automatically, but also makes it tied to one particular implementation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Latency (sub-cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min.</td>
</tr>
<tr>
<td>Krider_A</td>
<td>2</td>
</tr>
<tr>
<td>Krider_B</td>
<td>2</td>
</tr>
<tr>
<td>Krider_C</td>
<td>2</td>
</tr>
<tr>
<td>Prog_A</td>
<td>2</td>
</tr>
<tr>
<td>Prog_B</td>
<td>2</td>
</tr>
<tr>
<td>Prog_C</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 15.4: CBG's error detection latency

Since CBG's error signal (instead of the flag) is checked in the simulation, the results in Table 15.4 do not include the overhead of the exception handling mechanism: the interrupt system. CBG's average error detection latency is less than one clock-cycle. This is expected: the injected errors will be detected at S1P1 and the error signal set at S1P2, as shown in Figure 15.3. Hence, the minimum latency is two sub-cycles. The average detection latency of 8 sub-cycles is also expected: it lies roughly between two CBG check sub-cycles. On the other hand, the maximum error detection latency for a minority of cycles (the standard deviation is close to the average) can be up to 108 sub-cycles (roughly 9 clock-cycles). These rare occurrences are attributed to injected errors that could have been saved onto the stack (i.e. they occur during branching instructions – LCALL) and their effects only felt after the stack is popped.

By using the interrupt system, CBG's error correction latency is dependent on the conditions that can prevent interrupt requests from being processed. If the ideal conditions exist, CBG's error correction latency is four instruction cycles, one less than the alternative form of NF, and generally less than WDT's. During CBG's error correction latency period, instructions will be executed, which is not the case for NF, ST, ST2 and BCT. This should not be a problem for most designs, however.

Unlike ST, ST2 and BCT, CBG will detect errors after they occur. CBG will also detect program-flow errors from other sources (e.g. Stack Pointer error). Note: as with NF, CBG
Techniques intended to reduce the impact of program-flow errors on embedded systems will detect compiler/linker errors causing branching to locations outside programmed code memory locations or wrongly calculated branches due to erroneous condition values.

Table 15.5 shows the comparison between NF and CBG, the two directly related techniques.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>NF</th>
<th>CBG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection coverage</td>
<td>Only outside programmed code memory locations</td>
<td>Only outside programmed code memory locations</td>
</tr>
<tr>
<td>Detection rate (of detectable errors)</td>
<td>Practically 100%</td>
<td>Practically 100%</td>
</tr>
<tr>
<td>Maximum detection latency</td>
<td>5 instruction cycles (no interrupt - alternative implementation)</td>
<td>4 instruction cycles (best-case interrupt condition)</td>
</tr>
<tr>
<td>Software implementation</td>
<td>Trivial</td>
<td>3 lines of code!</td>
</tr>
<tr>
<td>Hardware implementation</td>
<td>N/A</td>
<td>Minor die size increase</td>
</tr>
<tr>
<td>Time to implement</td>
<td>Negligible</td>
<td>Shorter than NF</td>
</tr>
<tr>
<td>Portability</td>
<td>No problem</td>
<td>Minor reconfiguration</td>
</tr>
<tr>
<td>Configurability</td>
<td>Fully configurable</td>
<td>Fully configurable</td>
</tr>
<tr>
<td>System resources (normal condition)</td>
<td>None</td>
<td>Few ROM bytes</td>
</tr>
<tr>
<td>Processing overheads</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 15.5: Comparison between NF and CBG

15.7 Implementation

The results of the simulations have shown that CBG will detect practically all errors within its coverage. To ensure CBG can feasibly be implemented in hardware, it is described in VHDL according to TE-51’s framework (Listing 15.3), and then synthesized with XST.

```vhdl
entity te51CBG is
  port(
    clk: in STD_LOGIC; -- clock
    rst: in STD_LOGIC; -- reset
    csp: in te51stateT; -- CSP input
    EDCi: in STD_LOGIC_VECTOR(7 downto 0); -- Error detection/correction
    CBGLi: in STD_LOGIC_VECTOR(7 downto 0); -- CBG starting location
    CBGH: in STD_LOGIC_VECTOR(7 downto 0); -- CBG starting location
    PCi: in STD_LOGIC_VECTOR(15 downto 0); -- PC input
    CBGerro: out STD_LOGIC -- CBG error flag
  );
end te51CBG;
architecture bhv of te51CBG is
  signal CBG: STD_LOGIC_VECTOR(15 downto 0); -- CBG
  signal CBGen: STD_LOGIC; -- CBG enable flag
  signal COMP_A: STD_LOGIC; -- compare (AND with CBGen)
  signal CMP: STD_LOGIC; -- compare flag after ANDing
  signal CBGerr: STD_LOGIC; -- error flag
  alias CBGen0: STD_LOGIC is EDCi(0); -- to make life easier
```
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
alias CBGenl: STD_LOGIC is EDCl(4);

begin
process(clk, rst)
begin
  if rst= '1' then
    CBGen <= '0'; CBGerr <= '0'; -- reset flags
  elsif rising_edge(clk) then
    case csp is
      when C1S6P1 | C2S6P1 =>
        CBGerr <= '0'; -- reset error flag
      when C1S6P2 | C2S6P2 =>
        CBGerr <= CMP and CBGen;
      when others =>
        null; -- other sub-cycles
    end case;
    if (CBGenO = '0') and (CBGenl = '1') then -- CBG enabled if cond. met
      CBGen <= '1';
    elsif (CBGen0 = '1') and (CBGen1 = '0') then -- CBG disabled (else remain)
      CBGen <= '0';
    end if;
  end if;
end process;

CBG(7 downto 0) <= CBGLi;
CBG(15 downto 8) <= CBGHi;
CMP <= '1' when (PCi > CBG) else '0'; -- compare flag
CBGerro <= CBGerr;
end architecture;
```

Listing 15.3: VHDL source for Code Boundary Guard

CBG requires seven input signals/bus: three of them from the SFRs, one from the PC, and the rest from the clock, reset and sub-cycle state information (csp). Apart from clk, rst and csp, all other signals to and from CBG are suffixed with ‘i’ for input, and ‘o’ for output.

The csp input from TE-51 shows the current sub-cycle state of the processor: this is prefixed by ‘C1’ and ‘C2’ for first and second cycle of an instruction. Single-cycle instructions run from C1S1P1 to C1S6P2 and double-cycle instructions from C1S1P1 to C2S6P2 on the TE-51 (see Appendix H)\(^78\). CBG has one output for the error signal.

CBG was successfully synthesized according to TE-51’s framework, although it was not integrated with TE-51 at this stage. Based on XST’s report, CBG’s gate count – without the SFRs – is equivalent to 213 gates. If the SFR registers are considered (18 flip-flops), CBG’s implementation would be equivalent to approximately 357 gates, assuming 8 gates per flip-flop (see Table 9.3). Such an implementation is about 50% smaller than ST’s, and about 1.2% of TE-51’s implementation\(^79\). Therefore, CBG is not only implementable in hardware; it can do so with very small gate count overheads.

\(^78\) The four-cycle MUL and DIV 8051 instructions were reduced to two cycles on TE-51.
\(^79\) The overheads on the processor’s side (e.g. additional multiplexor paths) are not considered here.
15.8 Conclusion

CBG has been shown to detect all detectable injected errors (i.e. those within its coverage that were not overwritten). In most cases, its coverage region is not related to the physical ROM size, which is CBG’s main advantage over NF. CBG’s error detection latency is short: it detects program-flow errors within one clock-cycle. Its latency to execute the PFEH is determined by the interrupt handling mechanism (in this implementation), but can be as short as four clock-cycles, which may be significantly shorter than SM, FT and HWFT (and even NF). CBG’s implementation is small, especially when compared with other techniques. Its only disadvantage is the lack of coverage for the programmed code memory locations.

CBG is hardware implementable and requires approximately 357 gates. With its small footprint, good typical error detection coverage and 100% error detection rates, this technique is should be feasible for most embedded systems.
Techniques intended to reduce the impact of program-flow errors on embedded systems

16

Hardware Function Tokens

In the previous chapter, Code Boundary Guard – a peripheral-based technique equivalent to NOP Fills – was described, simulated and implemented. The simulation showed that CBG detected practically all injected errors that were within its detection coverage. Like NF, program-flow errors branching to programmed code memory locations were undetectable. Hence, another error detection technique is necessary to detect branches within the programmed code memory locations.

This chapter starts with the description of a technique that is principally based on FT, followed by a software-based simulation to test and evaluate its effectiveness.

16.1 Overview of Hardware Function Tokens

In Part One, Function Tokens was described as a technique that complements NF’s error detection coverage. Likewise, a peripheral based on FT’s principles would complement CBG’s coverage. Hence, when both techniques are employed, program-flow errors branching to any code memory locations are potentially detectable.

Although FT can be applied in addition to CBG, FT has several major drawbacks: it requires processing time, it increases the program size, it has a relatively long error detection latency (compared with NF) and its implementation process can be tedious and error prone.

The technique proposed here is known as Hardware Function Tokens (HWFT). HWFT has similarities with Signature Monitoring [Wilken 1993] (see Chapter 8) when signatures are assigned. Like SM, the signatures are embedded within the program, and checked at specific locations to detect program-flow errors. Unlike SM, HWFT does not require modifications to the processor’s core and instruction set [Sridhar & Thatte 1982]. HWFT can also be enabled or disabled within the program; since not every code segment will require HWFT checks.
16.2 Functional description

It is easier to describe HWFT based on FT’s principles. As discussed in Chapter 4, FT works by setting a RAM location – known as the token – to the unique identifier of the function that would be called. In HWFT, the same process is carried out: the token – FTval – holds the function’s ID. FTval is a SFR, and is loaded with a function’s ID by memory write instructions.

In FT, a comparison between a function’s ID and the token’s value is carried out to determine if the program-flow is correct. If a mismatch occurs, it signals a program-flow error. For HWFT, a comparison between the token and a function’s ID starts one instruction cycle after the function’s ID is written to another SFR, FTchk. HWFT will automatically perform a comparison between the FTval and FTchk SFRs, and signal a program-flow error when a mismatch occurs.

A schematic block diagram for HWFT is shown in Figure 16.1. The colour-inverted blocks are the SFRs located in the SFR module (see Appendix H).

![Block diagram of Hardware Function Tokens](image)

Figure 16.1: Block diagram of Hardware Function Tokens

The EN module function in the same way as the corresponding module of CBG: EDC holds the two enable bits, \( FT_{en0} \) and \( FT_{en1} \), that are respectively set to ‘0’ and ‘1’ to enable HWFT (\( FT_{en} \) goes high), and ‘1’ and ‘0’ to disable it. Other combinations will not affect HWFT’s state.
The comparator module differs slightly from CBG’s. HWFT’s COMP module latches its output \( \text{cmp} \) on a low to high transition of the \( \text{FTckwr} \) signal. This signal – from the SFR module – goes high for one sub-cycle immediately after the instruction cycle that FTchk is written. Such an arrangement is necessary: COMP must only perform the comparison at the time when FTchk is written. \( \text{cmp} \) goes high when COMP’s inputs differ.

The ECON module is similar to CBG’s: \( \text{FTerro} \) goes high if HWFT is enabled and \( \text{cmp} \) is set. HWFT’s ECON module reads \( \text{cmp} \) at S1P2, instead of S1P1 for CBG’s. This is necessary since \( \text{cmp} \) is latched at S1P1, and the correct value is only available at the following sub-cycle (S1P2). The same situation also means \( \text{FTerro} \) does not reflect HWFT’s status until S2P1.

### 16.2.1 Memory Mapping

Some registers share the same memory locations as CBG. No registers overlap, however. This arrangement allows both techniques to coexist and reduces the number of SFR addresses required. HWFT’s SFRs are shown in Figure 16.2.

![Figure 16.2: HWFT's control registers](image)

As with CBG, \( \text{FTerro} \) is a signal from HWFT to the SFR module. \( \text{FTerro} \) is read by the SFR module at S5P2, which then updates the FTerr bit (EDINT SFR). FTerr can be accessed by the processor, interrupt system and the program, \( \text{FTerro} \) cannot.
Techniques intended to reduce the impact of program-flow errors on embedded systems

HWFT can be controlled from within any program by including its SFR definitions in a header file (see example in Chapter 15). Listing 16.1 shows how HWFT is configured, how the token can be set to a function’s ID, and how HWFT checks are carried out.

```c
EDC = 0x20; //start HWFT
...
FTval = 24;  //set token to 24
...
FTchk = 24;  //token check - good
...
FTchk = 26;  //token check - bad
```

Listing 16.1: Sample C implementation of Hardware Function Tokens

16.3 Error detection procedure
The error detection cycle is similar to CBG’s apart from HWFT checks happening at S1P1 ('C' operation in Figure 15.3). Like CBG, the SFR module reads the state of FTerro and updates the corresponding bit – FTer - at S5P2. The SFR module will only set FTer when FTerro is high; it does not reset FTer when FTerro is low. It is up to the program to do so.

16.4 Simulation
Simulations, similar to CBG’s, are carried out on 8051Sim-NG to test and evaluate HWFT. The simulation scripts are located in Appendix P and on the CD-ROM (see Appendix R).

16.4.1 Test programs
Two test programs; ‘Krider’ and ‘Prog’ are used in the simulations. These are the same programs used in CBG’s simulation. Two variations of each program – suffixed ‘_A’ and ‘_B’ – were developed to evaluate HWFT at 100% function call coverage, and above 100% function call coverage respectively (see Table 16.1).

16.4.2 Simulation procedure
The simulation procedure is similar to CBG’s; hence, only the differences are discussed here (see Chapter 15). Table 16.1 shows the error injection window boundaries for each variant.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Program</th>
<th>Last code location</th>
<th>Error injection window</th>
<th>HWFT checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_A</td>
<td>32F</td>
<td>810 1810</td>
<td>7</td>
</tr>
<tr>
<td>Krider_B</td>
<td>368</td>
<td>810 1810</td>
<td>26</td>
</tr>
<tr>
<td>Prog_A</td>
<td>80F</td>
<td>710 1710</td>
<td>16</td>
</tr>
<tr>
<td>Prog_B</td>
<td>83F</td>
<td>710 1710</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 16.1: Statistics of test programs

The simulation procedure is the same as CBG’s up to the error injection. When the error is injected, up to 500 additional clock-cycles are microstep-simulated, as opposed to 10 for CBG. This increase is necessary to guarantee most, if not all, detectable errors will be detected by HWFT checks. Upon completion of each microstep, the status of the simulator and a variable denoting HWFT’s error detection status, are checked.

If a simulator error is detected, simulation stops and an UNK error is recorded. This condition takes precedence over the rest. On the other hand, a DET error is recorded if HWFT detects the injected error. If neither conditions were detected after executing the 500 additional clock-cycles, the error is classified as UNDET (if the PC is within the programmed code memory locations) or OTH (if it is not).

A modified version of 8051Sim-NG is used in this particular simulation. The original version had 8kB of ROM: this was reduced to 4kB. This modification was carried out as it made direct comparisons between the simulation and benchmark (described in Chapter 18) possible.

16.5 Results and discussion
Simulations were carried out with 8051Sim-NG on a Pentium 350 MHz class PC with 128MB RAM running Windows 2000. A variants took approximately 30 hours to complete, B variants required 18 hours. This difference is due to the higher number of HWFT checks implemented for B variants.

The results of the simulation are shown in Table 16.2.

80 HWFT’s UNDET and OTH classifications are opposite to CBG’s due to their complementary coverage.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Table 16.2: Results of HWFT simulation

<table>
<thead>
<tr>
<th>Program</th>
<th>DET</th>
<th>OTH</th>
<th>UNDET</th>
<th>UNK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_A</td>
<td>47</td>
<td>531</td>
<td>418</td>
<td>4</td>
</tr>
<tr>
<td>Krider_B</td>
<td>192</td>
<td>515</td>
<td>289</td>
<td>4</td>
</tr>
<tr>
<td>Prog_A</td>
<td>78</td>
<td>286</td>
<td>498</td>
<td>138</td>
</tr>
<tr>
<td>Prog_B</td>
<td>186</td>
<td>309</td>
<td>397</td>
<td>108</td>
</tr>
</tbody>
</table>

Table 16.2 shows HWFT detected between 4.7% and 19.2% of PC errors vectoring program-flow within the programmed ROM locations. This lower than expected result (the maximum calculated rate is between 19.9% and 51.5% based on the program’s size in proportion to the physical code memory size – see Table 16.1) shows HWFT’s error detection rate is less than 100% within its coverage. To be exact, the percentage of detected detectable errors is 23.6%, 90.1%, 15.5% and 36.1% for Krider_A, Krider_B, Prog_A and Prog_B respectively. This means the error detection rates can vary significantly.

Some of the undetected errors will be overwritten by PC writes. As a result, they will not affect program-flow, but were nevertheless classified as UNDET. Hence, UNDET’s value will be less than that shown in Table 16.2. The number of OTH-classified errors is less than the expected 801, 787, 496 and 485 for Krider_A, Krider_B, Prog_A and Prog_B respectively. This is also attributed to PC writes overwriting some injected errors, and some erroneous branches to unprogrammed code memory locations resuming execution within the programmed code memory region within the 500 additional clock-cycle limit. A few injected errors in the latter situation may have been detected by HWFT.

The number of detected errors increases by 382% and 233% (based on the detected detectable errors) for Krider and Prog respectively when more HWFT checks are implemented. Hence, it can only be concluded that the error detection rate is expected to increase with the number of checks implemented.

Although the increase in HWFT checks (see Table 16.3) is roughly proportional to the number of detected errors, it is not possible to conclude that this is always the case. The error detection rate depends on many factors, such as the program type (e.g. straight line code or time-based) and where the checks are placed.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Prog recoded over 100 UNK-classified errors as compared to 4 for Krider. Upon inspecting Prog’s simulation log, it was determined that almost all of these errors were caused by the execution of the RETI instruction (return from ISR) when the processor was not executing an interrupt service routine.

Table 16.3 shows the overheads incurred from HWFT’s implementation. The code size and execution overhead (in cycles) are based on the original version of Krider and Prog (without HWFT), shown in the ‘Orig.’ column.

<table>
<thead>
<tr>
<th>Program</th>
<th>HWFT checks</th>
<th>Code size</th>
<th>Cycles to breakpoint&lt;sup&gt;81&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Orig.</td>
<td>HWFT</td>
</tr>
<tr>
<td>Krider_A</td>
<td>8</td>
<td>748</td>
<td>815</td>
</tr>
<tr>
<td>Krider_B</td>
<td>26</td>
<td>748</td>
<td>879</td>
</tr>
<tr>
<td>Prog_A</td>
<td>16</td>
<td>1935</td>
<td>2063</td>
</tr>
<tr>
<td>Prog_B</td>
<td>33</td>
<td>1935</td>
<td>2111</td>
</tr>
</tbody>
</table>

Table 16.3: HWFT’s code memory and execution overheads

When HWFT is implemented, a code memory overhead of between 6.6% and 17.5% is recorded. As expected, the code memory overhead is higher for programs implementing HWFT checks at more than 100% coverage (B variants). Krider’s implementation overhead almost doubled due to its small program size, with the quadrupling of the number of HWFT checks. On the other hand, Prog’s increase was less than 3% with a doubling of the number of checks due to its larger program size.

Although HWFT’s overhead could be considered significant, especially for smaller programs, they are much lower than FT’s implementation. As a comparison, Prog’s implementation overhead for FT (100% coverage – see Table 4.4 and Table 7.7) was more than 30%, which is about five times higher than its HWFT implementation. This significant reduction is due to each HWFT check only requiring three bytes (about 1/7th the size of FT checks).

It was concluded in Part One that NF was the better technique than FT due to the reduction of Safe Locations. The reason for SL’s reduction is – apart from the first byte – all bytes of each FT check is categorised as UL or DL. Although this situation still applies to HWFT checks,

<sup>81</sup> Both variants of each program was simulated to a specified location.
Techniques intended to reduce the impact of program-flow errors on embedded systems

the amount of UL/DL each HWFT check produces is just two. HWFT still reduces SL, but the reduction is much less than with FT.

Referring to Table 16.3, there is less than a 3% execution overhead when HWFT checks are implemented. This is not surprising: each HWFT check only requires two clock-cycles. In comparison, each FT check requires, on average, of 15 clock-cycles: HWFT checks are approximately seven times more efficient. The number of clock-cycles and code memory required for loading the token is the same for FT and HWFT: three bytes and two clock-cycles. This overhead may be similar to WDT's if its refreshes are located at similar code memory locations to HWFT's checks.

HWFT's error detection rate can be significantly increased if it is implemented at the instruction level. In this case, all program-flow branches – not only those due to function calls – can implement HWFT checks, which is what SM does. This is possible with HWFT since each check is merely a single instruction. The minor disadvantage is the inevitably increase in program size and processing time.

Although HWFT's principle is similar to SM's (especially for assigned signatures), HWFT's major advantage is that it does not require special instructions for signature assignments and checks. This means the processor core and development tools do not require modifications, which potentially leads to significant cost reduction.

Like NF, FT and CBG, HWFT is able to detect program-flow errors some time after they occur. It also detects program-flow errors due to the corruption of other program-flow related registers, not just PC corruption as with ST, ST2 and BCT.

<table>
<thead>
<tr>
<th>Program</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min.</td>
</tr>
<tr>
<td>Krider_A</td>
<td>2</td>
</tr>
<tr>
<td>Krider_B</td>
<td>2</td>
</tr>
<tr>
<td>Prog_A</td>
<td>3</td>
</tr>
<tr>
<td>Prog_B</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 16.4: HWFT's error detection latency

The statistics of HWFT's error detection latency is shown in Table 16.4. As with CBG's simulation, the error signal (FTerro) is monitored by the script, instead of the error flag.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Hence, it is not surprising that latencies as low as two clock-cycles were achieved. It is not possible to determine the relationship between the number of HWFT checks and the average latency due to the many factors (e.g. where the checks are implemented) that are involved. This is shown by the large standard deviation that was recorded. The only conclusion that can be made about HWFT’s latency is that on average, it will be shorter than program’s implementing FT (assuming identical implementation) due to its lower overheads.

Since HWFT is similar to FT, its implementation can also be tedious and error prone, but it is certainly easier than WDT’s (for similar implementations). On the hand, if HWFT is implemented at the instruction level, a tool can be developed to automatically insert the checks. This can tremendously decrease implementation time – which translates to overall cost reduction – and can reduce the risk of implementing the checks wrongly.

Table 16.5 summarises the comparison between FT and HWFT.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>FT</th>
<th>HWFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection coverage</td>
<td>Only within programmed code memory locations</td>
<td>Only within programmed code memory locations</td>
</tr>
<tr>
<td>Detection rate (of detectable errors)</td>
<td>Implementation dependent (much less than 100%)</td>
<td>Potentially better than FT</td>
</tr>
<tr>
<td>Maximum detection latency</td>
<td>Longest between FT checks</td>
<td>Longest between HWFT checks. Lower than FT with identical implementation.</td>
</tr>
<tr>
<td>Software implementation</td>
<td>TTedious and error prone</td>
<td>Less tedious and error prone than FT</td>
</tr>
<tr>
<td>Hardware implementation</td>
<td>N/A</td>
<td>Minor die size increase</td>
</tr>
<tr>
<td>Time to implement</td>
<td>Substantial</td>
<td>Substantial</td>
</tr>
<tr>
<td>Portability</td>
<td>No problem</td>
<td>No problem</td>
</tr>
<tr>
<td>Configurability</td>
<td>Fully configurable</td>
<td>Fully configurable</td>
</tr>
<tr>
<td>System resources (normal condition)</td>
<td>Potential to double final program size.</td>
<td>Negligible</td>
</tr>
<tr>
<td>Processing overheads</td>
<td>Heavy especially with many function calls</td>
<td>Negligible</td>
</tr>
</tbody>
</table>

Table 16.5: Comparison between FT and HWFT

16.6 Implementation
The results of the simulations have shown that HWFT does detect errors, while complementing CBG’s error coverage: an issue that should be considered especially with
Techniques intended to reduce the impact of program-flow errors on embedded systems

large program sizes. Moreover, HWFT has been shown to be significantly better in many respects than FT. Therefore, as with CBG, HWFT has been implemented in VHDL – according to TE-51’s framework – to determine if it can be feasibly implemented (Listing 16.2).

```vhdl
entity te51FT is
  port(
    clk: in STD_LOGIC; -- clock
    rst: in STD_LOGIC; -- reset
    csp: in te51stateT; -- CSP input
    EDCi: in STD_LOGIC_VECTOR(7 downto 0); -- Error detection/correction
    FTvali: in STD_LOGIC_VECTOR(7 downto 0); -- HWFT token register
    FTchki: in STD_LOGIC_VECTOR(7 downto 0); -- HWFT check register
    FTckwri in STD_LOGIC; -- flag to denote FTtoki write
    FTerro: out STD_LOGIC -- Error detection flag
  );
end te51FT;

architecture bhv of te51FT is
  signal FTen:  STD_LOGIC
  signal CMP1:  STD_LOGIC
  signal CMP:  STD_LOGIC
  signal FTerr:  STD_LOGIC
  alias FTenO: STD_LOGIC is EDCi(1)
  alias FTenl: STD_LOGIC is EDCi(5)

  entity te51FT
begin
  process(elk, rst)
  begin
    if rst='1' then
      FTen <= '0'; FTerr <= '0'; CMP <= '0';
    elsif rising_edge(elk) then
      case csp is
        when C1S1P2 | C2S1P2 =>
          FTerr <= FTen and CMP;
        when C1S6P1 | C2S6P1 =>
          FTerr <= '0';
        when others =>
          null;
      end case;
      if FTckwri = '1' then
        CMP <= CMP1;
      end if;
      if (FTen0 = '0') and (FTenl = '1') then
        FTen <= '1';
      elsif (FTen0 = '1') and (FTenl = '0') then
        FTen <= '0';
      end if;
    end if;
  end process;
  CMPl <= '1' when (FTvali /= FTchki) else '0';
  FTerro <= FTerr;
end architecture;
```

Listing 16.2: VHDL source for Hardware Function Token's implementation

HWFT requires seven input signals/bus: four from the SFRs and the remainder from the clock, reset and sub-cycle state information (**CSP**). HWFT has one output: the error signal.
Techniques intended to reduce the impact of program-flow errors on embedded systems

HWFT was successfully synthesized according to TE-51's framework, although it had not been integrated with TE-51 at this stage. Based on XST’s report, HWFT’s gate count — without the SFRs — is equivalent to only 40 gates. When the SFR registers are considered, the equivalent gate count will be approximately 192 gates, which is about 0.7% of TE-51. As a result, HWFT’s implementation is feasible.

16.7 Conclusion

HWFT is capable of detecting program-flow errors. Its effectiveness is governed by the same issues as FT: the number of HWFT checks and their placement. The main advantage of HWFT over FT is that it requires far less code memory: this leads to a reduction in error detection latencies and processor overheads. Since HWFT’s checks are single instructions, tools can be developed to automatically implement the checks at the assembly level (as with SM). Unlike SM, HWFT does not require changes to the instruction set, or the development tools.

HWFT’s implementation overhead is the smallest of all the techniques discussed in this thesis (less than 1% of TE-51’s size). Hence, this technique should be feasible for most embedded systems.

---

82 The overheads on the processor’s side (e.g. additional multiplexor paths) are not considered here.
Techniques intended to reduce the impact of program-flow errors on embedded systems

SecurePorts 17

ST, ST2, BCT, CBG, HWFT and most other error detection techniques have one thing in common: they only detect and/or correct program-flow errors after they happen. Although low (or zero) latency techniques such as ST and PM may prevent processor state changes between error occurrence and detection, no technique can prevent this all the time.

Processor state changes between error occurrences and detection may be tolerable on some systems, but – particularly in safety-related and safety-critical systems – they must be kept to a minimum. For systems that control hardware (such as hydraulic pistons, antilock brakes, airbags) or communicate with one another (such as distributed automotive systems), it is vital that the interface between them and the outside world – which is the ports – does not change state due to program-flow errors.

In this chapter, a mechanism that may prevent port changes due to program-flow errors – known as SecurePorts (SP) – is discussed and evaluated.

17.1 Overview of SecurePorts

It is not possible to detect every program-flow error, as shown by the results of the simulations carried out in the previous chapters. For program-flow errors that are detected, it is important that these errors are detected before the processor executes ‘critical’ instructions. Hence, SecurePorts aims to prevent port changes before errors (that can be detected) are detected.

It must be emphasized that SP will not detect program-flow (or any other) errors. It is a mechanism that may prevent erroneous port writes before errors are detected by other some other technique.
Techniques intended to reduce the impact of program-flow errors on embedded systems

SP is a proxy module: it sits between the processor core and the port. The signals between the processor core and the port are routed through SP when it is enabled, or bypassed when it is not, as illustrated in Figure 17.1.

![Diagram of normal and SP-implemented port connection](image)

Figure 17.1: Normal and SP-implemented port connection

SP's concept is such: the port must be written\(^3\) twice before its latches are updated. Since two port writes are necessary, error detection and/or correction techniques – placed between the writes – should be able to detect errors before the port is updated.

The number of clock cycles between the ports' first and second write ('write window') is kept low (between 1 and 16 clock-cycles with the current implementation) to reduce the chances of two erroneous writes causing port updates. This possibility is further reduced when the second write must be the complement of the first before a port will be written. This approach eliminates the possibility of program-flow errors (or loops) executing the first or second port write in quick succession (i.e. within the 'write window').

In short, SP will only allow ports to be update when the following conditions are met:

- Two writes happen within a certain time frame.
- The value of the second write is the complement of the first.

---

\(^3\) Port reads are unaffected by SP.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Examples of successful and unsuccessful port writes, based on P1, are shown in Listing 17.1.

```
P1 = 0x45; // successful write
P1 = 0xBA;
P1 = 0xA4; // unsuccessful write
P1 = 0xA4; // not complement
P1 = 0x8D; // unsuccessful write
P1 = 0xA4; // different
P1 = 0x8D; // unsuccessful write
for(int I = 1; I <= 1000; I++); // delay
P1 = ~0x8D; // long delay
```

Listing 17.1: Examples demonstrating SecurePorts

17.2 Description of SP

The block diagram for SP - implemented on Port 1 as an example - is shown in Figure 17.2. As with CBG and HWFT, the colour-inverted blocks are registers in the SFR module. Some signals/flags described in this section are internal to the Control module, and are not shown in Figure 17.2. This includes the multiplexors that connects/disconnects SP from the system (see Figure 17.1).

![Figure 17.2: Block diagram of SecurePorts implemented on Port 1](image)

Referring to Figure 17.2, SP is enabled in the same way as CBG and HWFT: two bits (SPen0 and SPen1) have to be set to the correct state. When SP is enabled (SPen is high) and P1 is written (first write via P1i), its value is loaded into P1A. This brings SP from the 'idle' mode to the 'countdown' mode. In the countdown mode, the counter (CNT) decrements at S1P2 and the control module connects P1i to P1B.
If the second P1 write happens before the counter reaches zero, P1’s value is stored in P1B. SP now goes into ‘compare’ mode, which stops CNT decrementing. One sub-cycle after the second P1 write, SP would write P1A’s value to P1’s latches if, and only if, P1B is a complement of P1A. This is achieved by pulsing the P1wro signal, which mimics the write action from the processor core. If P1B is not a complement of P1A (CMP set high), P1’s latches are not updated and an error would be flagged (SPerro set high). In both cases, SP then reverts to idle mode and CNT is reloaded. CNT’s value is derived from SPCNT, the counter’s SFR.

If P1 is not written (for a second time) by the time CNT reaches zero when SP is in countdown mode, P1’s latches are not written and an error is signalled. SP then reverts to idle mode and CNT is reloaded.

As with CBG and HWFT, the SPerro signal will update the SPerr bit (in EDINT) by the SFR module. It is this bit that is polled by the interrupt system, and not the SPerro signal. SPerro reverts low at S6P1.

### 17.3 Memory mapping

Some of SP’s control registers share the same memory locations as CBG’s and HWFT’s. No registers however, overlap; this allows CBG, HWFT and SP to coexist simultaneously. SP’s memory map is shown in Figure 17.3.
17.4 Sub-cycle level execution

As SP is a complex technique, it requires five sub-cycles. Four of SP’s operations occur between S1P1 and S2P2. At S6P1, the error flag is reset, like CBG and HWFT.

17.4.1 Execution at S1P1

SP’s execution at S1P1 – based on Port 1 – is illustrated in Figure 17.4. Execution starts by detecting if P1 has been written during the last clock-cycle. The write is detected with the help of the processor core: a signal \( p_{iwr1} \) is pulsed for one sub-cycle (S1P1) when P1 has been written. This signal is identical to the \( FT_{ckwri} \) signal for HWFT (see Chapter 16).

When a write is detected, SP determines – via the \( Awr \) and \( Bwr \) flags – if it is the first or second write. If \( Awr \) has not been set, the write value (from \( P_{li} \)) is stored in P1A and \( Awr \) goes high. If \( Awr \) is set (second write), the write value is stored in P1B and \( Bwr \) is taken high.

The main activity at S1P1 is to configure all the flags and update the appropriate registers. Hence, this sub-cycle determines SP’s execution path (see next sub-section) at S1P2.

17.4.2 Execution at S1P2

The bulk of SP’s execution occurs during S1P2, as shown in Figure 17.5. In this diagram, ‘Ph0’ to ‘Ph6’ denotes the possible execution paths based on the flags and registers determined during S1P1.

If SP is not enabled, it executes a reset by setting \( Sprst \). When SP is enabled, the \( Idle \) flag would determine its course of action. If the \( Idle \) flag is set (SP in idle mode), the \( Awr \) flag is
Techniques intended to reduce the impact of program-flow errors on embedded systems

checked to determine if P1A has been written. If P1A was written, the idle flag is reset. CNT is reloaded if P1A was not written (Awr is low).

When SP is not in idle mode, the Bwr flag determines the next course of action. If Bwr is low (P1B not written), CNT would decrement if it is not zero. If CNT is zero (CNTzr high), CNT would not be decremented. Instead, a reset – mentioned in the previous paragraph – is carried out and SPerro is set.

If Bwr is high (P1B was written), either P1’s latches are written with P1A’s value when P1B is the complement of P1A, or SPerro is set when P1B is not the complement of P1A. In both cases, SP is reset.

Figure 17.5: SP’s execution at sub-cycle S1P2

17.4.3 Execution at S2P1, S2P2 and S6P1
Sub-cycle S2P1 deals with resetting SP and updating P1. This reset only happens if the SPRst flag was set at S1P2 (path 0, 3, 5 and 6 – see Figure 17.5).
Figure 17.6 shows the execution at S2P1. If SP is not to be reset, SP would determine if P1's latches need to be updated. If a reset was signalled, SP would reset the reset flag (SPrst), reload CNT, set the Idle flag and reset the Awr and Bwr flags.

If P1's latches are to be updated, P1's write signal (P1wro) is taken high. This signal latches P1A’s value into the P1 latches.

The only action that occurs at S2P2 and S6P1 is the resetting of the P1wro and SPerro flags respectively. This occurs even if P1wro and SPerro have not been set.

17.5 Simulation
SP is not an error detection technique; hence, it is not simulated like CBG or HWFT. Instead, SP is tested for correctness by checking various combinations of P1 writes.

17.5.1 Test program
The test program – TestSP – was written with the Keil C51 compiler. This simple program, partly shown in Listing 17.2, test SP with various P1 write combinations. P1 writes are alphabetically referred to in the remark. For this test, it is assumed that the default CNT’s value of four is used to start with. The ‘delay’ statement (e.g. between ‘E’ and ‘F’ P1 write) inserts an eight-clock-cycle delay.

```
P1 = 0x55; // (A)
P1 = ~0x55; // (B)
P1 = 0xAA; // (C)
P1 = 0x0F; // (D)
P1 = 0x0F;
DELAY;
P1 = 0x0F; // (E)
P1 = 0xF0; // (F)
```
Techniques intended to reduce the impact of program-flow errors on embedded systems

Techniques to reduce the impact of program-flow errors on embedded systems...

Listing 17.2: SecurePort test program

17.5.2 Results

The results of the test are shown in Table 17.1.

Port writes are grouped according to the results they produce. Invalid port writes occur due to the port writes not complementing each other, or the second port write occurred outside the write window (CNT decremented to zero). For the latter condition, two separate port writes are recorded.

<table>
<thead>
<tr>
<th>P1 write</th>
<th>Results</th>
<th>Remark (when not valid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>C, D</td>
<td>Not valid</td>
<td>P1 writes not complement</td>
</tr>
<tr>
<td>E</td>
<td>Not valid</td>
<td>P1 second write not in time</td>
</tr>
<tr>
<td>F, G</td>
<td>Not valid</td>
<td>P1 writes not complement</td>
</tr>
<tr>
<td>H</td>
<td>Not valid</td>
<td>P1 second write not in time</td>
</tr>
<tr>
<td>I, J</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>K, L</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Not valid</td>
<td>P1 second write not in time</td>
</tr>
<tr>
<td>N</td>
<td>Not valid</td>
<td>P1 second write not in time</td>
</tr>
</tbody>
</table>

Table 17.1: Results of SecurePort test

The first write pair ('A' and 'B') is valid. This is expected: they are written with complemented values within four clock-cycles.

The second pair ('C' and 'D') is invalid as they are not complementary values.
Since no port write occurred within four clock-cycles after the ‘E’ write, this write is invalid.

The ‘F’ write – initially meant to be the second port write for ‘E’ – is considered the first port write. Hence, the ‘G’ write will be considered the second write for ‘F’. This port write pair is invalid since they do not complement each other.

The ‘H’ write – initially meant to be the second port write for ‘G’ – is invalid as no port write happens before CNT underflows.

Between the ‘H’ and ‘I’ writes, CNT’s was increased to 14 clock-cycles. Due to this increase, the delay between the ‘I’ and ‘J’ writes no longer cause CNT to reach zero. Hence, the ‘I’ and ‘J’ write pair is valid.

CNT’s value was altered between the first and second P1 write for the ‘K’ – ‘L’ pair. Since CNT only reload its counter after SP is reset, or when it is in the idle mode, CNT is not loaded with the new value before the ‘L’ write is detected (or an erroneous write is detected). Hence, the ‘K’ – ‘L’ port write pair is valid.

CNT has reverted to four clock-cycles by the time the ‘M’ write occur. Hence, the ‘M’ write is invalid as CNT reaches zero before the second write (‘N’) occurred.

The ‘N’ write – which was meant to be the second write for the ‘M’ – ‘N’ pair, is invalid, as the second write is not detected within four clock-cycles.

17.5.3 Discussion
The test carried out shows that SP works according to its description. Since SP only controls the data and write signals to registers, it does not interfere with register reads. SP also works without needing any delay between port writes as shown in the test for ‘J’ and ‘K’ writes.

SP is easy to implement: all that is needed is for SP to be enabled, and another write of complementary value carried out after each port write. It is vital to note that only one port

---

84 If the delay had not been inserted between ‘E’ and ‘F’, an error would not have occurred and the ‘G’ and ‘H’ write pair would be correct.
write is necessary when SP is disabled. The complementary write can cause problems when SP is disabled.

Since two writes are needed to update a port, SP will decrease the maximum write rate to a port by 50%. This is however, the worst-case scenario: port are never written anywhere near this rate. Hence, the processor overhead for SP is usually minimal.

A potentially problematic situation involves the ports' update latency. Since the window between each port write could be up to 16 clock-cycles (in this implementation), the latency between the first port write and the port actually being updated, may cause problems with fast response situations (such as inter processor communications). This issue can be resolved by reducing the duration of the 'write window' and/or taking such latencies into consideration during the design stage.

SP's implementation is not limited to the I/O ports. Since this module merely acts as a proxy between two (or more) processor components, it could be used to increase the processor-peripheral communication immunity, for example. This would be beneficial in many systems, such as implementing secure timing units for scheduler-based designs. SP could even be used to implement more secure RAM locations, or more secure communication channels when implemented with the USART's output buffer (SBUF). In designs that use one (or more) ports to select the desired memory bank, SP offers the protection against erroneous memory bank switches.

It must be emphasised again that SP does not attempt to detect errors. What it does is to provide a mechanism where error detection can take place before a register is updated. Hence, the ability to detect errors before register changes lies solely with the effectiveness of the error detection techniques. SP will also not prevent direct register corruption due to induced voltage spikes, ionising radiation and other similar phenomenon.

It can be argued that SP's implementation may barely reduce a register's susceptibility to errors when HWFT checks are carried out immediately before register writes since the likelihood of program-flow errors 'landing' between the check and register write is very slim. SP however, should be used in conjunction with error detection techniques that spans over many clock-cycles. For example, when FT checks are implemented between the first and second register writes (HWFT checks should also be implemented in this manner), corruption
Techniques intended to reduce the impact of program-flow errors on embedded systems

of the FT check (i.e. program-flow ‘landing’ somewhere within it) will prevent updates to an
SP-enabled register, although the FT check may fail to detect the program-flow error. The
source of the two writes can also be derived differently. This may prevent errors occurring
along one or more parallel calculation paths from updating a register, and at the same time
detecting the error.

Since SP will only work correctly when all the flags and registers are at specific values, the
probability of any form of errors causing SP to write registers prematurely may be slim.
Errors causing its flags and registers to change will potentially cause SP to detect the error.
However, an assessment on SP’s vulnerability in this sense has not been carried out.

17.6 Implementation
As with CBG and HWFT, SP is implemented in VHDL according to TE-51’s framework to
determine if it can be feasibly implemented. SP’s implementation, based again on Port 1, is
shown in Listing 17.3.

```vhdl
entity te51SecurePort is
  port (  
    elk: in STD_LOGIC; -- clock  
    rst: in STD_LOGIC; -- reset  
    csp: in te51stateT; -- csp  
    EDCi: in STD_LOGIC_VECTOR(7 downto 0); -- SFR input  
    SPNT1: in STD_LOGIC_VECTOR(7 downto 0);  
    Pli: in STD_LOGIC_VECTOR(7 downto 0); -- P1 input  
    Plwri: in STD_LOGIC; -- P1 write enable input  
    Pl: out STD_LOGIC_VECTOR(7 downto 0); -- PI output  
    Plwro: out STD_LOGIC; -- P1 write enable output  
    SPerro: out STD_LOGIC -- error flag output
  );
end entity te51SecurePort;
architecture bhv of te51SecurePort is
  signal PI A: STD_LOGIC_VECTOR(7 downto 0); -- PI first write  
  signal PlB: STD_LOGIC_VECTOR(7 downto 0); -- PI second write  
  signal PATH: STD_LOGIC_VECTOR(2 downto 0) ; -- execution path  
  signal CNT: STD_LOGIC_VECTOR(7 downto 0); -- 4 bit counter  
  signal CMP: STD_LOGIC; -- compare flag  
  signal CNTrz: STD_LOGIC; -- CNT zero flag  
  signal Spen: STD_LOGIC; -- SP enable flag  
  signal Idle: STD_LOGIC; -- idle flag  
  signal Awr: STD_LOGIC; -- P1 first write flag  
  signal Bwr: STD_LOGIC; -- P1 second write flag  
  signal SPf: STD_LOGIC; -- SP enable flag  
  signal Puid: STD_LOGIC; -- update P1  
  alias Sper: STD_LOGIC is EDCi(2); -- error flag  
  alias Spen0: STD_LOGIC is EDCi(6); -- to make life easier
begin
```

212
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
process(clk, rst)
begin
if rst = '1' then
  CNT <= SPCNTi;
  Idle <= '0';
  Awr <= '0';
  Bwr <= '0';
  SPerr <= '0';
  SPrst <= '0';
  PlA <= x"00";
end if;
elsif rising_edge(clk) then
  case CSP is
    when C1S1P1 | C2S1P1 =>
      if Plwr = '1' then
        if Awr = '1' then
          Bwr <= '1';
          P1B <= not (Pli);
        else
          Awr <= '1';
          P1A <= Pli;
        end if;
      end if;
      if Plwri = '1' then
        if Awr = '1' then
          Bwr <= '1';
          P1B <= not (Pli);
        else
          Awr <= '1';
          P1A <= Pli;
        end if;
      end if;
    when C1S1P2 | C2S1P2 =>
      if PATH = "000" then
        SPrst <= '1';
      elsif PATH = "001" then
        CNT <= SPCNTi;
      elsif PATH = "010" then
        Idle <= '0';
      elsif PATH = "011" then
        SPerr <= '1';
        SPrst <= '1';
      elsif PATH = "100" then
        CNT <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(CNT(7 downto 0))-1),8);
      elsif PATH = "101" then
        SPerr <= '1';
        SPrst <= '1';
      elsif PATH = "110" then
        Plupd <= '1';
        SPrst <= '1';
      elsif PATH = "111" then
        Sperr <= '1';
        SPrst <= '1';
      elsif PATH = "100" then
        Plupd <= '1';
        Plwr <= '1';
      end if;
    when C1S2P1 | C2S2P1 =>
      if SPen = '1' then
        SPrat <= '1';
      elsif SPen = '0' then
        no processing, reset SP
      else
        -- S2P1: resetting flags
        SPrat <= '1';
      end if;
    when C1S2P2 | C2S2P2 =>
      Plwr <= '0';
    when C1S6P1 | C2S6P1 =>
      SPerr <= 'O';
    when others =>
      null;
  end case;
  if (SPenO = '0') and (SPenl = '1') then
    SPen <= '1';
  elsif (SPenO = '1') and (SPenl = '0') then
    SPen <= 'O';
  end if;
end if;
end process;
PATH <= "000" when SPen = '0' else "001" when SPen = '1' and Idle = '1' and Awr = '0' else
"010" when SPen = '1' and Idle = '0' and Awr = '1' else
"011" when SPen = '1' and Idle = '0' and Bwr = '0' and CNTzr = '1' else
"100" when SPen = '1' and Idle = '1' and Bwr = '0' and CNTzr = '0' else
"101" when SPen = '1' and Idle = '0' and Bwr = '1' and CMP = '1' else
"110";
CMP <= '1' when PlA /= PlB else '0';
```

213
Techniques intended to reduce the impact of program-flow errors on embedded systems

CNT\_zr <= '1' when CNT = "0000" else '0'; -- CNT zero flag
SP\_erro <= SP\_err;
P\_lo <= P\_1A when SP\_en = '1' else P\_li; -- P\_lo connect to P\_li or SP
PL\_Wro <= PL\_wr when SP\_en = '1' else PL\_wri; -- PL\_Wro connect to PL\_wri or SP
end architecture bhv;

Listing 17.3: VHDL source for SecurePort's implementation

SP requires seven input signals/bus: two from the SFRs, two from the processor core and the remainder from the clock, reset and sub-cycle state information (CSP). SP has three outputs: two to the external register (P1 in this situation) and one for the error signal.

The PATH, CMP and CNT\_zr signals are derived from combinational logic while the rest are register based. As shown in Figure 17.5, the execution paths are generated based on the flags, and enumerated as the PATH signal. This signal is read by the process at S1P2 to determine the course of action.

The multiplexors shown in Figure 17.1 to connect or bypass SP is implemented in the last four lines of Listing 17.3. When SP is not enabled (SP\_en low), P\_li and PL\_wri directly connect to P\_lo and PL\_Wro respectively. With SP enabled however, the connection is no longer direct: P\_lo and PL\_Wro is controlled by SP.

As with CBG and HWFT, the SP\_err bit is set when SP\_erro is read as a '1' by the SFR module at S5P2. This bit is reset at S6P1.

SP was successfully synthesized as a standalone unit (not integrated with TE-51, but developed according to its framework). Without the SFR registers, SP's implementation is equivalent to 723 logic gates, as reported by XST. When the respective registers were included, SP's gate count is approximately 782 gates. SP's equivalent gate count is more than CBG's and HWFT's due to its complexity. Its overhead is roughly 2.7% that of TE-51. As with CBG and HWFT, processor side gate count overhead is not included.

17.7 Acknowledgement

The idea of updating a register only after two writes are carried out within a specified window is the original idea of the author. The initial design called for the use of a separate register –
Techniques intended to reduce the impact of program-flow errors on embedded systems

the shadow register – that has to be written (second write) before the register implementing SP is updated. Mr. Bertram\textsuperscript{85} suggested that only one register – written twice with complemented values to update an SP-implemented register – led to the reduction in the number of SFRs needed. This approach reduced the gate count considerably.

17.8 Conclusion
SecurePorts is a mechanism that will only allow the changing of a port’s state to occur when a specific write sequence is followed. SP is effective in preventing erroneous port changes when used in conjunction with effective error detection and/or correction techniques. SP will not detect errors; it is a mechanism that allows detectable errors to be detected before register updates. SP’s implementation is by no means limited to the I/O ports: it can be tailored for use with almost all registers. Although its implementation size is larger than CBG or HWFT, SP’s footprint is still relatively small.

\textsuperscript{85} of Trenz Electronik GmbH
Techniques intended to reduce the impact of program-flow errors on embedded systems

Benchmarking CBG, HWFT and SP

Code Boundary Guard, Hardware Function Tokens and SecurePorts have been described and evaluated separately in their respective chapters. Although these techniques have been shown – via simulations – to be effective in detecting errors, the true test comes when working under real-world conditions.

In this chapter, CBG, HWFT and SP, which were synthesized as standalone modules (according to TE-51’s framework), are integrated with TE-51. The memory map of TE-51_EDC – as the new core is known – and core modifications to incorporate the techniques, are described in the following section. The second part of this chapter deals with the benchmarking of CBG, HWFT and SP individually. Although this process is similar to the simulations carried out in the previous chapters, the main difference is that they are implemented in hardware, integrated with a real 8051 processor, and evaluated in real-time, which is the closest ‘real-world’ conditions that this project can achieve. In addition, the benchmarking is carried out over a few seconds in real-time, unlike the few microseconds during simulations. The third part of the chapter evaluates the impact of using all these techniques on the same processor.

18.1 Memory mapping

CBG, HWFT and SP’s memory map – shown in Figure 15.2, Figure 16.2 and Figure 17.3 respectively – can be combined without modifications since they do not have overlapping SFR addresses. Their combined SFR map is shown in Figure 18.1. Two additional registers shown in Figure 18.1: EVECL and EVECH; are discussed in the next section. The REDC bit is also discussed there.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Special Function Register EDC (0xC0)  Reset Value: 0x07
Special Function Register CBGL (0xC1)  Reset Value: 0x00
Special Function Register CBGH (0xC2)  Reset Value: 0x00
Special Function Register FTval (0xC3)  Reset Value: 0x00
Special Function Register FTchk (0xC4)  Reset Value: 0x00
Special Function Register EVECL (0xC5)  Reset Value: 0x00
Special Function Register EVECH (0xC6)  Reset Value: 0x00
Special Function Register SPCNT (0xC7)  Reset Value: 0x04
Special Function Register EDINT (0xC8)  Reset Value: 0x00

Figure 18.1: Combined CBG, HWFT and SP SFR map

18.2 Immediate vectoring
As mentioned in Chapter 15, there are two ways to interrupt the processor when an error is detected: 1) by using the interrupt system, or 2) by immediately vectoring (immediate vectoring) to a specified code memory location. The advantages and disadvantages of both options were discussed in Table 15.1. Due to the potentially deadly nature of program-flow errors, the latter option was adopted. Hence, EVECL and EVECH are used to store the code memory address of the vectoring location (low and high byte respectively), which is the starting address of the program-flow error handler.

Immediate vectoring is executed by changing the operands in TE-51’s instruction fetch module to the ‘Long Jump’ instruction. This is similar to the method used by the interrupt system to branch to the interrupt vectors (see Appendix H). The only difference is that a
Techniques intended to reduce the impact of program-flow errors on embedded systems

‘Long Jump’ – instead of ‘Long Call’ – instruction is issued, as it is unnecessary to store the address of the vectored location (this address will be that of the erroneous program-flow). Note: this also means that the PFEH must not end with the RET or RETI instruction to prevent ‘popping’ of the stack and changing the interrupt status. The PFEH should restart the program at a safe location by branching into it.

Immediate vectoring does not have any blocking conditions apart from the fact that it will only happen on instruction boundaries to prevent EMIT errors. If the interrupt occurs while an instruction is being executed, it waits for the current instruction to complete. If a ‘normal’ (high or low priority) interrupt is being serviced by the processor, immediate vectoring takes precedence. Immediate vectoring will also prevent all other interrupts from happening before the REDC bit – described in the following paragraph – is pulsed.

For the ‘normal’ interrupt system, a RETI instruction at the end of the interrupt service routine branches program execution back to the code memory location preceding the interrupt. The RETI instruction also resets the processor’s interrupt level to allow other interrupts to occur. With immediate vectoring however, a RETI instruction cannot be issued. Hence, REDC is used instead. A low to high transition of REDC (by setting this bit in software) signals the processor to end the immediate vectoring state and return to the state that the processor was in before an immediate vectoring process occurred.

18.3 Benchmarking CBG

CBG has been shown to work correctly and effectively under 8051Sim-NG (see Chapter 15). In this section, the synthesized version of CBG is evaluated – with the help of the MCU Testbench – to assess it based on ‘real-world’ situations.

Ideally, such a test should be carried out with equipment capable of irradiating the device-under-test (DUT) with known doses of electromagnetic energy and/or high-energy particles, as described in [Engel et. al. 1996, Marot & Dall'Agnese 1998, Melear 1997a, Nitsch et. al. 2000]. Unfortunately, this project lacks the necessary funds to rent or purchase such equipment, and the time required to carry out such experiments. The alternative was to

\[86\] It is quite probable that the PFEH must be coded in assembly since it may be impossible to prevent high-level languages from inserting ‘RET’ or ‘RETI’ instructions.
Techniques intended to reduce the impact of program-flow errors on embedded systems

simulate the effects of errors induced by EMI and high-energy particles by corrupting TE-51_EDC's registers.

The benchmarking could have been carried out with two identical processors: errors would have then been injected into one processor while the other served as a control [Benso et. al. 1999]. Therefore, the register values of both processors could be compared to determine if an injected error had been overwritten, or other registers have been corrupted. Such a set up was not possible here due to the lack of space on the FPGA (TE-51_EDC required about 60% of the CLBs available).

18.3.1 Test programs
Two test programs – 'Krider' and 'Prog' – were used in this benchmark. Both test programs had three variations, the same as those used in Chapter 15. The only difference between these two sets of test programs is in the program-flow error handler. The PFEH has been written to show the contents of EDINT on Port 3 and execute an infinite loop.

The difference between the variants is in the CBGL and CBGH values, which determine CBG’s threshold. To prevent confusion with the other test programs, these variants are prefixed with the error detection method and the variant (e.g. 'Krider_CBG_A').

<table>
<thead>
<tr>
<th>Programs</th>
<th>End address (hex)</th>
<th>Code/ROM ratio 4kB</th>
<th>Code/ROM ratio 64kB</th>
<th>CBG’s threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_CBG_A</td>
<td>2DF</td>
<td>0.179</td>
<td>0.011</td>
<td>2E1</td>
</tr>
<tr>
<td>Krider_CBG_B</td>
<td>2DF</td>
<td>0.179</td>
<td>0.011</td>
<td>7FF</td>
</tr>
<tr>
<td>Krider_CBG_C</td>
<td>2DF</td>
<td>0.179</td>
<td>0.011</td>
<td>7FF</td>
</tr>
<tr>
<td>Prog_CBG_A</td>
<td>72A</td>
<td>0.448</td>
<td>0.028</td>
<td>72C</td>
</tr>
<tr>
<td>Prog_CBG_B</td>
<td>72A</td>
<td>0.448</td>
<td>0.028</td>
<td>7FF</td>
</tr>
<tr>
<td>Prog_CBG_C</td>
<td>72A</td>
<td>0.448</td>
<td>0.028</td>
<td>7FF</td>
</tr>
</tbody>
</table>

Table 18.1: CBG benchmarking test program statistics

Table 18.1 shows the last programmed location, the ratio of programmed to the physical (4kB) and addressable (64kB) code memory, and CBG’s threshold values. The end address for all variants of each program is the same.

18.3.2 Benchmarking procedure
The benchmarking procedure is very similar to the simulations carried out in Chapter 15 (see Figure 18.2). Each test cycle starts with resetting TE-51_EDC, followed by the generation of
Techniques intended to reduce the impact of program-flow errors on embedded systems

the erroneous cycle, erroneous sub-cycle and erroneous PC’s value. ECY, ESC and EPC values are randomly generated and downloaded to the MCU Testbench’s error injection module. The error is injected between the 1\textsuperscript{th} and 10,000,000\textsuperscript{th} clock-cycle.

Ideally, the error injection process should be asynchronous: it should not be tied to the edge of TE-51_EDC’s clock pulses, nor the clock feeding the MCU Testbench. This is not achievable with the MCU Testbench, however.

Each benchmarking cycle starts by configuring the error injection module, followed by executing TE-51_EDC in run mode. During the execution phase, the benchmarking script continuously polls a flag to determine if the error has been injected. Once this occur, TE-51_EDC stops (the error injection module was configured to stop execution upon error injection) and the second phase begin.

![Flow-chart](image)

Figure 18.2: CBG’s benchmarking procedure flow-chart

In the second phase, the benchmark script steps TE-51_EDC for up to 10 additional clock-cycles. After each step, the content of EDINT is read to determine if CBG has detected the error. If it has, the benchmark cycle restarts. If CBG had not detected the program-flow error after the additional clock-cycles, the error is classified as ‘other’ (OTH) if the PC is within
Techniques intended to reduce the impact of program-flow errors on embedded systems

the programmed code memory location, or undetected (UNDET) if otherwise. Once the errors are classified, the benchmarking process restarts. This carries on for 1000 cycles.

18.3.3 Results and discussion

The benchmarking took between 90 and 117 minutes on a Pentium 350MHz machine with 128MB of RAM running Windows 2000. CBG’s benchmarking results are shown in Table 18.2.

Note: the ‘Error range’ (column 2 and 3) of Table 18.2 is classified based on the location of the last programmed code memory location of each variant. If EPC’s value lies within the programmed code memory region, it is classified under ‘Outside’, as in ‘outside CBG’s error coverage’, and vice versa.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Error range</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Within</td>
<td>Outside</td>
</tr>
<tr>
<td>Krider_CBG_A</td>
<td>988</td>
<td>12</td>
</tr>
<tr>
<td>Krider_CBG_B</td>
<td>989</td>
<td>11</td>
</tr>
<tr>
<td>Krider_CBG_C</td>
<td>983</td>
<td>17</td>
</tr>
<tr>
<td>Prog_CBG_A</td>
<td>971</td>
<td>29</td>
</tr>
<tr>
<td>Prog_CBG_B</td>
<td>973</td>
<td>27</td>
</tr>
<tr>
<td>Prog_CBG_C</td>
<td>976</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 18.2: Results of CBG’s benchmarking

For A variants, none of the injected errors that fell outside CBG’s coverage region was UNDET-classified. This confirms CBG’s 100% error detection rate for detectable program-flow errors. When CBG’s limit was increased, as in B and C variants, the number of UNDET-classified errors increased as expected. Since the increase is proportional to the ratio of the variant’s size to the addressable code memory locations, and not the physical code memory locations, this proves that – unlike NF – CBG’s effectiveness is not affected by memory aliasing.

About half the number of injected errors for C variants should have been UNDET-classified since CBG’s threshold is half of the addressable code memory. Instead, less than 32% of injected errors were. This anomaly, also shown in the simulations (see Chapter 15), is due to some injected errors being OTH-classified. The possible situations for OTH-classified errors were discussed in Chapter 15.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Programs</th>
<th>DET Sim.</th>
<th>DET Ben.</th>
<th>OTH Sim.</th>
<th>OTH Ben.</th>
<th>UNDET Sim.</th>
<th>UNDET Ben.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_CBG_A</td>
<td>845</td>
<td>841</td>
<td>155</td>
<td>159</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Krider_CBG_B</td>
<td>851</td>
<td>849</td>
<td>138</td>
<td>137</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>Krider_CBG_C</td>
<td>469</td>
<td>449</td>
<td>160</td>
<td>240</td>
<td>371</td>
<td>311</td>
</tr>
<tr>
<td>Prog_CBG_A</td>
<td>857</td>
<td>832</td>
<td>143</td>
<td>168</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prog_CBG_B</td>
<td>843</td>
<td>845</td>
<td>157</td>
<td>153</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Prog_CBG_C</td>
<td>446</td>
<td>444</td>
<td>159</td>
<td>240</td>
<td>395</td>
<td>316</td>
</tr>
</tbody>
</table>

Table 18.3: Comparison between simulated and benchmarked results for CBG

An important aspect of the simulation (see Table 15.3) and benchmarking process is the close correlation between their results, especially for DET-classified errors (see Table 18.3). The only significant differences between the simulation and benchmarking process are the OTH- and UNDET-classified errors for C variants. This difference favours the benchmarking process however, which is a better representation of ‘real-world’ conditions.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_CBG_A</td>
<td>4</td>
</tr>
<tr>
<td>Krider_CBG_B</td>
<td>4</td>
</tr>
<tr>
<td>Krider_CBG_C</td>
<td>4</td>
</tr>
<tr>
<td>Prog_CBG_A</td>
<td>4</td>
</tr>
<tr>
<td>Prog_CBG_B</td>
<td>4</td>
</tr>
<tr>
<td>Prog_CBG_C</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 18.4: CBG’s error detection latency

CBG’s error correction latency is about four clock-cycles with a very low standard deviation. Although this is higher than the simulated result (average of 8 sub-cycles), error correction is taken into account. The simulation process (see Chapter 15) also checked the CBGerro signal, which is set almost a clock-cycle earlier than the CBGerr bit. Note: Table 18.4’s latency is quoted in clock-cycles, which means a difference of up to 12 sub-cycles is possible between the simulated and benchmarked results.

When immediate vectoring is employed, vectoring occurs at the next instruction boundary when CBG’s error bit (CBGerr) is set. The vectoring process itself takes two clock-cycles (equivalent to an ‘LJMP’ instruction). If the interrupt system had been used as the exception
Techniques intended to reduce the impact of program-flow errors on embedded systems

handling mechanism (i.e. as in Chapter 15’s implementation), the error correction latency will be dependent on the interrupt condition, making it unpredictable. This is not the case with immediate vectoring.

18.3.4 Conclusion
CBG has been shown to detect practically all detectable program-flow errors. Moreover, errors are detected with minimal state changes to the processor. CBG’s benchmarking results also correlate well with their simulation results.

CBG has a short error detection latency, virtually non-existent processing and code memory overhead, and small implementation footprint. These characteristics should make it very suitable for most embedded applications. In addition, CBG’s implementation only requires trivial program modifications, and can be implemented without any modifications to all commercially available development tools.

18.4 Benchmarking HWFT
The benchmarking procedure for HWFT is very similar to CBG’s. Hence, only the differences are discussed here.

18.4.1 Test programs
As with HWFT’s simulation, ‘Krider’ and ‘Prog’ are used in this benchmark. Each test program had two variants employing 100% and more than 100% coverage for the function calls (‘A’ and ‘B’ variants respectively). The PFEH is the same as CBG’s. Table 18.5 shows the statistics of the test programs for HWFT’s benchmark.

<table>
<thead>
<tr>
<th>Programs</th>
<th>End address (hex)</th>
<th>Code/ROM ratio</th>
<th>FT checks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4kB</td>
<td>64kB</td>
</tr>
<tr>
<td>Krider_HWFT_A</td>
<td>311</td>
<td>0.192</td>
<td>0.012</td>
</tr>
<tr>
<td>Krider_HWFT_B</td>
<td>34A</td>
<td>0.206</td>
<td>0.013</td>
</tr>
<tr>
<td>Prog_HWFT_A</td>
<td>79E</td>
<td>0.476</td>
<td>0.030</td>
</tr>
<tr>
<td>Prog_HWFT_B</td>
<td>7D1</td>
<td>0.489</td>
<td>0.031</td>
</tr>
</tbody>
</table>

Table 18.5: HWFT benchmarking test program statistics
Techniques intended to reduce the impact of program-flow errors on embedded systems

18.4.2 Benchmarking procedure
The benchmarking procedure is the same as CBG's with the exception that up to 500 (instead of 10) additional clock-cycles are simulated after the error is injected (phase two). The increase is necessary to ensure detection between the longest HWFT checks is possible.

Note: if the PC is within the programmed code memory locations at the end of the additional cycles, the error is classified as UNDET, or OTH if it is outside this region. As with HWFT's simulation, this is opposite to CBG's simulation and benchmarking.

18.4.3 Results and discussion
The benchmarking for each variant required approximately 310 minutes on a Pentium 350MHz machine with 128MB of RAM running Windows 2000. HWFT's benchmarking results are shown in Table 18.6. The six-fold increase in benchmarking time, as compared with CBG's, is attributed to the higher number of additional clock-cycles executed during the second phase. These additional clock-cycles are step-executed, which is a relatively slow process with the MCU Testbench.

As with Table 18.2, the 'Error range' (column 2 and 3) is classified based on the location of the last programmed code memory location. If EPC's value lies within the programmed code memory region (memory aliasing considered), it is classified as 'Within', as in 'within HWFT's error coverage', and vice versa.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Error Range</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Within</td>
<td>Outside</td>
</tr>
<tr>
<td>Krider_HWFT_A</td>
<td>168</td>
<td>832</td>
</tr>
<tr>
<td>Krider_HWFT_B</td>
<td>221</td>
<td>779</td>
</tr>
<tr>
<td>Prog_HWFT_A</td>
<td>477</td>
<td>523</td>
</tr>
<tr>
<td>Prog_HWFT_B</td>
<td>497</td>
<td>503</td>
</tr>
</tbody>
</table>

Table 18.6: Results of HWFT's benchmarking

There are a few important differences between HWFT and CBG's results. As mentioned in Chapter 16, HWFT is affected by memory aliasing when the physical code memory is less than that addressable. Hence, the number of errors falling within programmed code memory locations is similar to the ratio of the programmed code memory locations to the amount of physical code memory implemented, as shown in Table 18.6 (second and third column).
Techniques intended to reduce the impact of program-flow errors on embedded systems

Between 12.1% (Prog_HWFT_A: 58 out of 477) and 63.8% (Krider_HWFT_B: 141 out of 221) of detectable injected errors were detected. This conclusively shows that HWFT’s error detection rate is less than 100%. As with the simulation results, when UNDET-classified errors are compared with the number of errors inside HWFT’s error coverage, the higher than expected UNDET rate is due to undetected injected errors and those overwritten by PC writes; hence, not affecting program-flow (but nevertheless UNDET-classified). The lower than expected amount of OTH-classified errors, is also partially due to PC writes overwriting some injected errors. Both issues were discussed in Chapter 16.

Table 18.6 shows that increasing the number of HWFT checks does increase the error detection rate. To be exact, the error detection rates increased by 249% and 316% (based on the detected detectable errors) for Krider and Prog, although the number of HWFT checks increased by 19 and 18 respectively. This shows that there is no direct correlation between the number of HWFT checks and the error detection rate. This strengthens the conclusions, made in Chapter 16, that HWFT’s error detection rate is expected to rise with the increase in the number of checks employed.

<table>
<thead>
<tr>
<th>Programs</th>
<th>DET</th>
<th>OTH</th>
<th>UNDET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_HWFT_A</td>
<td>47</td>
<td>43</td>
<td>531</td>
</tr>
<tr>
<td>Krider_HWFT_B</td>
<td>192</td>
<td>141</td>
<td>515</td>
</tr>
<tr>
<td>Prog_HWFT_A</td>
<td>78</td>
<td>58</td>
<td>286</td>
</tr>
<tr>
<td>Prog_HWFT_B</td>
<td>186</td>
<td>190</td>
<td>309</td>
</tr>
</tbody>
</table>

Table 18.7: Comparison between simulated and benchmarked results for HWFT

The number of detected errors is lower than that obtained during simulation, as shown in Table 18.7, but two of the four variants recorded similar results. The larger difference between the simulation and benchmarking results for Krider_B and Prog_A is mainly due to the simulation’s smaller error injection window. With smaller error injection windows, the dynamic instruction-type profile will be further from the average as compared to one that is produced after one complete software cycle. This is also the reason for the differences between the simulated and benchmark results for OTH- and UNDET-classified errors.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Programs</th>
<th>Latency (cycles)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_HWFT_A</td>
<td>5</td>
<td>133</td>
<td>49.37</td>
<td>42.61</td>
</tr>
<tr>
<td>Krider_HWFT_B</td>
<td>5</td>
<td>141</td>
<td>17.33</td>
<td>16.81</td>
</tr>
<tr>
<td>Prog_HWFT_A</td>
<td>6</td>
<td>298</td>
<td>50.53</td>
<td>55.28</td>
</tr>
<tr>
<td>Prog_HWFT_B</td>
<td>5</td>
<td>476</td>
<td>73.81</td>
<td>84.43</td>
</tr>
</tbody>
</table>

Table 18.8: HWFT’s error detection latency

HWFT’s average error detection latency and range (standard deviation) is much higher than CBG’s, as shown in Table 18.8. This is expected: the latency could be as long as the interval between two HWFT checks. What is surprising is the fact that the average latency for Prog_HWFT_B actually increased as compared to Prog_HWFT_A even though its error detection rate increased. This trend was also shown in Table 16.4. By examining the log file, it was clear that the additional HWFT checks detecting errors that would have been missed by the A variant are far apart, hence the increased average latency.

The latency shown in Table 18.8 also differs significantly from the simulation’s (see Table 16.4). This latency should be a better estimate since the benchmarking process has a much wider error injection window. Note: the standard deviation for HWFT’s benchmarking is significantly lower than the simulation’s.

<table>
<thead>
<tr>
<th>Program</th>
<th>Cycles</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider</td>
<td>815042</td>
<td>-</td>
</tr>
<tr>
<td>Krider_HWFT_A</td>
<td>931824</td>
<td>14.33</td>
</tr>
<tr>
<td>Krider_HWFT_B</td>
<td>952051</td>
<td>16.81</td>
</tr>
<tr>
<td>Prog</td>
<td>480511</td>
<td>-</td>
</tr>
<tr>
<td>Prog_HWFT_A</td>
<td>482605</td>
<td>0.436</td>
</tr>
<tr>
<td>Prog_HWFT_B</td>
<td>482626</td>
<td>0.440</td>
</tr>
</tbody>
</table>

Table 18.9: Execution overhead for test programs employing HWFT

The execution overhead for implementing HWFT is significantly higher than CBG, and proportional to the number of checks employed. This is shown in Table 18.9: the execution overhead for Krider is between 14.3% and 16.8% while Prog’s is approximately 0.4%. When more HWFT checks were implemented, the number of execution cycles increases measurably.
Techniques intended to reduce the impact of program-flow errors on embedded systems

for Krider, but only marginally with Prog. Hence, execution overhead greatly depends on where checks are implemented, which in turn determines how often they are executed.

Krider's overhead is significantly higher its simulated results (see Chapter 16) since the benchmarking process was carried out over a much wider portion of code.

18.4.4 Conclusion

HWFT's benchmarking results further strengthens the fact that it is capable of detecting some - but not all - program-flow errors (within its coverage region). Its error detection rate will probably increase with the number of checks employed, although there is no simple correlation between these parameters. When the checks are employed at the instruction level (similar to Signature Monitoring), HWFT's error detection rate will almost certainly increase.

In general, HWFT's latency is significantly longer than CBG's, but it will be shorter than FT's for identical implementations. Although HWFT's gate count overhead is insignificant, its overall cost may be substantial due the time involved in inserting checks. HWFT's main advantage over FT is its significantly lower execution and code memory overhead, and the potential increase in error detection rate. Hence, where possible, HWFT should be implemented in place of FT.

18.5 Benchmarking SecurePorts

SecurePorts was benchmarked in a different way. For a start, TE-51_EDC was modified to inject an error into SP's PIA register (see Figure 17.2), instead of the PC. The MCU Testbench was also modified to lengthen the signal - meant to corrupt the PC in CBG and HWFT - for SP to treat it like a write from the processor. EPC is used to hold the corrupted PIA's value: only the lower 8-bits are used.

Note: this modified version of TE-51_EDC and the MCU Testbench is only used in this particular benchmark.

18.5.1 Test programs

As with CBG and HWFT, 'Krider' and 'Prog' are the test programs used in the benchmark. Each test program had two variants: A variants uses SPCNT's default value of four while B variants increased it to 15 (maximum). The statistics of the test programs are shown in Table 18.10.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Programs</th>
<th>End address (hex)</th>
<th>Code/ROM ratio</th>
<th>SPCNT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4kB</td>
<td>64kB</td>
</tr>
<tr>
<td>Krider_SP_A</td>
<td>2EC</td>
<td>0.183</td>
<td>0.011</td>
</tr>
<tr>
<td>Krider_SP_B</td>
<td>2EC</td>
<td>0.183</td>
<td>0.011</td>
</tr>
<tr>
<td>Prog_SP_A</td>
<td>729</td>
<td>0.448</td>
<td>0.028</td>
</tr>
<tr>
<td>Prog_SP_B</td>
<td>729</td>
<td>0.448</td>
<td>0.028</td>
</tr>
</tbody>
</table>

Table 18.10: SP benchmarking test program statistics

18.5.2 Benchmarking procedure

The benchmarking procedure is the same as CBG’s and HWFT’s during the first phase. Once the error had been injected (i.e. the P1A register is corrupted), up to a maximum of 50 additional clock-cycles are executed. This is the second phase of the benchmarking process.

Phase two starts by saving P1’s value immediately after the error is injected, but before any step-execution takes place. This value, together with EDINT’s, are checked after each step-execution. If – within the 50 additional clock-cycle limit – the saved P1 differ from its current value, step-execution stops and a ‘P1 error’ (P1_ERR) is recorded. Step-execution also stops if SPerr is set, meaning an error has been detected. This situation is recorded as a ‘detected error’ (DET). Note: the former condition takes precedence.

If the additional 50-cycle limit is reached, the PC is checked to determine if program-flow is within the programmed code memory (OTH-classified), or outside this region (UNDET-classified). The benchmarking cycle then restarts until 1000 runs are completed.

18.5.3 Results and discussion

The benchmarking process took approximately 95 and 105 minutes for A and B variants respectively on a Pentium 350MHz machine with 128MB of RAM running Windows 2000. The time difference is due to the longer ‘write window’ for B variants. SP’s benchmarking results are shown in Table 18.11.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DET</td>
</tr>
<tr>
<td>Krider_SP_A</td>
<td>996</td>
</tr>
<tr>
<td>Krider_SP_B</td>
<td>997</td>
</tr>
<tr>
<td>Prog_SP_A</td>
<td>988</td>
</tr>
<tr>
<td>Prog_SP_B</td>
<td>996</td>
</tr>
</tbody>
</table>

Table 18.11: SP’s benchmarking results
Techniques intended to reduce the impact of program-flow errors on embedded systems

It is clear from Table 18.11 that SP detected 99% of all injected errors, making it extremely effective. This does not mean SP will detect 99% of erroneous register writes. SP is only a mechanism that prevents register state changes; it is up to error detection techniques to detect errors. Looking from another perspective, as long as an error is detected by some other technique, the probability of an SP-enabled register being erroneously written is only one in a hundred.

SP did not receive any UNDET-classified errors. This is not surprising: UNDET-classified errors would have meant SP failed to detect an error: a situation that should be unlikely to occur due to SP’s complex nature.

P1_ERR-classified error rates show an anomaly: the number of P1 errors for Prog fell marginally when SPCNT’s value increased. This is due to the relatively low number of benchmark runs involved. In reality, the number of P1_ERR errors should increase when SPCNT increases. The explanation is such: the second port write must be written before SPCNT decrements to zero. Hence, if SPCNT increases, the time when a port could be written with the conjugate of the first write, increases. This will reduce SP’s statistical error detection rate, but it should be small.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider_SP_A</td>
<td>5</td>
</tr>
<tr>
<td>Krider_SP_B</td>
<td>5</td>
</tr>
<tr>
<td>Prog_SP_A</td>
<td>5</td>
</tr>
<tr>
<td>Prog_SP_B</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 18.12: SP’s error detection latency

The average latency between error injection and detection for A variants is a little short of nine clock-cycles as shown in Table 18.12. This is expected: when the error is injected, it requires about one clock-cycle before SP starts counting down. Five clock-cycles would be needed before SPCNT reaches zero. Hence, it is only by the seventh clock-cycle that SP detects the error, and the respective bit in EDINT is set at the eight clock-cycle. For B variants, the average error detection latency is longer – as expected – since SPCNT was increased to 15.
Techniques intended to reduce the impact of program-flow errors on embedded systems

SP’s implementation size is the highest of the techniques proposed in this thesis. Its larger implementation size is mainly due to the fact that flip-flops are needed to store the parameters. Even so, SP’s approximate gate count of 782 gates is still very small (2.7%) as compared with the gate count for TE-51.

18.5.4 Conclusion
SP has been shown to be 99% effective in preventing register changes before detectable errors are detected. Its error detection latency is proportional to the ‘write window’ governed by SPCNT.

18.6 Combined CBG, HWFT and SP benchmark
In this section, CBG, HWFT and SP are employed concurrently to gauge their overall effectiveness in detecting program-flow errors. This is especially important for CBG and HWFT: they have complementary error detection coverage.

18.6.1 Test programs
As with the previous benchmarks, ‘Krider’ and ‘Prog’ are used. Each test program has two variants: ‘A’ variants with 100% coverage for HWFT checks and ‘B’ variants implementing more than 100% coverage. The test programs are prefixed ‘_ALL’ followed by their variant (‘_A’ or ‘_B’) in this benchmark. Table 18.13 shows the statistics of the test programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>End address (hex)</th>
<th>Code/ROM ratio</th>
<th>HWFT checks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4kB</td>
<td>64kB</td>
</tr>
<tr>
<td>Krider_ALL_A</td>
<td>328</td>
<td>0.197</td>
<td>0.012</td>
</tr>
<tr>
<td>Krider_ALL_B</td>
<td>361</td>
<td>0.211</td>
<td>0.013</td>
</tr>
<tr>
<td>Prog_ALL_A</td>
<td>7A7</td>
<td>0.478</td>
<td>0.030</td>
</tr>
<tr>
<td>Prog_ALL_B</td>
<td>7DA</td>
<td>0.491</td>
<td>0.031</td>
</tr>
</tbody>
</table>

Table 18.13: Test program statistics for the combined benchmark

18.6.2 Benchmarking procedure
The benchmarking procedure is the same as CBG and HWFT’s for the first phase. In the second phase, up to 500 additional clock-cycles are step-executed. After each step-execution, EDINT’s value is read. If EDINT is non-zero – signifying one of the error flags has been set – step-execution stops and the error is classified as DET.
Techniques intended to reduce the impact of program-flow errors on embedded systems

EDINT’s value also determines which technique (or combination of them) detected the error, and the respective hits are recorded. If the additional 500 clock-cycles is reached without the error being detected, the error is classified as OTH if the PC is within the programmed code memory region, or UNDET if it is not. These classifications are only meant to assess CBG. This completes one benchmarking run. Each variant was benchmarked for 1000 runs.

18.6.3 Results and discussion

The benchmarking required approximately 135 minutes on a Pentium 350MHz machine with 128MB of RAM running Windows 2000. The results of this benchmarking process are shown in Table 18.14 and Table 18.15.

<table>
<thead>
<tr>
<th>Program</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DET</td>
</tr>
<tr>
<td>Krider_ALL_A</td>
<td>862</td>
</tr>
<tr>
<td>Krider_ALL_B</td>
<td>870</td>
</tr>
<tr>
<td>Prog_ALL_A</td>
<td>851</td>
</tr>
<tr>
<td>Prog_ALL_B</td>
<td>840</td>
</tr>
</tbody>
</table>

Table 18.14: Results of combined CBG, HWFT and SP benchmarking

From Table 18.14, it is clear that at least 84% of injected errors were detected by one of the error detection techniques. What is important is that none of the injected errors were UNDET-classified, which would have meant CBG’s error detection rate would be less than 100%. All the injected errors that were undetected by CBG, HWFT or SP did not branch program-flow to unprogrammed code memory locations by the end of the additional clock-cycles executed, hence OTH-classified. Not all OTH-classified errors however, would have been overwritten by PC writes. Some would have branched program-flow within programmed code memory locations, and may have been detected by HWFT.

More errors branching program-flow within the programmed code memory region will be undetected (OTH-classified) when larger test programs are used. This is due to the increase of HWFT’s coverage region, and the corresponding decrease in CBG’s, which reduces the number of detectable errors (HWFT’s detection rate is less than 100%).
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected errors (DET)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CBG</td>
<td>Lat.</td>
<td>HWFT</td>
</tr>
<tr>
<td>Krider_ALL_A</td>
<td>858</td>
<td>3.12</td>
<td>4</td>
</tr>
<tr>
<td>Krider_ALL_B</td>
<td>862</td>
<td>3.11</td>
<td>8</td>
</tr>
<tr>
<td>Prog_ALL_A</td>
<td>849</td>
<td>3.82</td>
<td>2</td>
</tr>
<tr>
<td>Prog_ALL_B</td>
<td>824</td>
<td>3.56</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 18.15: Breakdown of CBG, HWFT and SP's detection rate for the combined benchmark

The most interesting aspect of Table 18.15 is the fact that CBG accounted for almost all the injected errors that were detected. This is not a surprise: a majority of the injected errors would have fallen within CBG's coverage region due to the small program sizes. What is less obvious is that although HWFT's coverage is about 20% and 49% for Krider and Prog respectively (due to memory aliasing), hardly any errors were detected by it. This is due to HWFT's higher latency: errors that are within HWFT and CBG's coverage would almost certainly be first detected by CBG. Increasing the number HWFT checks only marginally affected HWFT's error detection rate.

SP did not detect any errors since only the PC was corrupted in this benchmark. There is a slim chance however, that SP may detect a corruption if program-flow erroneously branches to an instruction that writes to PL.

It is also interesting to note that HWFT's latency for Krider decreases when more HWFT checks were employed: by contrast the latency for Prog increased under the same situation. The same was also observed in HWFT's benchmarking (see Table 18.6), and the explanation was given there.

With small programs, CBG alone may be sufficient to achieve effective error detection. For example, a 4kB program will have a statistical error detection rate of 93.75%. With larger programs, both techniques should be implemented as CBG’s coverage region decreases with an increase in code size.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The error detection rate when HWFT and CBG are implemented is:

\[ P(\text{Detect}) = (\text{CBG} / M) + [P(\text{HWFT}) \times (P^* / M)] \]

**Equation 18.1**

where:
- \( \text{CBG} \) = CBG’s value
- \( P(\text{HWFT}) \) = Probability of HWFT detecting errors within its coverage
- \( P^* \) = Program size
- \( M \) = Physical ROM size

*Equation 18.1* assumes a 100% error detection rate for CBG and it detect errors falling on all aliased code memory locations within HWFT’s coverage. HWFT’s error detection probability \( P(\text{HWFT}) \) depends on its implementation, program type and other factors; this can only be estimated via simulation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Cycles</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krider</td>
<td>815042</td>
<td>-</td>
</tr>
<tr>
<td>Krider_ALL_A</td>
<td>902049</td>
<td>10.68</td>
</tr>
<tr>
<td>Krider_ALL_B</td>
<td>936957</td>
<td>14.96</td>
</tr>
<tr>
<td>Prog</td>
<td>480511</td>
<td>-</td>
</tr>
<tr>
<td>Prog_ALL_A</td>
<td>482645</td>
<td>0.44</td>
</tr>
<tr>
<td>Prog_ALL_B</td>
<td>482655</td>
<td>0.45</td>
</tr>
</tbody>
</table>

Table 18.16: Execution overhead for test programs employing CBG, HWFT and SP

The execution overhead incurred when all three techniques were used is negligible. The processing overhead, as shown in *Table 18.16*, is between 10.7% and 15% for Krider and 0.5% for Prog. The increase between A and B variants – especially for Krider – due to the increase in HWFT checks, is mirrored in *Table 18.9*.

<table>
<thead>
<tr>
<th>Core configuration</th>
<th>FPGA CLBs</th>
<th>Equivalent gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Amount</td>
<td>Overhead (%)</td>
</tr>
<tr>
<td>TE-51</td>
<td>1372</td>
<td>-</td>
</tr>
<tr>
<td>TE-51_EDC</td>
<td>1599</td>
<td>16.55</td>
</tr>
</tbody>
</table>

Table 18.17: Implementation size comparison of processors with and without error detection
Techniques intended to reduce the impact of program-flow errors on embedded systems

The implementation overhead, in terms of FPGA slices and gate count, is shown in Table 18.17. The gate count difference between TE-51 and TE-51_EDC of about 3200 gates, which is trice the sum of CBG, HWFT and SP. This difference is accounted for by the processor core modifications, additional SFRs and immediate vectoring units. In any case, this increase is only 7.5% of TE-51 (in terms of gate count).

The difference between the increase in FPGA CLBs and gate count is due to each CLB being only configurable to a certain extent. Hence, the equivalent gate count gives a better indication of implementation size when it comes to fabrication.

18.6.4 Conclusion
The combined benchmark has shown that no injected errors were UNDET-classified (i.e. CBG detected all the errors that it was supposed to). Of the errors that were detectable, almost all of them were detected by CBG within four clock-cycles. HWFT’s longer latency meant that most errors that could be detectable by it, were first detected by CBG instead.

Overall, the combined effectiveness of CBG, HWFT and SP, coupled with its relatively small footprint, makes these techniques suitable for many applications.
18.7 Conclusion

The effectiveness of CBG and HWFT in detecting program-flow errors, based on ‘real-world’ conditions, was evaluated in this chapter. Overall, CBG was shown to detect all errors that were within its coverage region; hence, it is 100% effective. On the other hand, HWFT only managed to detect a portion of detectable errors. HWFT’s error detection latency was also much longer than CBG’s. This greatly influenced the outcome when both techniques were implemented: most errors within HWFT’s coverage were detected by CBG instead.

With small programs, the use of CBG alone should be sufficient to ensure detection of program-flow errors. Both techniques should be implemented with larger programs, since HWFT will provide some error detection within the programmed code memory region. This approach is better than no error detection.

SP was shown to detect almost all injected errors meant to present erroneous register writes. Hence, with effective program-flow error detection techniques, SP is a suitable mechanism that will prevent register writes before errors are detected.

CBG, HWFT and SP’s implementation (together) increases TE-51’s gate count by approximately 7.5%. This is a small price to pay considering the level of error detection which these techniques provide. TE-51_EDC’s gate count overhead is also significantly less than other comparable techniques such as Signature Monitoring or the WatchDog Processor.
Techniques intended to reduce the impact of program-flow errors on embedded systems

DISCUSSION AND CONCLUSIONS
19 Discussion and conclusion

This chapter discusses the key findings of this thesis, and draws some conclusions based on the work that has been carried out.

19.1 Program counter corruption
The outcome of the studies presented in Part One and Part Two showed that the consequences of PC corruption are more serious than initially thought. For processor families with multi-read instruction sets, von Neumann processor were deduced to have higher chances of EMIT and LMIT errors occurring than Harvard architecture devices. On the other hand, Harvard architecture processor families without multi-read instruction sets are not affected by EMIT and LMIT errors, except when data – stored in code memory (constants) – are misinterpreted (LMIT error). This is not the case for von Neumann processor families with single-read instruction sets: LMIT errors will occur when data is misinterpreted as instructions.

A model to predict the outcome of PC corruption, based on the static and dynamic instruction-type profile of a program, was presented in Part Two. The results of the simulations suggested that this model is accurate to within 5% of the simulated results, with an average difference of approximately 2%. Hence, this model should be accurate enough for most applications.

Although this model is specific to the 8051 instruction set, it can be tailored for other Harvard architecture processors with multi-read instruction sets.

19.2 Software-based error detection and/or correction techniques
The experiments carried out in Chapter 4 and Chapter 7 is further proof that NOP Fills and Function Tokens detect program-flow errors. Both techniques complement each other in
Techniques intended to reduce the impact of program-flow errors on embedded systems

their error detection coverage: the former only detects program-flow errors branching execution to unprogrammed code memory locations, and vice versa.

NF’s error detection rate – within its coverage – is practically 100%; with a very low latency. On the other hand, FT’s error detection rate is less than 100%, and its latency can be significantly longer than NF’s. The experimental results also show that FT’s error detection rate does increase with the number of checks implemented. However, a direct relationship between both parameters does not exist.

In terms of processing resources, NF does not require any; but its coverage (hence detection rate) is solely dependent on the size of the unprogrammed code memory locations. FT however, can require a significant amount of code memory and processing cycles; the former is related to the number of checks implemented, the latter to the number of checks executed for a given time frame.

NF’s implementation is trivial and usually does not require alterations to the program code (it only requires the addition of the PFEH). Although it is not difficult to employ FT, its implementation can be very tedious and error prone, especially for large programs with complex function calls. Hence, FT’s implementation can substantially increase design cost due to the time overhead.

NF and FT can be implemented on practically all processor families without modifications since they work on the software level. The only issue that may arise is when FT is implemented on processors with multi-level priority interrupt systems (i.e. more than one ISR could be executing): care must be taken to prevent the token from accidental corruption.

If NF and FT are implemented together, NF’s coverage will decrease since FT requires code memory. This decrease, may significantly affect NF’s error detection rate, as shown in Chapter 7. The overall error detection rate should increase due to the larger error detection coverage, but it does not necessarily mean an increase in system safety, as shown in the model described in Part One. In addition, FT checks will only work as intended if the PC ‘lands’ before a check takes place, not in the middle of a check.

In conclusion, NF is an error detection technique that should be employed in all cases, especially when ratio between the size of the programmed and physical code memory is
Techniques intended to reduce the impact of program-flow errors on embedded systems

small. On the other hand, the issues discussed in this section must be considered before FT is implemented. This thesis has also shown that implementing both techniques may actually decrease a system’s safety level.

19.3 Chip-based error detection and/or correction techniques

ST, ST2 and BCT – described in Part Three – have been shown to detect and correct between 83% and 98% of PC corruptions. These techniques have practically zero error detection and correction latencies over the entire addressable code memory (coverage). In addition, their implementation is transparent to the processor, which leads to two advantages: 1) the programs that are to be ported to a processor that implements these techniques do not require modifications, and 2) modifications to development tools are unnecessary.

These techniques have a few drawbacks. Firstly, they cannot detect program-flow errors that were not caused by PC corruption (e.g. Stack Pointer and Instruction Register corruption), unless implemented on other program-flow related registers (ST2 and BCT can also detect address bus errors). Secondly, their error detection rates will vary significantly if they are implemented on other processor families. Thirdly, these techniques are most effective when they work on clock-multiplied cycles: this may not be achievable on other processors.

<table>
<thead>
<tr>
<th>Technique</th>
<th>First Gates</th>
<th>% of TE-51</th>
<th>Subsequent Gates</th>
<th>% of TE-51</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>567</td>
<td>1.95</td>
<td>463</td>
<td>1.59</td>
</tr>
<tr>
<td>ST2</td>
<td>817</td>
<td>2.81</td>
<td>704</td>
<td>2.42</td>
</tr>
<tr>
<td>BCT</td>
<td>1331</td>
<td>4.58</td>
<td>624</td>
<td>2.15</td>
</tr>
</tbody>
</table>

Table 19.1: ST, ST2, BCT and SP’s initial and subsequent implementation overhead

ST, ST2 and BCT’s gate-count overhead for the first and subsequent implementation (i.e. on many registers) shows that these techniques should have very little impact on the processor’s production cost. Hence, they should be feasibly implemented on the 8051 processors, especially those that are safety and reliability related.
Techniques intended to reduce the impact of program-flow errors on embedded systems

19.4 Peripheral-based error detection and/or correction techniques

CBG and HWFT – two techniques that are similar to NF and FT respectively – have been shown to detect program-flow errors. These techniques however, have many advantages over NF and FT. For a start, CBG’s coverage is, in most cases, significantly higher than NF’s since it is unaffected by memory aliasing; a major advantage over NF.

CBG’s error detection rate should be the same as NF’s. On the other hand, HWFT’s error detection rate may be better than FT’s (for similar implementation). For both techniques, their latencies are generally shorter than their software-based counterparts.

CBG requires a few code memory locations and clock cycles for configuration, unlike NF. This is usually a one-off process however. On the other hand, HWFT’s implementation will require significantly less processor resources than an equivalent implementation with FT: HWFT checks are only single instructions. In addition, HWFT can be implemented at the instruction level, which may significantly increase error detection rate; and HWFT’s implementation may be automated, which can drastically reduce its implementation cost. HWFT checks also have a far higher chance of working successfully than FT checks due to their smaller size (code-wise).

CBG and HWFT may be easily implemented on other processor families that support SFR-like memory spaces, and may do so without modifications to the processor core. The only issues that must be addressed are: 1) the processor and peripheral SFR access times, and 2) the exception handling mechanism. For processors without SFR-like memory architectures, CBG and HWFT’s control registers have to be mapped to other memory spaces (e.g. RAM).

<table>
<thead>
<tr>
<th>Technique</th>
<th>Gates</th>
<th>% of TE-51</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBG</td>
<td>29730</td>
<td>2.27</td>
</tr>
<tr>
<td>HWFT</td>
<td>29662</td>
<td>2.04</td>
</tr>
<tr>
<td>CBG &amp; HWFT</td>
<td>30041</td>
<td>3.34</td>
</tr>
</tbody>
</table>

Table 19.2: Overall implementation overhead for CBG, HWFT and SP

CBG and HWFT’s implementation only marginally increases the processor’s gate count (with respect to TE-51’s 29069 gates), as shown in Table 19.2, which means they should be very
Techniques intended to reduce the impact of program-flow errors on embedded systems

feasible for most embedded systems. In addition, the gate count overhead only increases marginally when both techniques are implemented.

When CBG and HWFT are implemented together, CBG’s coverage only decreases slightly. CBG and HWFT’s coverage will overlap; but in most cases, CBG will be the technique that detects such errors. This is desirable due to CBG’s 100% error detection rate.

19.5 SecurePorts

SP - a mechanism that allows errors to be detected before ports (and registers) are written - has been shown detect 99% of erroneous port writes that do not conform to the strict writing procedure. SP can be used in many ways such as: 1) allowing program-flow error checks to be inserted between both writes, 2) allowing two (or more) different algorithms to calculate the first and second register write values, and 3) creating more secure RAM locations whereby two writes are necessary to store a value.

SP’s error detection latency can be as short as three clock cycles, or as long as three clock cycles more than the ‘write window’ (value of SPCNT). SP will double the access time when writing to SP-enabled registers, and each write also doubles the code memory overhead. Both overheads will not be significant in most implementations however, unless SP-enabled registers are written a very high rate. A more important issue is its double writes must be converted to single writes, and vice versa, when a program is transferred between processors that implement SP and those that do not. This issue can be resolved by using a tool to automatically insert (or remove) the second write.

SP’s gate count overhead is the highest of all the techniques presented in this thesis (6.54% and 3.66% of TE-51 for first and subsequent implementations respectively). Even so, its overhead should not significantly increase a processor’s production cost, as long as it is not implemented on many registers.
## 19.6 Summary of program-flow error detection and/or correction techniques

A summary of the program-flow error detection and/or correction techniques proposed and discussed in this thesis is shown in Table 19.3. This summary is based on some of the factors in selecting the suitable error detection technique that were presented in Chapter 1.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Overall cost</th>
<th>Error detection rate</th>
<th>Error detection latency</th>
<th>Processing resources</th>
<th>Program alterations</th>
<th>Development tool alterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>Insignificant. Die overhead should be trivial.</td>
<td>Average &gt; 80% for PC corruption (PC overwrites considered).</td>
<td>Immediate. PC errors are corrected before processor reads.</td>
<td>Not required.</td>
<td>Not required.</td>
<td>No modifications necessary.</td>
</tr>
<tr>
<td>ST2</td>
<td>Insignificant. Die overhead should be trivial.</td>
<td>Average &gt; 90% for PC corruption (PC overwrites considered).</td>
<td>Immediate. PC errors are corrected before processor reads.</td>
<td>Not required.</td>
<td>Not required.</td>
<td>No modifications necessary.</td>
</tr>
<tr>
<td>BCT</td>
<td>Trivial. Die overhead should be trivial.</td>
<td>Average &gt; 95% for PC corruption (PC overwrites considered).</td>
<td>Immediate. PC errors are corrected before processor reads.</td>
<td>Not required.</td>
<td>Not required.</td>
<td>No modifications necessary.</td>
</tr>
<tr>
<td>CBG</td>
<td>Insignificant. Trivial die and implementation overhead.</td>
<td>Within coverage: ~ 100%. Outside coverage: 0%</td>
<td>With immediate vectoring: less than 4 clock-cycles.</td>
<td>Negligible. Cycles and code memory for 5 SFR writes.</td>
<td>Negligible. A few instructions to configure CBG.</td>
<td>No modifications necessary.</td>
</tr>
<tr>
<td>HWFT</td>
<td>May be non-trivial. Die overhead is negligible but implementation time may not.</td>
<td>Within coverage: variable, less than 100%. Outside coverage: 0%</td>
<td>Min.: 4 clock-cycles. Max.: period between longest HWFT checks</td>
<td>Usually negligible. Dependent on number of HWFT checks implemented.</td>
<td>Tedious. Functions calling structure must be mapped out in detail.</td>
<td>No modifications necessary. Tool could be developed to automatically insert checks.</td>
</tr>
</tbody>
</table>

Table 19.3: Summary of proposed program-flow error detection and/or correction techniques
19.7 Conclusion

This thesis has successfully shown that the effects of PC corruption can be estimated from a program’s static and dynamic instruction-type profile, which can be used as a guide to determine a program’s safety level. In addition, this thesis has also evaluated, and successfully implemented, a collection of error detection and/or correction techniques that are feasible for most embedded systems.

Some form of error detection technique should be implemented to increase a system’s tolerance to errors. The impact of their implementation however, must be carefully studied: it is not always the case that they would increase a system’s safety level.

No firm conclusion can be made as to which techniques are most suitable to detect program-flow errors; many issues such as program type (process oriented, stimulus driven), market (consumer mass-produced, military) and application (safety-critical, reliability-critical) must be factored when selecting the suitable techniques. Some general recommendations however, can be made:

- CBG should be implemented for all designs. Designs unaffected by memory aliasing can implement NF instead.
- HWFT should be implemented for most designs, especially those with large program sizes. They should not be implemented as an afterthought, and can be employed at the instruction level to improve error detection rates.
- SP should be implemented on some important registers, especially the I/O ports, for systems that interact with external devices (e.g. relays, pneumatic pumps). SP must be used in conjunction with other program-flow error detection techniques such as HWFT.
- ST2 or BCT could be used in conjunction with HWFT to improve the error detection rate in programmed code memory locations for 8051 family processors. They could also be implemented on multiple registers to improve error detection and correction.
Future Work

Although a lot of work has been carried out in this project to investigate the outcome of PC corruption and develop techniques to detect program-flow errors, there are still many avenues that can be pursued. In this penultimate chapter, certain aspects of this research that could be improved upon, and some avenues for further studies, are discussed.

20.1 Increasing the robustness of peripheral-based techniques

Currently, CBG, HWFT and SP’s enable bits remain at the state that they were written. If two errors - each toggling the state of one enable bit - occur, these modules may be unwittingly enabled/disabled. To reduce the chances of such situations occurring, the enable bits should revert to the current state (i.e. 1,0 or 0,1) within a few clock-cycles after they were written.

Some registers such as CBGL and EDC are crucial for the peripheral-based techniques to work correctly. These registers could be duplicated and a simple comparison module used to determine if both register values are identical: differences would signify register corruption. If register correction is necessary, they could be triplicated and a simple majority-voting scheme used. This should vastly increase their robustness.

Alternatively, registers that only need to be written once when a program starts (e.g. CBGL), could be designed in such a way that only one write is possible after a reset.

The implications of CBG, HWFT and SP’s register corruption have not been investigated. This should be done.

20.2 Improving ST, ST2 and BCT’s error detection and correction rate.

Although ST, ST2 and BCT are able to detect a majority of PC errors, their problems are 1) only PC and address bus corruptions are detected (unless implemented on other program-flow
Techniques intended to reduce the impact of program-flow errors on embedded systems

related registers), 2) they works on the 'quan-cycle' and, 3) their effectiveness is governed by the 'quan-cycle' to instruction cycle ratio.

By only detecting PC changes of three or more for non-branching instructions on the last sub-cycle of each clock-cycle, and including a register to store the previous PC’s value (history buffer), program-flow errors could be detected during non-branching instruction sequences. In addition, since this potential improvement means a technique may no longer operate on the clock-multiplied cycle, this makes it more portable amongst different processor families, while potentially maintaining the same level of effectiveness. As with ST, ST2 and BCT’s implementation, branching instructions would also be ignored, unless other program-flow related registers are monitored (e.g. PSW). The drawback of this proposed modification is automatic PC correction is no longer possible: this can be done in software, however. In addition, erroneous program-flow branches within three address locations of the PC’s correct value are undetectable.

CBG could also be integrated with this proposed technique to detect program-flow into unprogrammed code memory locations. Therefore, increasing the techniques ability to detect program-flow errors.

The impact of implementing ST, ST2 and BCT on other program-flow related registers (i.e. as described in Chapter 12) have not been investigated. A complete study is necessary to fully quantify and qualify their effectiveness with such implementations.

20.3 Multiple tokens for HWFT
Many embedded systems use real-time operating systems (RTOS). On such systems, the program should be considered as a collection of tasks that could be executed concurrently, if the RTOS is pre-emptive. As long as each task does not call functions that are shared between other tasks, the current implementation of HWFT can be used. In this situation, the token is changed by the RTOS: only comparisons operations take place within the tasks and functions.

87 The probability of this is slim, assuming that the PC has the same probability to hold any value.
Techniques intended to reduce the impact of program-flow errors on embedded systems

If a function is shared between tasks, either token comparison is not carried out in the shared function, or the token has to be compared with all possible IDs. The latter approach is preferred, but cannot be implemented on HWFT: HWFT would raise an error as one or more IDs would not match. A way around this is to have two separate token and token checks, i.e. duplicating HWFT. As before, one token is controlled by the RTOS, which is checked by each task. The second token however, is only used by each task and treated as a local token. The RTOS would store and reload the local token upon task switching.

The token could also be made incrementable/decrementable after each token check is carried out. Provided there is not overlap in the token’s value between different functions, this will allow the detection of program branches within a function, which should be especially effective for long (code-wise) functions. To increase the number of possible function IDs on 8-bit devices, two SFR locations could be used instead and a writing scheme – similar to SP – used to ensure both locations are written correctly. A single SFR should be sufficient for processors with wider data buses (e.g. 16-bit devices can have 65536 unique IDs).

20.4 Reducing SP’s overhead

SP’s implementation could be reduced in three ways. The first is to tailor SP’s operations to meet (say) TE-51’s P1 write signal. Since TE-51 was only made available after it was designed (see Chapter 17), it was decided that a module to stretch P1’s write signal (PULSE_STRETCHER – see Figure Q.2) be used, instead of redesigning and resimulating SP. If SP’s internal operations are reassigned to coincide with a processor’s write signal, the ‘pulse stretching’ module can be discarded, saving approximately 239 gates (reported by Xilinx’s XST) for each SP-implemented register.

The second potential gate count reduction is to remove the P1B register (see Figure 17.2). The second P1 writes’ value will directly connect to the Comparator, but not stored within SP. It will also trigger the Control module, and the appropriate action will be taken (P1WRO pulsed or not) based on COMP’s result (the CMF signal). This saves approximately 8 D-flip-flops per implementation (64 gates).

SP’s gate count can be further reduced if the write window (i.e. the maximum clock-cycles between the first and second register writes) is fixed. This means the SPCNT SFR is no
longer necessary, which will save approximately 32 gates (4 D-flip-flops) for the implementation described in this thesis.

If these approaches are adopted, SP's gate count overhead can be significantly reduced (to approximately 730 gates) to a level comparable with BCT's, especially for subsequent implementations.\(^8\)

20.5 Enhancements to 8051Sim/8051Sim-NG

Although 8051Sim-NG can be over an order faster than 8051Sim, its graphical and scripting interface relied on Visual Basic (the simulator itself is written in Visual C++). Hence, communication between the front-end and the simulation engine is via an intermediate interface, which does impact 8051Sim-NG's performance. A major enhancement would be to completely code 8051Sim in Microsoft Visual C++ (or another compiled languages). This means 8051Sim's performance will significantly improve and could possibly outperform 8051Sim-NG (in scripting) due to the lack of the intermediate interface. In addition, the current development lines (8051Sim and 8051Sim-NG) would be merged, making it easier to maintain the program.

Other enhancements that would improve 8051Sim's usability and execution speed are:

- Creating a framework that allow custom peripherals to be written as DLLs, which can be loaded when required. Such a workflow will improve simulation speeds with custom peripherals, and modularise each peripheral.
- Include the ability to read programs in other formats (e.g. OMF – object module format) since the Intel Hex8 format does not include any debugging information or symbols.
- Allow programs to be displayed at higher levels of abstraction (e.g. source code). This multi-layered view shows the relationship between instructions and source code, which is especially useful when it comes to debugging programs and custom peripherals.
- Allow the size of the physically implemented code memory to be altered within the program.

\(^8\) Implementations from the second register onwards.
Techniques intended to reduce the impact of program-flow errors on embedded systems

- Allow handling of complex breakpoints. Ideally, they should be describable as complex syntax constructs (e.g. ‘break when P1 = 0x00 and PC = 0x3214’).
- Include the ability to watch variables. The changes in watched variables should also be tied to the breakpoint system.

20.6 Enhancements to the MCU Testbench

It was briefly discussed in Chapter 18 that two processors should be executed concurrently during the benchmarking process, one of which will be injected with errors. This arrangement allows the concurrent detection of differences between both processors after error injection. An extra module that can be instructed to compare and report the status of certain registers could be implemented. Hence, phenomena such as the effect of PC writes overwriting a corrupted PC can be easily identified.

An arrangement like this would require FPGAs with a higher gate count. As the Spartan-II FPGA used in this project is the largest in its family, a dual-processor testbench would have to be implemented on the Virtex family of FPGAs, or by connecting two FPGAs. Porting TE-51 and the MCU Testbench will not be hard: they are written in VHDL, and both FPGA families are similar. The Virtex family of FPGAs also contain more block RAM (up to 1 megabits), allowing for greater code memory sizes.

The MCU Testbench is also incomplete in the sense that not all RAM/SFR locations are accessible. Above this, most, if not all, registers should be made corruptible. The fault injection unit should also be tailored to read fault injection parameters from a file (probably stored in RAM) in order to corrupt all registers with any value at any time, such as that described in [Delong et. al. 1996]. The breakpoint system should also be expanded to handle register assesses.
Techniques intended to reduce the impact of program-flow errors on embedded systems

APPENDICES
Appendix A References


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems


(http://www.mse.vt.edu/faculty/hendricks/mse4206/projects97/group02/hardening.htm#soi)


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems


Techniques intended to reduce the impact of program-flow errors on embedded systems

Appendix B  Publications

M. J. Pont, H. L. R Ong, Using watchdog timers to improve the reliability of TTCS embedded systems: Seven new patterns and a case study, VikingPlop 2002, Copenhagen, Denmark, Sept. 2002.


Appendix C  Hardware- and design-based error detection techniques

Some hardware- and design-based (see Table 1.1) error detection techniques are discussed here. These techniques were briefly mentioned in Chapter 1.

C.1 Chip level EMI reduction

A major source of broadband electromagnetic energy is the processor itself [Glenewinkel 1996, Engel et. al. 1996]. Since a device that emits electromagnetic energy at a particular spectrum is equally susceptible to interference in the same spectrum [Paul 1992] (Kolodziejski & Kucinski [Kolodziejski & Kucinski 2000] experimentally showed this relationship), EMI reduction techniques should first be implemented at the chip level.

The issue of increasing a processor's immunity to EMI (and high-energy particles) is gathering support due to the ever-shrinking semiconductor dimensions and increasing clock frequencies, which exacerbates the chances of processors being corrupted [Kiencke et. al. 1996, Tosaka et. al. 1998].

Techniques that may increase a processor's immunity (at the chip level) includes the following:

- Processors should be clocked by trapezoidal waveforms. The amplitude envelop of higher order harmonics reduces faster than square waves [Steinecke & Anafang 1998].
- Input and output pins should be capacitively buffered to filter out high frequency signals [Steinecke & Anafang 1998, Ham et. al. 1998, John et. al. 2000].
- High frequency signal lines should be fabricated in microstrip formation [Ham et. al. 1998], which treats the tracks as transmission lines that have the lowest electric and magnetic field couplings of any interconnections [Armstrong 1999b].
- Diodes should be used to prevent latch-up at I/O pins (particularly for ESD) [AN435 1998].
- Phase-locked loops (PLL) should be used to multiply the processor's clock frequency internally. This removes the need for external high frequency oscillators [Siemens 1996b].
Techniques intended to reduce the impact of program-flow errors on embedded systems

• Clock signals with duty cycles that is greater or less than 50% (e.g. 30%, 60%) should be used in order to reduce the amplitude of the higher order harmonics [John et. al. 2000].

• Voltage potential wells should be integrated between I/O cell and other MCU modules, in order to reduce crosstalk [AN435 1998].

C.2 Circuit level EMI reduction

Most designers do not have control at the chip level [Banyai & Gerke 1996]. Hence, EMI reduction is usually applied at the circuit level. The following list summarises some EMI reduction techniques:

• Signal and power net return paths should be kept as short as possible to reduce the resulting loop area [AN435 1998, Vitek 1998, Williamson 1984].

• Printed circuit board tracks should be routed as curves or small-angled bends to reduce transmission reflection [Glenewinkel 1996].

• High speed nets should be kept as short as possible and not run in parallel [Banyai & Gerke 1996, Vitek 1998].

• Transmission line techniques (e.g. microstrip) could be used for high speed signals [Armstrong 1999b].

• High speed and analogue signals should not be in close proximity and separate ground planes – tied at one low impedance point – should be used [Banyai & Gerke 1996, Glenewinkel 1996, Deb 1998, Armstrong 1999a, Williamson 1984].

• Electronic components should be segregated according to their operating speeds as fast components generates noise which may affect slow (and generally quieter) components. [Banyai & Gerke 1996, Glenewinkel 1996, Vitek 1998].

• Power supplies should be filtered to remove high frequency spikes [AN435 1998, Campbell 1995, Banyai & Gerke 1996, Glenewinkel 1996, Armstrong 1999b].

• Low-pass filters should be implemented for I/O lines to reduce high-frequency noise that is easily picked up by the wiring connected to them [Campbell 1995, Deb 1998, Williamson 1984].

• PCBs could be made from high permittivity material (e.g. EmCap) to form embedded capacitance89 [Hubing et. al. 2000].

---

89 The capacitance is formed by the PCB material acting as the dielectric.
Techniques intended to reduce the impact of program-flow errors on embedded systems

- Compound circuits should be integrated as multichip modules (MCM), which have been shown in [Marot & Dall'Agnese 1998, Lankford et. al. 1997, Bosley et. al. 1995, Simsek et. al. 1999] to reduce EMI (and electromagnetic emission – EME).

C.3 System level EMI protection

System level protection involves physically shielding the electronics and filtering the wiring harnesses that connect to it. Some possible approaches include:

- Enclosing systems in metal or metallic-coated cases (shielding) can be very effective in reducing EMI [Klemmer 1996]. Although metal shielding is more effective, conductive coatings can be applied to surfaces with various contours [Jackson & Bleeks 1999, Klemmer 1996].

- The use of RF (radio frequency) chokes or ferrite beads to remove high-speed transients on wiring harnesses [Soohoo & Wu 1998, Lee et. al. 1998].

- The use of LC-style (inductor-capacitor) filters at wiring-circuit board junctions can also reduce EMI by up to 100dB [O'shea 1996]. The usage of 360 degree terminating connectors (e.g. D-shell connectors) will prevent EMI/EME from entering/exiting at the wiring harness and system connection points [Soohoo & Wu 1998].

- Screened and shielded cables prevent EMI and reduce EME in the same way as the housing for the electronic devices.

C.4 High-energy radiation protection

High-energy cosmic ray particles can be shielded from semiconductor devices in the same manner as EMI shielding [Caldwell & Rennels 2000, Rennels et. al. 1997]. These techniques will also protect processors from particles generated during coronal mass ejections (CME) [Caldwell & Rennels 2000, Rennels et. al. 1997].

High-energy particle originating from the processor's packaging material (e.g. alpha particles [Grey 2000]) and those produced by galactic cosmic rays (GCR) cannot be shielded in the same manner as EMI [Caldwell & Rennels 2000, Rennels et. al. 1997]. The only approach that can be taken is to 'radiation hardened' these devices. Possible 'radiation hardening' processes [Carson et. al. 1997] include:

- Junction Isolation – This technique involves reverse biasing the PN junctions to isolate on-chip components from one another.
Techniques intended to reduce the impact of program-flow errors on embedded systems

- **Dielectric Isolation** – This technique involves thermally growing a thick layer of silicon dioxide between adjacent devices to provide component isolation.

- **Silicon-on-Sapphire** – This is a more complex form of dielectric isolation where a silicon film is grown over a sapphire substrate, a dielectric with an inherently high tolerance to radiation.

- **Silicon-on-Insulator** – Very similar to the previous technique except that silicon dioxide is used as the substrate.

**C.5 Conclusion**

The hardware- and design-based techniques to increase the system and processor’s immunity to EMI and high-energy particles have been shown to be effective by various authors. Some of these techniques are commonly employed in commercial designs, others are used in specialised systems.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Appendix D Software- and chip-based error detection and correction techniques

Two software-based error detection and correction techniques, NOP Fill and Function Token, are discussed in Chapter 4. In this appendix, other software- and chip-based (see Table 1.1) techniques that do not specifically target program-flow errors are described.

D.1 Overview
Software-based techniques typically involve coding programs to a specific framework. Table D.1 summarises some of these techniques.

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port Voting</td>
<td>Digital Port</td>
<td>Port pin read odd times. Level with higher hits taken as correct level.</td>
</tr>
<tr>
<td>Port Refreshing</td>
<td>Digital Port</td>
<td>Port updated regularly with shadow value stored in RAM.</td>
</tr>
<tr>
<td>Range Rejection</td>
<td>Analogue Port</td>
<td>Analogue values outside permissible range rejected.</td>
</tr>
<tr>
<td>Digital Filtering</td>
<td>Analogue Port</td>
<td>DSP filtering (e.g. FIR filter, Input Averaging) performed on input value.</td>
</tr>
<tr>
<td>Checksum</td>
<td>RAM constants</td>
<td>Checksum for all RAM constants stored at unused RAM locations. Checksum is regularly calculated and compared with the stored value.</td>
</tr>
<tr>
<td>Range Rejection</td>
<td>RAM/SFR</td>
<td>RAM/SFR values regularly read to determine if values are within a permissible range.</td>
</tr>
<tr>
<td>Data Duplication</td>
<td>RAM/SFR</td>
<td>Multiple copies of critical RAM/SFR locations are stored at unused RAM locations.</td>
</tr>
<tr>
<td>Function Tokens</td>
<td>Core</td>
<td>A token stores a function’s ID. On function calls, the token would be compared with the ID (see Chapter 4).</td>
</tr>
<tr>
<td>NOP Fills</td>
<td>Core</td>
<td>Unused ROM locations filled with NOP instructions (see Chapter 4)</td>
</tr>
<tr>
<td>Watchdog Timer</td>
<td>Core</td>
<td>The Watchdog Timer value is set to a value slightly greater than the time taken to go through a block of code (see Chapter 13).</td>
</tr>
</tbody>
</table>

Table D.1: Summary of software-based error detection and recovery techniques

90 This is considered a peripheral-based technique in this thesis.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Generally speaking, software-based techniques can be classified into three groups based on their applications: inputs and outputs, memory and program flow. Some of these techniques can be applied in more than one area; some can be used in conjunction with others to improve a system’s overall error detection and correction ability.

D.2 Inputs and Outputs
As MCUs are connected to sensors and actuators through their I/O ports, it is vital that adequate precautions are taken to prevent corruption of ports, and that the correct inputs values are read. This is necessary as undetected corrupted digital/analogue input values may wrongly be processed by the processor, and corrupted output port values may accidentally trigger actuators. Therefore, these situations must be addressed.

D.2.1 Port Voting
‘Port Voting’ (also known as ‘Majority Voting’) [Niaussat 1998, AN435 1998, Campbell 1995, Campbell 1998, Caldwell & Rennels 2000, MISRA 1994, Rennels et. al. 1997] is a simple technique that can be applied to improve the accuracy of digital inputs (e.g. for keypad debouncing). This technique involves reading the port an odd number of times, and inferring that the correct level to be the one that recorded the most hits. Although simple to implement, Port Voting is effective against transient errors, as long as the period between each read is longer than the period of a transient.

```c
char Port30Read(char Iter) {
    char ZERO, ONE, Count;           // declaring variables
    for(Count=1; Count<=Iter; Count++) // port reading loop
        if((P3 & 0x01) == 0x01) {       // check if P3.0 = 1
            ONE ++;
        } else {                      // P3.0 = 0
            ZERO ++;
        }
    if(ONE > ZERO) return 1;         // return value with
    else return 0;                   // more hits
}
```

Listing D.1: ‘Port Voting’ C implementation example

91 Most commercial MCUs only have digital outputs; hence, analogue outputs are not considered.
Techniques intended to reduce the impact of program-flow errors on embedded systems

The effectiveness of Port Voting is proportional to the number of port reads. As the number of reads increases however, so does the period of the input signal. Hence, a trade-off must be made between the maximum input rate, and the number of reads.

Ports employing Port Voting require at least three memory locations, two for each possible input state and one for the counter. Port Voting can be implemented as a function, as shown in Listing D.1.

D.2.2 Port Refreshing

With 'Port Refreshing' [Niaussat 1998, Campbell 1995, Campbell 1998, Coulson 1999, Burnham & Cowling 1984, MISRA 1994], a port’s value is periodically refreshed with a value stored in a corresponding memory location. Instead of writing to the ports, the program writes to this memory location. A timer or a scheduled task is then used to periodically update the ports with the value held in their corresponding memory location, hence refreshing the port’s content. Therefore, if the port latches are corrupted, the ‘refreshing’ operation will restore the port latches to their intended value.

This technique is only effective when the outputs are connected to slow reaction devices, or those that employ techniques similar to Port Voting. For normal electronic devices, the potentially long delay may cause errors before correction occurs.

D.2.3 Range Rejection

‘Range Rejection’ [Banyai & Gerke 1996] is a technique to reject input values that are not within a specified range. This technique is suited for digitised analogue values: it acts as a magnitude filter for continuous signals. Hence, induced voltage transients above a preset threshold can be prevented from reaching other processing stages.

For this technique to work with different noise levels, the thresholds should be adaptable. One method is to add an offset equal to the average noise level, to the preset threshold. This offset can be determined at runtime by signalling the input device to send a known signal level, which is digitised and read by the MCU. The difference between the average digitised and known value would be the offset.

---

92 In both cases, the output ports in question must be refreshed before other device reads.
Range Rejection, although crude in comparison to digital signal processing techniques (e.g. FIR filters – see next sub-section), may be beneficial in situations where the noise level is fairly constant [MISRA 1995].

**D.2.4 Digital Filtering**

'Digital Filtering' involves digital signal processing techniques such as finite-impulse filters and infinite-impulse response filters (IIR). Techniques like these are well known and have been implemented into numerous electronic appliances, especially those associated with communication.

As DSP techniques are based on proven mathematical concepts, their characteristics can be accurately modelled, and tailored for specific needs. Most of them however, require substantial processing capabilities\(^9\); which are not always available on standard processor architectures (apart from digital signal processors).

One simple technique – 'Input Averaging' (also known as 'Majority Voting') – was presented in [Niaussat 1998, AN435 1998, Coulson 1999, MISRA 1994]. This technique is, in many ways, similar to Port Voting. An analogue signal is repeatedly digitised and the average value represents the required signal\(^9\).

Input Averaging is less processing intensive than most DSP techniques as it only requires additions, shifts and one division for each sample. This technique may still be relatively time consuming – especially for relatively slow processors like the 8051 – and should only be applied to slow-changing signals. When implemented after Range Rejection, the averaged signal may be more accurate as readings outside the permissible range will not influence the results.

An example implementation of Range Rejection and Input Voting, based on Infineon’s C515 8051-compatible MCU, is shown in Listing D.2. It is assumed that only the upper eight converted bits are needed.

---

\(^9\) It is a common practice to synthesize DSP-specific routines as hardware block, instead of coding them in software.

\(^9\) 'Input Averaging' is equivalent to an FIR filter with all coefficients being \(1/(\text{number of samples})\).
Techniques intended to reduce the impact of program-flow errors on embedded systems

```c
char Port63Read(char Iter, char Min, char Max)
{
    char Count = 0;
    long SUM;

    do {
        ADDTL = 0x00;
        //reading loop
        //start conversion
        while(BSY); //wait till ADC finish
        if((ADDTH >= Min) && (ADDTH <= Max)) //Range Rejection
        { //Range Rejection
            Count ++;
            SUM += ADDTH;
            //inc. read counter
            //add new value
        }
    } while(Count < Iter); //loop # iterations
    return (char) (SUM / Iter); //average value
}
```

Listing D.2: 'Range Rejection' and 'Input Averaging' C implementation example

D.3 RAM and SFR

It is not feasible to perform software-based tests on the entire RAM/SFR memory due to the processing overhead involved. By limiting the error detection and recovery scope to critical RAM/SFR locations however, a good compromise is usually achievable.

RAM variables and constants implementing error detection and recovery techniques should be contiguously stored to reduce processing and memory overhead since it is generally easier to handle these locations as a collection.

D.3.1 Checksum

'Checksum' [AN435 1998, Banyai & Gerke 1996, Coulson 1999] involves mathematically generating a value derived from a set of 'critical locations'. Whenever any 'critical data' are written, the checksum will be generated and stored. The checksum is then calculated and compared with the stored valued before any 'critical data' are read. This ensures that the unadulterated 'critical data' is read.

This technique requires an abundance of processing power, and delays the memory accesses to 'critical data' locations. The number of mathematical operations and data moves necessary is proportional to the size of the 'critical data' set.

D.3.2 Range Rejection

Identical in methodology to that for analogue inputs, 'Range Rejection' can identify variables and constants that are not within an 'acceptable range' [Burnham & Cowling 1984].
Techniques intended to reduce the impact of program-flow errors on embedded systems

value of a variable is checked when it is being written or read, and a check is carried out when a constant is read.

The ‘acceptable range’ thresholds can be stored in code memory, if it remains constant. These thresholds can be shared amongst variables and constants with similar values to reduce the overhead involved in storing the threshold values.

Range Rejection can be less computationally intensive than Checksum, but is only effective when the ‘acceptable range’ is small as compared with the ‘entire range’. For example, Range Rejection would be more effective on a 16-bit ‘critical data’ with a range of 2300-2500, than one ranging from 0-50000.

D.3.3 Data Duplication
‘Data Duplication’ [AN435 1998, Burnham & Cowling 1984, Shuette & Shen 1987] (also known as ‘massive redundancy’) works by using unused RAM locations to hold multiple copies of ‘critical data’. When writing to a ‘critical data’, all locations associated with it would be updated. Upon a read, all the associated locations have to be read and a voting scheme (or just a simple comparison) determines the correct value. Data duplication is effective in detecting transient errors, but can be very costly in terms of processing resources [Shuette & Shen 1987].

Data Duplication is an alternative method of implementing Checksum where, instead of dealing with a group of ‘critical data’, each ‘critical data’ is dealt with individually. The advantages would be less processing overhead resulting in faster reads, and the possibility of correcting corrupted ‘critical data’. On the other hand, this method requires more RAM.

D.4 Program Flow
Three techniques are available to detect program-flow errors. Two of them, NF and FT, are purely software-based techniques: they are described in Chapter 4. The Watchdog Timer is a peripheral-based technique that was described in Chapter 13.
D.5 Conclusion

Software-based techniques can, and do, provide basic I/O filtering, detection of RAM/SFR errors and program-flow errors. Although some techniques are processing intensive, many of them can be implemented on cost conscious systems, especially when the errors occur infrequently. Some techniques can also be combined with others to increase the overall detection/filtering rate.
Appendix E  User guide for 8051Sim

The 8051Sim (see Chapter 6) users’ guide starts with the installation of 8051Sim, followed by the graphical interface, program execution, breakpoint control, scripting and program termination.

E.1  Installation

8051Sim is completely contained in a single executable (8051Sim.exe). Since it is programmed in Visual Basic, it does however require the run-time library to be installed. For scripting, the Windows Scripting Control is also necessary (it is installed by default on Windows 2000/XP). The run-time library and scripting control can be downloaded from these locations respectively:


8051Sim generates two files – ‘desktop.ini’ and ‘settings.ini’ – to store window positions and settings respectively, in the same directory as the executable. If these files are not present, they would be created and the default settings used.

The files for 8051Sim and 8051Sim-NG are located on the CD-ROM (refer to Appendix R). An up-to-date version of each program can be downloaded from the department’s website at http://www.engg.le.ac.uk/8051Sim.

E.2  User interface

E.2.1  ROM features

Figure E.1: ‘Fill ROM’ dialogue box
Techniques intended to reduce the impact of program-flow errors on embedded systems

The ROM can be filled with a specific value by selecting the EPROM->Fill ROM option and entering the start address, end address and fill value (all in hexadecimal format) in the ‘Fill ROM’ dialogue box, as shown in Figure E.1.

The ROM can be erased by selecting the EPROM->Erase ROM option. If the ‘Empty EPROM = 0x00’ check box is checked in the Option->Settings menu (see Figure E.5), the ROM would be erased with 0x00, else it would be erased with 0xFF (this is the state of a physical ROM when it is empty).

E.2.2 Windows

A total of 8 windows opened from the Window menu, 7 from the Peripheral menu and one from the Debug menu shows the various status of the system. The following describes the displayed parameters:

- Window->CPU Core – Key registers and number of cycles executed since ‘Reset’.
- Window->SFR – SFR memory space.
- Window->Internal RAM – RAM (direct and indirect) memory space.
- Window->Internal ROM – ROM memory space.
- Window->Toolbox – Frequently used simulation commands.
- Window->Program Code – Program listing window.
- Window->Virtual RS232 Terminal – Virtual RS232 terminal.
- Window->Script Output – Window for scripts to print information.
- Peripheral->Ports->Port <3:0> – Port 0 to Port 3.
- Peripheral->Timers/Counters – Timer 0 and Timer 1.
- Peripheral->Serial Comm. – USART.
- Peripheral->Interrupts – Generic interrupt system.
- Debug->Script Input – Writing and executing short script commands.

Some windows (described in the previous sub-section) are interactive. They are described in their respective sub-sections.

E.2.3 Toolbox window

The ‘Toolbox’ window (see Figure 6.1) consists of seven commonly used simulation commands: ‘Full run’, ‘Update run’, ‘Animate’, ‘Single step’, ‘Micro step’, ‘Stop’ and ‘Reset’. Pressing this buttons is the same as selecting the relevant commands from the Debug menu.
Techniques intended to reduce the impact of program-flow errors on embedded systems

### E.2.4 Program Code window

The ‘Program Code’ window, shown in Figure E.2, lists the program (in assembly) that has been loaded into code memory. A red bar hi-lights the instruction that is being executed. A ‘!’ appears at the second column to signify that it is still being read by the processor. Instructions that have been executed are marked ‘+’, which also shows the execution coverage. Upon a ‘Reset’, program execution would start at location 0x0000 and all ‘+’ and ‘!’ are cleared. If a new program is reloaded, the breakpoints are also cleared.

![Program Code window](image)

Double clicking on any line would toggle its breakpoint status. When an execution breakpoint is set, a blue bar appears at that particular location, and a ‘B’ is indicated at the first column.

When a program is being executed (except in ‘Full run’ mode), this window would position the currently executed code in the middle of window if the ‘Enable Code Tracking’ check box in the Option->Settings menu (see Figure E.5) is checked.

### E.2.5 Virtual RS232 Terminal window

The Virtual RS232 Terminal, shown in Figure E.3, works at the same baud rate as that set by the 8051’s USART. Only modes 1, 2 and 3 are supported. In modes 2 and 3, the ‘9th bit’ section would appear (bottom left of window). The 9th bit is transmitted by checking the ‘Transmit’ check box while the ‘Receive’ check box shows the status of that bit from the 8051’s USART.

When the ‘local echo’ check box is checked, whatever that is typed in the terminal area will be shown. Three status ‘LEDs’ lights up when bits are transmitted, received, or if any form
Techniques intended to reduce the impact of program-flow errors on embedded systems

of communication error has occurred. The ‘Clear Terminal’ button clears the terminal. Note: only one character can be transmitted at a time to the 8051’s USART.

![Virtual RS232 Terminal window](image)

Figure E.3: ‘Virtual RS232 Terminal’ window

E.2.6 Port 0, Port 1, Port 2 and Port 3 windows
All port windows work the same. The port latch values are shown in the upper row of check boxes, the port pin values are shown on the lower row (see Figure 6.1). Only when a corresponding port latch bit is set can the port pin values be toggled.

E.2.7 Script Input window

![Script Input window](image)

Figure E.4: ‘Script Input’ window

The ‘Script Input’ window – shown in Figure E.4 – is for users to enter short script commands directly. The script is written in the main window and the ‘OK’ button is pressed to execute the script. The ‘Clear Screen’ button clears the script input. If single line commands are entered, the subroutine declarations (i.e. ‘Sub Main’ and ‘End Sub’) can be omitted by checking the ‘Single Subroutine’ check box.
E.2.8 Settings window

Seven settings can be configured from the ‘Settings’ window, shown in Figure E.5, which is called by selecting the Option->Settings menu. A description of these settings are given below:

- **XTAL frequency** – Sets the oscillator frequency.
- **Animate Rate** – Sets the delay between each step-execution in ‘Animate’ mode.
- **Anti-alias PC** – Enable PC antialiasing (PC values above the ROM size would be wrapped round) when checked. If it is not checked, PC values above the ROM size would cause simulator errors.
- **Enable Serial Module** – 8051’s USART will be enabled when checked (processing intensive).
- **Enable Code Tracking** – The ‘Program Code’ window will always show the currently executed instruction when checked.
- **Micro Step Update** – Windows will be updated at the sub-cycle level under ‘Update Run’ mode when checked. If it is not checked, update occurs on every sub-cycle.
- **Empty EPROM = 0x00** – The code memory will be erased with 0x00 when checked. If it is not checked, the code memory will be erased with 0xFF.

These settings can be saved in the ‘settings.ini’ file when the ‘Save Default’ button is pressed. If only the ‘OK’ button is pressed, these settings would only be valid until 8051Sim is terminated. The ‘Cancel’ button discards any changes.

E.2.9 The Status Bar

The Status Bar (bottom of 8051Sim – see Figure 6.1) shows the following information:

- **Cycles** – Number of clock-cycles executed since last 'Reset'.
- **Time** – Execution time in 'real-world' since last 'Reset'.
- **BP Active** – Number of active breakpoints.
Techniques intended to reduce the impact of program-flow errors on embedded systems

• **Int. Level** – Interrupt level: ‘0’ for no interrupt, ‘1’ for handling low priority interrupt, and ‘2’ for handling high priority interrupt.

• **Script running** – ‘LED’ lit when script is running.

• **State to Execute** – Following sub-cycle to execute.

• **Sim. Status** – Simulator run status.

E.3 Program simulation

E.3.1 Program loading

8051Sim only accepts programs in the Intel Hex8 format. Programs are selected from the ‘Open’ dialogue box that is displayed when the **File->Load Hex File** menu is selected. The simulator is automatically reset before the code memory is loaded. Note: binary (*.bin) files are currently not supported.

E.3.2 Program execution

Programs can be executed in five different modes:

- **Full Run** – Simulation runs without graphical interface updates (fastest).
- **Update Run** – Simulation runs with graphical interface updates.
- **Animate** – The 8051 is step-simulated at a pre-specified rate.
- **Single Step** – The 8051 is executed for 1 clock-cycle (to the boundary of the next cycle).
- **Micro Step** – The 8051 is executed for 1 sub-cycle.

For all execution modes, the USART can be disabled to increase simulation speeds by unchecking the ‘Enable Serial Module’ check box in the **Option->Settings** menu (see **Figure E.5**). In ‘Update Run’ mode, graphical interface updates can be set to occur on every sub-cycle, if the ‘Micro Step Update’ check box in the same dialogue box is checked. The animate rate is also set in the ‘Animate Rate’ entry.

All execution modes can be triggered by pressing the respective buttons in the ‘Toolbox’ window, from the **Debug** menu, or by pressing their respective ‘hot-keys’ (e.g. ‘Update Run’ is F5). Two other commands, ‘Stop’ and ‘Reset’ are also accessible in the same manner.
Techniques intended to reduce the impact of program-flow errors on embedded systems

E.4 Breakpoints

Up to 10 breakpoints can be configured at any given time. The addition, edition, removal, enabling and disabling of breakpoints are discussed in the following sub-sections. Note: the ‘Reset’ command does not reset the breakpoints.

E.4.1 Adding Breakpoints

Breakpoints are set in the ‘Breakpoint’ window – shown in Figure E.6 – by selecting the Debug->Breakpoint menu. The ‘Add Breakpoint’ button is used to add an existing breakpoint. When this button is pressed, the ‘Add/Edit Breakpoint’ dialogue box appears (see Figure E.7).

The first selection list shows the type of breakpoint that can be selected. They are described as follows:

- **CYCLES** – Break when execution cycle is reached.
- **PC** – Break when the code location is executed.\(^\text{95}\)
- **SFR** – Break based on an SFR’s value.
- **RAM** – Break based on an RAM’s value.

The second selection list would be active if the breakpoint type is selected as SFR. All the generic SFR locations can be selected (including the port pins). If other SFR locations that are not listed are desired, the last option enables the ‘Address’ field where the desired location can be entered. This field is also enabled when the breakpoint is of type RAM.

\(^{95}\) This is the same as double-clicking on the ‘Program Code’ window.
The forth field (from left) is the value that a breakpoint must meet – according to the conditions selected in the third (from left) list – for one to occur. This value can be entered in hexadecimal or decimal, depending on the radix selected. Pressing the ‘OK’ button sets the breakpoint; ‘Cancel’ exits this dialogue box without enabling it.

![AddEdit Breakpoint](image)

**Figure E.7: ‘Add/Edit Breakpoint’ window**

### E.4.2 Editing Breakpoints
Breakpoints can be edited by double-clicking it in the ‘Breakpoint’ window, or by selecting it and pressing the ‘Edit Breakpoint’ button (see Figure E.6). This launches the ‘Add/Edit Breakpoint’ dialogue box (see Figure E.7) with the fields set accordingly. The procedure for editing a breakpoint is the same as adding one.

### E.4.3 Removing Breakpoints
Breakpoints can be removed by selecting it in the ‘Breakpoint’ window and pressing the ‘Remove Breakpoint’ button (see Figure E.6). If all breakpoints need to be removed, the ‘Remove ALL BP.’ button can be used instead.

### E.4.4 Enabling/Disabling Breakpoints
Breakpoints can be enabled or disabled by checking or unchecking the check box beside each breakpoint in the ‘Breakpoint’ window (see Figure E.6). Breakpoints are always enabled when they are added.

### E.5 Scripting
The scripts can either be entered from the ‘Script Input’ window, or more commonly, from a text file loaded via the Debug -> Run Script menu. If errors are detected in the script, they would be reported in a message box. When the script is running, the simulator’s icon appears in the system tray. Note: 8051Sim/8051Sim-NG would freeze when scripts are being executed.
Techniques intended to reduce the impact of program-flow errors on embedded systems

All scripts must conform to the following format to work:

- There must be at least one subroutine called ‘Main’. This is the subroutine that execution starts.
- All variables and constants must be prefixed by ‘sv.’.
- All functions and subroutines must be prefixed by ‘sf.’.

Please refer to Appendix F for the description of all script variables, constants and functions.

E.6 Program Termination

8051Sim can be terminated by either selecting File->Exit menu, or by pressing the ‘close window’ button (top right corner). The user will then have a chance to save the window positions on exit.
Appendix F  Description of 8051Sim’s scripting functions and variables

8051Sim’s scripting functions are directly controlled by the MSC, which is part of the Scripting Interface sub-module (see Chapter 6). Since 8051Sim is monolithic, it is relatively easy to 'expose' the variables, constants and functions with MSC. This is not the case for 8051Sim-NG. 8051Sim-NG – written as a DLL – only exposes the essential variables, constants and functions necessary (known here as the DLL functions and variables).

For 8051Sim and 8051Sim-NG to ‘expose’ the same variables, constants and functions to ensure script compatibility, the DLL functions and variables are encapsulated by the Scripting Interface sub-module (written in Visual Basic – see Figure 6.5). Additional functions – derived from the DLL functions and variables – were also included in this sub-module. 8051Sim-NG’s Scripting Interface sub-module ‘exposes’ the variables, constants and functions in exactly the same format as 8051Sim. Hence, scripts will perform identically when executed under 8051Sim or 8051Sim-NG96.

The interactions between the DLL functions and variables and those ‘exposed’ are shown in Table F.1. Some DLL functions (Note 2) are only used for communication between the DLL and wrapper program, hence not ‘exposed’. Scripting functions are either ‘wrapped’ up versions of corresponding DLL functions, derived from more than one DLL function (Note 1), or solely written without using DLL functions (Note 3).

<table>
<thead>
<tr>
<th>DLL functions</th>
<th>Derived Scripting Functions</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetVal_DLL</td>
<td>GetRAMByte</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetRAMWord</td>
<td></td>
</tr>
<tr>
<td>Memory access</td>
<td>GetROMByte</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetROMWord</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSFR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetVal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>InsByte_A</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>InsType</td>
<td></td>
</tr>
</tbody>
</table>

96 Some functions on 8051Sim-NG do not perform operations. They are included for compatibility.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Module</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PutVal_DLL</strong></td>
<td></td>
<td>PutRAMByte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PutRAMWord</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PutROMByte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PutROMWord</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PutSFR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PutVal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ResetBit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CheckBit</td>
</tr>
<tr>
<td><strong>GetRegVal_DLL</strong></td>
<td></td>
<td>GetRegVal</td>
</tr>
<tr>
<td><strong>PutRegVal_DLL</strong></td>
<td></td>
<td>PutRegVal</td>
</tr>
<tr>
<td><strong>EraseMemory_DLL</strong></td>
<td></td>
<td>EraseMemory</td>
</tr>
<tr>
<td><strong>FillMemory_DLL</strong></td>
<td></td>
<td>FillMemory</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AddBreakpoint_DLL</strong></td>
<td></td>
<td>AddBreakpoint</td>
</tr>
<tr>
<td><strong>DisableAllBreakpoint_DLL</strong></td>
<td></td>
<td>DisableAllBreakpoint</td>
</tr>
<tr>
<td><strong>DisableBreakpoint_DLL</strong></td>
<td></td>
<td>DisableBreakpoint</td>
</tr>
<tr>
<td><strong>EnableAllBreakpoint_DLL</strong></td>
<td></td>
<td>EnableAllBreakpoint</td>
</tr>
<tr>
<td><strong>EnableBreakpoint_DLL</strong></td>
<td></td>
<td>EnableBreakpoint</td>
</tr>
<tr>
<td><strong>RemoveAllBreakpoint_DLL</strong></td>
<td></td>
<td>RemoveAllBreakpoint</td>
</tr>
<tr>
<td><strong>RemoveBreakpoint_DLL</strong></td>
<td></td>
<td>RemoveBreakpoint</td>
</tr>
<tr>
<td><strong>MicroStepSimulation_DLL</strong></td>
<td></td>
<td>MicroStepSimulation</td>
</tr>
<tr>
<td><strong>ResetSimulation_DLL</strong></td>
<td></td>
<td>ResetSimulation</td>
</tr>
<tr>
<td><strong>RunSimulation_DLL</strong></td>
<td></td>
<td>RunSimulation</td>
</tr>
<tr>
<td><strong>StepSimulation_DLL</strong></td>
<td></td>
<td>StepSimulation</td>
</tr>
<tr>
<td><strong>StopSimulation_DLL</strong></td>
<td></td>
<td>StopSimulation</td>
</tr>
<tr>
<td><strong>InsLBoundary_DLL</strong></td>
<td></td>
<td>InsLBoundary</td>
</tr>
<tr>
<td><strong>InsHBoundary_DLL</strong></td>
<td></td>
<td>InsHBoundary</td>
</tr>
<tr>
<td><strong>InsByteCycle_DLL</strong></td>
<td></td>
<td>InsByte_I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InsByte_A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InsSize_I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InsSize_A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CheckBitEqPort</td>
</tr>
</tbody>
</table>

F-2
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Function</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckSFREqPort</td>
<td>3</td>
</tr>
<tr>
<td>CloseFile</td>
<td>3</td>
</tr>
<tr>
<td>Complement</td>
<td>3</td>
</tr>
<tr>
<td>CreateFile</td>
<td>3</td>
</tr>
<tr>
<td>Echo</td>
<td>3</td>
</tr>
<tr>
<td>EchoAll</td>
<td>3</td>
</tr>
<tr>
<td>GetUserlnput</td>
<td>3</td>
</tr>
<tr>
<td>Hex2Dec</td>
<td>3</td>
</tr>
<tr>
<td>LoadHexFile</td>
<td>3</td>
</tr>
<tr>
<td>OpenFile</td>
<td>3</td>
</tr>
<tr>
<td>ReadFile</td>
<td>3</td>
</tr>
<tr>
<td>WriteFile</td>
<td>3</td>
</tr>
</tbody>
</table>

Table F.1: Interaction between DLL functions and 8051Sim-NG's script functions

F.1 8051 Special Function Registers

All 8051-generic SFRs can be read and be written. Reading will return the SFR value while writing alters it. These SFRs are in the Scripting Variable class and must be prefixed with 'sv.' (e.g. loading the accumulator with 0x45 is written as sv.\texttt{ACC} \texttt{= 0x45})\textsuperscript{97}.

<table>
<thead>
<tr>
<th>SFR</th>
<th>Description</th>
<th>SFR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Accumulator</td>
<td>P3_pin</td>
<td>Port 3 pin*</td>
</tr>
<tr>
<td>B</td>
<td>B register</td>
<td>PCON</td>
<td>Power Control</td>
</tr>
<tr>
<td>DPH</td>
<td>Data pointer (low byte)</td>
<td>PSW</td>
<td>Program Status Word</td>
</tr>
<tr>
<td>DPL</td>
<td>Data pointer (high byte)</td>
<td>SBUF</td>
<td>Serial Buffer**</td>
</tr>
<tr>
<td>IE</td>
<td>Interrupt Enable</td>
<td>SCON</td>
<td>Serial Control</td>
</tr>
<tr>
<td>IP</td>
<td>Interrupt Priority</td>
<td>SP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>P0</td>
<td>Port 0 latch</td>
<td>TCON</td>
<td>Timer Control</td>
</tr>
<tr>
<td>P0_pin</td>
<td>Port 0 pin*</td>
<td>TH0</td>
<td>T0 high register</td>
</tr>
<tr>
<td>P1</td>
<td>Port 1 latch</td>
<td>TH1</td>
<td>T1 high register</td>
</tr>
<tr>
<td>P1_pin</td>
<td>Port 1 pin*</td>
<td>TL0</td>
<td>T0 low register</td>
</tr>
<tr>
<td>P2</td>
<td>Port 2 latch</td>
<td>TL1</td>
<td>T1 low register</td>
</tr>
<tr>
<td>P2_pin</td>
<td>Port 2 pin*</td>
<td>TMOD</td>
<td>Timer Mode</td>
</tr>
<tr>
<td>P3</td>
<td>Port 3 latch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table F.2: SFRs accessible by scripts (type 'byte')

\textsuperscript{97} The scripting functions, \texttt{GetSFRVal()} and \texttt{PutSFRVal()} could also access the SFRs. Non 8051-generic SFRs (e.g. those declared for custom peripherals) can only be accessed with these functions.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Note: * These registers are not part of the 8051 memory map but are the registers for the corresponding port pin.

** Reading from SBUF reads the received buffer. Writing to it accesses the transmit buffer.

Individual bits in bit-addressable SFRs could be accessed directly (e.g. \texttt{sv.IE0 = 0}), as shown in Table F.3. This excludes the port latches and port pins. The enable and priority bits of the user-definable interrupt are accessible, however.

<table>
<thead>
<tr>
<th>SFR</th>
<th>SFR Bits</th>
<th>SFR</th>
<th>SFR Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>EX0, ET0, EX1, ET1, ES, E_EXTD0*, E_EXTD1*, EA</td>
<td>TCON</td>
<td>IT0, IE0, IT1, IE1, TR0, TF0, TR1, TF1</td>
</tr>
<tr>
<td>IP</td>
<td>PX0, PT0, PX1, PT1, PS, P_EXTD0*, P_EXTD1*</td>
<td>SCON</td>
<td>SM0, SM1, SM2, REN, TB8, RB8, TI, RI</td>
</tr>
<tr>
<td>PSW</td>
<td>CA, AC, RS1 RS0, OV, PA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table F.3: SFR bits accessible by scripts (type 'boolean')

Note: * These bits are the UDI enable and priority bits.
Techniques intended to reduce the impact of program-flow errors on embedded systems

F.2 Simulator variables and constants

Simulator variables and constants are in the same class as the 8051-generic SFRs; hence, they must also be prefix with ‘sv’, as shown in Table F.4.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>R/W*</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_INS</td>
<td>Integer</td>
<td>R</td>
<td>Mnemonic of current instruction.</td>
<td>0 to 255</td>
</tr>
<tr>
<td>CYCLES</td>
<td>Long</td>
<td>R</td>
<td>Cycles executed since Reset.</td>
<td>&gt;= 0</td>
</tr>
<tr>
<td>CYCLE_LENGTH</td>
<td>Integer</td>
<td>R</td>
<td>Cycles left to execute for instruction.</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>CYCLE_STATE</td>
<td>Integer</td>
<td>R</td>
<td>Sub-cycle state (enumerated: S1P1 = 0).</td>
<td>0 to 11</td>
</tr>
<tr>
<td>INS_EXECUTED</td>
<td>Long</td>
<td>R</td>
<td>Instructions executed since Reset.</td>
<td>&gt;= 0</td>
</tr>
<tr>
<td>MULTI-READ</td>
<td>Boolean</td>
<td>R</td>
<td>True when not all operands read.</td>
<td>False, True**</td>
</tr>
<tr>
<td>PC</td>
<td>Long</td>
<td>R/W</td>
<td>Program Counter.</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>PC_ANTIALIAS_ENABLE</td>
<td>Boolean</td>
<td>R/W</td>
<td>True to enable PC antialias mode.</td>
<td>False, True**</td>
</tr>
<tr>
<td>PC_PRE</td>
<td>Long</td>
<td>R/W</td>
<td>Program Counter last value.</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>RAM_CHANGE</td>
<td>Boolean</td>
<td>R</td>
<td>True when RAM written last sub-cycle.</td>
<td>False, True**</td>
</tr>
<tr>
<td>RAM_CHG_VAL</td>
<td>Integer</td>
<td>R</td>
<td>RAM address last written to.</td>
<td>0 to 255</td>
</tr>
<tr>
<td>SCRIPT_STATUS</td>
<td>Integer</td>
<td>R</td>
<td>Script status (see Table F.5).</td>
<td>0, 1</td>
</tr>
<tr>
<td>SFR_CHANGE</td>
<td>Boolean</td>
<td>R</td>
<td>True when SFR written last sub-cycle.</td>
<td>False, True**</td>
</tr>
<tr>
<td>SFR_CHG_VAL</td>
<td>Integer</td>
<td>R</td>
<td>SFR address last written to.</td>
<td>0x00 to 0xFF</td>
</tr>
<tr>
<td>SIM_ERROR</td>
<td>Integer</td>
<td>R</td>
<td>Simulation error status (see Table F.5).</td>
<td>0 to 5</td>
</tr>
<tr>
<td>SIM_ERROR_PROMPT</td>
<td>Boolean</td>
<td>R/W</td>
<td>True to enable simulation to prompt when error detected(^{96})</td>
<td>False, True**</td>
</tr>
<tr>
<td>SIM_STATUS</td>
<td>Integer</td>
<td>R</td>
<td>Simulation status (see Table F.5).</td>
<td>0 to 4</td>
</tr>
<tr>
<td>XTAL</td>
<td>Long</td>
<td>R/W</td>
<td>Oscillation frequency in Hz</td>
<td>&gt;0</td>
</tr>
</tbody>
</table>

Table F.4: Simulator variables and constants

Note: * ‘R’ for read-only variables (constants), ‘R/W’ for read/write variables.

** Variables/constants of type ‘Boolean’ are not interpreted the same across different programming languages.

\(^{96}\) No effect on 8051Sim-NG.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Table F.5 lists the enumerated constants that can be used in some variables (shown in Table F.4) and some functions. These enumerated constants must be prefixed with 'sv.'.

<table>
<thead>
<tr>
<th>Enumerated Constants</th>
<th>Description</th>
<th>Constants linked to variables (see Table F.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>cROM</td>
<td>0</td>
<td>Memory space enumerations</td>
</tr>
<tr>
<td>cRAM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>cSFR</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ROM_SIZE</td>
<td>*</td>
<td>Memory size constants</td>
</tr>
<tr>
<td>XRAM_SIZE</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>cSTOP</td>
<td>0</td>
<td>Simulation run status</td>
</tr>
<tr>
<td>cUPDATERUN</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>cFULLRUN</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>cANIM</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>cSTEP</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>cSCRIPTSTOP</td>
<td>0</td>
<td>Simulation script status</td>
</tr>
<tr>
<td>cSCRIPTRUN</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>eNOERROR</td>
<td>0</td>
<td>Simulator error status</td>
</tr>
<tr>
<td>eRETI</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>eMOVX</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>eMOVC</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ePCOVERFLOW</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Table F.5: Enumerated constants (type 'integer')

Note:  
* These constants are of type ‘Long’ and varies with the ROM/XRAM size (8191 and 2047 respectively for the default implementation).
** These memory space constants can be used by many script functions.

These enumerated constants make scripting functions easier to write and comprehend. For example, checking a bit value in RAM location 0x2A bit 5 can be written as:

```
SF.CheckBit 0, &h2A, 5
```

or

```
SF.CheckBit sv.cRAM, &h2A, 5
```
F.3 Scripting Functions

Scripting functions, unless explicitly stated, return a −1 when execution fails. Successfully executed functions return a non-negative value. Scripting function must be prefixed by 'SF.'.

**AddBreakpoint**

*(BPtype As Integer, BPaddress As Byte,
BPcomp As Integer, BPvalue As Long) As Integer*

**Description:** Add a new breakpoint

**Parameters:**
- **BPtype**: 0 for cycles, 1 for PC, 2 for SFR, 3 for RAM
- **BPaddress**: Breakpoint address. Only used when **BPtype** is 'SFR (others selected)' or RAM.
- **BPcomp**: Breakpoint comparison operator: 0 for 'equal', 1 for 'equal or greater than', 2 for 'equal or smaller than', 3 for 'not equal'.
- **BPvalue**: Comparison value

**Return:**
- 1 – on success.

**AnimateSimulation()**

**Description:** Continuously single-step the simulation at the predefined rate.

**Parameters:** None.

**Return:** None.

**CheckBit**

*(MemSpace As Byte, address As Byte,
bit As Byte) As Integer*

**Description:** Return the value of a bit.

**Parameters:**
- **MemSpace**: The memory space to access (cRAM or cSFR).
- **address**: Location of byte to test (0 to 255).
- **bit**: Bit number to test (0 to 7).

**Return:** Bit state (0 or 1).

**CheckBitAdd**

*(bitadd As Byte) As Integer*

**Description:** Determine the bit value of bit-addressable location.

**Parameters:** **bitadd**: SFR bit address.

**Return:** Bit state (0 or 1).

**CheckBitEqPort**

*(address As Byte) As Integer*

**Description:** Determine if a bit address is a port bit.

**Parameters:** **address**: Bit location of bit to test.

**Return:** 0 - bit is not part of port, 1 - bit is part of port.

**CheckEXTD_IF**

*(INT_num As Integer) As Integer*

**Description:** Return the state of a user-defined interrupt flag.

**Parameters:** **INT_num**: User-defined interrupt module (0 or 1).

**Return:** State of the interrupt flag (0 or 1).
Techniques intended to reduce the impact of program-flow errors on embedded systems

CheckSFREqPort (address As Byte) As Integer
Description: Determine if an SFR address is a port.
Parameters: address SFR location of byte to test.
Return: 0 - SFR is not a port, 1 - SFR is a port.

CloseFile (file_num As Integer) As Integer
Description: Close a file.
Parameters: file_num File number to close.
Return: 1 - on success.

Complement (value As Byte) As Integer
Description: 2's complements a byte.
Parameters: value Value to complement.
Return: Complemented value (-1 is a legitimate return value).

CreateFile (filename As String) As Integer
Description: Create a text file.
Parameters: filename Name of file to create.
Return: positive value - file number.

DisableAllBreakpoint() As Integer
Description: Disable all breakpoints.
Parameters: None.
Return: 1 - on success.

DisableBreakpoint (BP_num As Integer) As Integer
Description: Disable a particular breakpoint.
Parameters: BP_num Breakpoint number to disable.
Return: 1 - on success.

DisableEXTD (INT_num As Integer) As Integer
Description: Disable a user-defined interrupt module.
Parameters: INT_num User-defined interrupt module (0 or 1).
Return: 1 - on success.

Echo (str As String)
Description: Display text on the Script Output terminal.
Parameters: str String to display.
Return: None.

EchoAll (file_num As Integer, str As String) As Integer
Description: Write a line to a text file and the console
Parameters: file_num File number to write.
str String to write.
Return: 1 - on success.
Techniques intended to reduce the impact of program-flow errors on embedded systems

EnableAllBreakpoint() As Integer
Description: Enable all breakpoints.
Parameters: None.
Return: 1 - on success.

EnableBreakpoint (BP_num As Integer) As Integer
Description: Enable a particular breakpoint.
Parameters: BP_num Breakpoint number to enable.
Return: 1 - on success.

EnableEXTD (INT_num As Integer) As Integer
Description: Enable a user-defined interrupt module.
Parameters: INT_num User-defined interrupt module (0 or 1).
Return: 1 - on success.

EraseMemory (space as Integer) As Integer
Description: Erase (set to 0x00) an entire memory space.
Parameters: space Memory space to erase (cROM, cRAM, cXRAM, or cSFR).
Return: 1 - on success.

FillMemory (space As Integer, start_add As Long, stop_add As Long, fill As Byte) As Integer
Description: Fill a memory range with value.
Parameters: space Memory space to fill (cROM, cRAM, cXRAM, or cSFR).
   start_add Start address of fill.
   stop_add End address of fill.
   fill Fill value (0 to 255).
Return: 1 - on success.

GetRAMByte (address As Long) As Integer
Description: Returns the value stored at a RAM location.
Parameters: address RAM location of read.
Return: Value stored at RAM(address) location.

GetRAMWord (address As Long) As Long
Description: Returns the word value stored at a RAM location.
Parameters: address RAM locations of read (&h0 to &hfe).
Return: 16-bit combined value at RAM(address) and RAM(address + 1) location.
Note: address must be less or equal to 0xFE.

GetRegVal (reg_num As Byte) As Integer
Description: Returns the value stored at a register (selected bank).
Parameters: reg_num Register number (0 to 7)
Return: Value stored in that register
Techniques intended to reduce the impact of program-flow errors on embedded systems

GetROMByte (address As Long) As Integer
Description: Returns the value stored at a ROM location.
Parameters: address ROM location of read.
Return: Value stored at ROM(address) location.

GetROMWord (address As Long) As Long
Description: Returns the word value stored at a ROM location.
Parameters: address ROM locations of read (&h0 to ROM_SIZE-1).
Return: 16-bit combined value at ROM(address) and ROM(address + 1) location.
Note: address must be less or equal to ROM_SIZE-1.

GetSFRVal (address As Byte) As Integer
Description: Returns the value stored at an SFR location.
Parameters: address SFR location of read (0x80 to 0xFF).
Return: Value stored in that SFR.

GetUserInput (prompt As String, title As String) As String
Description: Display a dialogue box for user to enter values.
Parameters: prompt Text message to display for user input.
title Dialogue box title.
Return: User input as string.

Hex2Dec (hex As String, length As Integer) As Long
Description: Converts a hexadecimal string into a decimal equivalent number.
Parameters: hex hexadecimal string
length number of characters to convert.
Return: Decimal equivalent of the string.

InsByteCycle_A (address As Long) As Integer
Description: Returns the instruction length and cycle time.
Parameters: address ROM address of first byte of instruction.
Return: 11 - 1B 1C, 12 - 1B 2C, 14 - 1B 4C,

InsByteCycle_I (ins As Byte) As Integer
Description: Returns the instruction length and cycle time.
Parameters: ins Instruction hex code.
Return: 11 - 1B 1C, 12 - 1B 2C, 14 - 1B 4C,

InsLBoundary (address As Long) As Integer
Description: Returns the PC position (bytes) from the first byte of the current instruction.
Parameters: address ROM address.
Return: 0 – on the first byte of an instruction, 1 to 2 – not the first byte.
Techniques intended to reduce the impact of program-flow errors on embedded systems

**InsHBoundary** *(address As Long) As Integer*
Description: Returns the PC position (bytes) to the first byte of the next instruction.
Parameters: address ROM address.
Return: 1 to 3.

**InsType** *(address As Long) As Integer*
Description: Returns the instruction type.
Parameters: address ROM address of first byte of an instruction.
Return: 1 - Single, 2 - DN, 3 - D2, 4 - TN, 5 - T23, 6 - T3.

**InsSize_A** *(address As Long) As Integer*
Description: Returns the length of an instruction.
Parameters: address ROM address of first byte of instruction.
Return: Instruction length.

**InsSize_I** *(ins As Byte) As Integer*
Description: Returns the length of an instruction.
Parameters: ins Instruction hex code.
Return: Instruction length.

**LoadHexFile** *(filename As String) As Integer*
Description: Loads a program file (Intel HEX format).
Parameters: filename Name of hex file (including path).
Return: 1 - file loaded, -1 if file is not in HEX format, -2 if program too big, -3 other errors.

**MicroStepSimulation**()
Description: Execute one sub-cycle (e.g. from S1P2 to S2P1).
Parameters: None.
Return: None.

**OpenFile** *(filename As String, mode As Integer) As Integer*
Description: Opens a file for reading/appending.
Parameters: filename Filename of the file to open/append.
mode 0 for writing, 1 for appending.
Return: Positive value - file number.

**PutRAMByte** *(address As Long, value As Byte) As Integer*
Description: Loads the specified RAM location with a value.
Parameters: address RAM location for loading.
value Value to load.
Return: 1 - on success.

**PutRAMWord** *(address As Long, value As Long) As Integer*
Description: Loads two specified consecutive RAM locations with an integer value.
Parameters: address Starting RAM location for loading.
value Value to load.
Return: 1 - on success.
Note: address must be less or equal to 0xFE.
Techniques intended to reduce the impact of program-flow errors on embedded systems

PutRegVal  
\[(\text{reg\_num As Byte, reg\_val As Byte}) \text{ As Integer}\]

Description: Loads the specified register a value (selected bank).

Parameters: 
- \text{reg\_num}  register number (0 to 7).
- \text{reg\_val}  value to load.

Return: 1 - on success.

PutROMByte  
\[(\text{address As Long, value As Byte}) \text{ As Integer}\]

Description: Loads the specified ROM location with a value.

Parameters: 
- \text{address}  ROM location for loading.
- \text{value}  Value to load.

Return: 1 - on success.

PutROMWord  
\[(\text{address As Long, value As Long}) \text{ As Integer}\]

Description: Loads two specified consecutive ROM locations with an integer value.

Parameters: 
- \text{address}  Starting ROM location for loading.
- \text{value}  Value to load.

Return: 1 - on success.

Note: \text{address} must be less or equal to (\text{ROM\_SIZE} - 1).

PutSFRVal  
\[(\text{address As Byte, value As Byte}) \text{ As Integer}\]

Description: Loads the SFR location with a value.

Parameters: 
- \text{address}  SFR location for loading.
- \text{value}  Value to load.

Return: 1 - on success.

ReadFile  
\[(\text{file\_num As Integer}) \text{ As String}\]

Description: Reads a line from a text file.

Parameters: 
- \text{file\_num}  File number to read.

Return: read line, "" if read not successful.

RemoveAllBreakpoints() \text{ As Integer}

Description: Remove all breakpoints.

Parameters: None.

Return: 1 - on success.

RemoveBreakpoint (BP\_num As Integer) \text{ As Integer}

Description: Remove a breakpoint.

Parameters: 
- \text{BP\_num}  Breakpoint to remove.

Return: 1 - on success.

ResetBit  
\[(\text{MemSpace As Byte, address As Byte, bit As Byte}) \text{ As Integer}\]

Description: Resets a specific bit.

Parameters: 
- \text{MemSpace}  Memory space (cRAM or cSFR).
- \text{address}  Location of byte to reset.
- \text{bit}  Bit number to reset (0 to 7).

Return: 1 - on success.
Techniques intended to reduce the impact of program-flow errors on embedded systems

ResetBitAdd (bitaddr As Byte) As Integer
Description: Resets a bit located at bit-addressable SFR location.
Parameters: bitaddr bit-address (0x00 to 0xFF)
Return: 1 - on success.

ResetEXTD_IF (INT_num As Integer)
Description: Resets a user-defined interrupt flag.
Parameters: INT_num User-defined interrupt module (0 or 1).
Return: 1 - on success.

ResetSimulation()
Description: Resets the virtual 8051 MCU.
Parameters: None.
Return: None.

RunSimulation (speed As Integer) As Integer
Description: Run the virtual 8051 MCU.
Parameters: speed Simulation speed (cUPDATE_RUN or cFULL_RUN).
            No effect on 8051Sim-NG.
Return: 1 - on success.

SetBit (MemSpace As Byte, address As Byte, bit As Byte) As Integer
Description: Sets a specific bit.
Parameters: MemSpace Memory space (cRAM or cSFR).
            address Location of byte to set.
            bit Bit number to set (0 to 7).
Return: 1 - on success.

SetBitAdd (bitaddr As Byte) As Integer
Description: Sets a bit located at bit-addressable SFR locations.
Parameters: bitaddr bit-address (0x00 to 0xFF).
Return: 1 - on success.

SetEXTDCodeVector (INT_num As Integer, INT_vec As Integer)
Description: Sets a user-defined interrupt vector location.
Parameters: INT_num User-defined interrupt module (0 or 1).
            INT_vec Interrupt vector location (ROM).
Return: 1 - on success.

SetEXTDFlagLoc (INT_num As Integer, INT_flag As Integer)
Description: Sets a user-defined interrupt flag location.
Parameters: INT_num User-defined interrupt module (0 or 1).
            INT_flag Interrupt flag location (SFR).
Return: 1 - on success.
Techniques intended to reduce the impact of program-flow errors on embedded systems

SetEXTD_IF (INT_num As Integer)
Description: Sets a user-defined interrupt flag.
Parameters: INT_num User-defined interrupt module (0 or 1).
Return: 1 - on success.

StepSimulation()
Description: Single-steps the virtual 8051 MCU.
Parameters: None.
Return: None.

StopSimulation()
Description: Stops the virtual 8051 MCU.
Parameters: None.
Return: None.

UpdateStatusBar()
Description: Updates the status bar (not for 8051Sim-NG).
Parameters: None.
Return: None.

UpdateWindows()
Description: Updates all opened windows (not for 8051Sim-NG).
Parameters: None.
Return: None.

WriteFile (file_num As Integer, str As String) As Integer
Description: Write a line into a text file.
Parameters: file_num file number to write.
str string to write.
Return: 1 - on success.
Appendix G 8051Sim custom peripheral example

Based on the framework presented in Chapter 6, a simple 16-bit up/down timer – T16 – is constructed as a custom peripheral. Section G.1 describes its functionality, which is then expressed in VBScript as part of the simulation script in Section G.2. The MCU program, compiled with the Keil C51, is written to control T16 via its dedicated SFRs.

G.1 Description of T16
T16 is designed as a 16-bit up/down timer with the following requirements:

- Two 8-bit registers, T16L and T16H, are used as timing registers. The timing registers are incremented/decremented at S3P2 when enabled.
- One 8-bit register, T16CON is used to control the timer.
- The count direction is controlled by the bit T16_DIR, that counts upwards when set.
- T16 is enabled whenever bit T16_RUN is set.
- Upon T16 overflowing/underflowing, bit T16_OV is set for one clock-cycle.
- Bit T16_IF, its interrupt flag, is set whenever the timer overflows/underflows. This bit is only reset when the processor vectors to its ISR. EXTDO, the first UDI source, is used to handle T16’s interrupt.

The timing and control register addresses and reset values are shown in Figure G.1.

| Special Function Register T16_CON (0x2F) | Reset Value: 0x00 |
| Special Function Register T16L (0x91) | Reset Value: 0x00 |
| Special Function Register T16H (0x92) | Reset Value: 0x00 |

![Figure G.1: Special Function Register definitions for T16](image-url)
Techniques intended to reduce the impact of program-flow errors on embedded systems

G.2 Script with T16

Const T16L = &H91 ' SFR enumerations
Const T16H = &H92
Const T16CON = &H2F
Const T16_DIR = &H78 'bit enumerations
Const T16_RUN = &H79
Const T16_OV = &H7A
Const T16_IF = &H7F
Const S1P1 = 1 'sub-cycle enumeration
Const S3P2 = 6

Sub Main()
  Dim CY, SCY, RUN_CYCLES ' variables
  SF.LoadHexFile "T16.hex" 'load program file
  ResetT16() 'resets T16
  RUN_CYCLES = Clng(SF.GetUserlnput("Cycles to execute", "User input"))
  For CY = 0 To RUN_CYCLES 'simulate cycles
    For SCY = 1 To 12 '12 sub-cycles
      SF.MicroStepSimulation() 'step one sub-cycle
      UpdateT16() 'update peripheral
    Next
    SF.Echo "CY=" & Cstr(SV.CYCLES) & "
    Hex(SF.GetSFRVal(&h92)) & ": T16H=" & _
    Hex(SF.GetSFRVal(&h91))
  Next
End Sub

Sub ResetT16()
  SF.SetEXTDCodeVector 0, &h2B 'set int. vector
  SF.SetEXTDFlagLoc 0, &h7F 'set int. flag
  SF.EnableEXTD 0 'enable EXTDO
  SF.PutSFRVal T16L, 0 'reset T16L counter
  SF.PutSFRVal T16H, 0 'reset T16H counter
  SF.PutSFRVal &H2F, 255 'reset control bits
End Sub

Sub UpdateT16()
  If SV.CYCLE_STATE = S1P1 Then 'only at S1P1
    If SF.CheckBitAdd(T16_OV) = 1 Then 'check if T16_OV set
      SF.ResetBitAdd T16_OV 'reset T16_OV
    End If
  End If
  If SV.CYCLE_STATE = S3P2 Then 'at S3P2
    If SF.CheckBitAdd(T16_RUN) = 1 Then
      If SF.CheckBitAdd(T16_DIR) = 1 Then
        If SF.GetSFRVal(T16L) = 255 Then
          SF.PutSFRVal T16L, 0 'reset T16L counter
        ElseIf SF.GetSFRVal(T16H) = 255 Then
          SF.PutSFRVal T16H, 0 'set interrupt flag
          SF.SetEXTD_IF 0 'set T16_OV flag
        Else
          SF.PutSFRVal T16H, SF.GetSFRVal(T16H) + 1 'underflow occurred
        End If
      End If
    Else
      SF.PutSFRVal T16L, SF.GetSFRVal(T16L) + 1 'underflow occurred
    End If
  End If
End Sub
Techniques intended to reduce the impact of program-flow errors on embedded systems

Listing G.1: Script to simulate T16

Simulation begins by resetting the virtual 8051 MCU, indirectly through execution of the script function `SF.LoadHexFile()`: note the class prefix, `SF`. This function’s main process is to load the program into code memory. Once this is completed, the extended interrupt system, interrupt flag and interrupt vector location is then configured and enabled in the subroutine `ResetT16()`: in this case, this is a script-declared subroutine and no prefix is required. T16’s memory locations are also reset in this subroutine.

Two ‘For...Next’ loops are used in subroutine ‘Main()’ for cycling through the required number of clock (entered by the user) and sub-cycles. By executing `UpdateT16()` after every microstep, the custom peripheral is updated on the sub-clock-cycle basis. Changes to T16-dedicated SFRs are taken into account by the custom peripheral when the control bits and timing registers are read and updated during relevant sub-cycles.

G.3 Program code with T16

```c
#include <reg51.h>
sfr T16L = 0x91;              //T16L occupies 0x91
sfr T16H = 0x92;              //T16H occupies 0x92
bdata char T16CON _at_ 0x2F; //T16CON occupies 0x2F
sbit T16_DIR = T16CON A  0;  //direction bit
sbit T16_RUN = T16CON A  1;  //run bit
sbit T16_OV = T16CON A  2;   //over/under flow bit
sbit E_EXTD0 = OxAD;          //interrupt enable bit
sbit P_EXTD0 = 0xBD;          //interrupt priority

void main()
{
    T16_DIR = 1;                //T16 is up counter
    T16L = OxEO;                //starting value =
    T16H = 0x45;                //0x45E0
    E_EXTD0 = 1;                //enable EXTD0 int.
    P_EXTD0 = 1;                //set high priority
    T16_RUN = 1;                //start T16
}
```

Custom peripherals should not be accessed at SFR/RAM access sub-cycles to prevent bus contention.
Techniques intended to reduce the impact of program-flow errors on embedded systems

```
EA = 1; //global int. enable
for(;;);
} //loop forever

void T16ISR() interrupt 5 //T16 ISR at 0x2B
{ 
T16_RUN = 0; //stop timer
T16L = 0xE0;
T16H = 0x45;
T16_RUN = 1; //start timer
if(P0 == 0xA8) P0 = 0x55; //Toggle P0
else P0 = 0xA8;
}
```

Listing G.2: Firmware using T16

A simple program that toggles Port 0 between 0x55 and 0xAA is shown in Listing G.2. T16’s registers and control bits are defined at the start of the program, as well as those of the extended interrupt system\(^\text{100}\). The location of T16’s control registers is reserved with compiler-specific keywords (‘sfr’, ‘bdata’ and ‘sbit’) to prevent it from being used as data storage.

The main part of the code sets up T16 to count from a predetermined value, enables its interrupt, starts T16 and loops forever. \texttt{T16ISR()}, the interrupt service routine, is executed upon T16’s timing register overflowing. Port 0 is toggled between 0xAA and 0x55 upon each ISR execution as an indicator that the custom peripheral is functioning correctly.

---

\(^{100}\) The included header only defines 8051 generic SFRs. It is also good practice to define custom peripheral SFRs in headers for code maintainability.
Appendix H  TE-51

TE-51 (see Chapter 14) is a synthesizable 8051 core written in VHDL. The processor core was developed by Mr. Felix Bertram of Trenz Electronik GmbH, while the peripherals (except the USART) and error detection techniques (CBG, HWFT and FT) were implemented by the author101. The core is briefly described here, followed by the peripherals developed by the author.

H.1 Description of TE-51

TE-51 is designed at the register transfer level (RTL) with Aldec’s Active-HDL, and synthesized with Xilinx’s XST. The entire design, except registers associated with peripherals, is the same as the design illustrated in Figure 2.2.

Although the generic 8051 core had only 128 bytes of RAM (direct access), TE-51 implemented the additional 128 bytes of indirectly addressable RAM found on newer devices102.

The limitations and differences between TE-51 and the generic 8051 core are summarised as such:

- Power-down and idle modes are not supported.
- The RETI (return from interrupt) instruction is identical to the RET (return from subroutine) instruction.
- Code and external memory accesses are carried out through dedicated buses instead of the ports. This is possible since TE-51 is designed for SoC environments.
- The MUL and DIV instructions (multiply and divide respectively) require two execution cycles instead of four.

H.1.1 States and phases

TE-51’s internal sequencer is 24-stages long (C1S1P1 to C2S6P2), equivalent to 2 clock-cycles on the generic 8051. For single cycle instructions, only the first 12 stages are used.

101 The basic port structure was develop by Mr. Bertram.
102 The generic 8051 support this additional RAM but it was not implemented.
Techniques intended to reduce the impact of program-flow errors on embedded systems

(C1S1P1 to C1S6P2). As previously mentioned, MUL and DIV have been shortened to two clock-cycles.

H.1.2 Instruction execution

TE-51’s instruction execution is identical to the generic 8051 MCU at the cycle level (except MUL and DIV), and accurate to the sub-cycle for instruction/operand reads. The operations within TE-51 are divided into data operations (i.e. manipulation to registers and moving of values) and address operations (i.e. manipulation of the PC). Data and address operations operate at specific sub-cycles during each cycle, as shown in Table H.1. The data and address operations are denoted by ‘D’ and ‘A’ respectively.

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 P1</td>
<td>S1 P1</td>
</tr>
<tr>
<td>D D D D A A D D D D A A</td>
<td></td>
</tr>
<tr>
<td>A A</td>
<td></td>
</tr>
</tbody>
</table>

Table H.1: Data and address operations for TE-51

Operands are read at C1S1P2, C1S4P2 and C2S1P2, which is before the data operations cycles. When operands would be read at C1S4P2, no address operations are carried out in this segment (C1S4P1 and C1S4P2) to prevent erroneous operand reads. In any way, address operations that would alter the PC can only take place during the second address operation segment (C2S3P2 and C2S4P1) since part, or the whole, branching address is in the third operand, which is only read at C2S1P2.

For many single-cycle instructions (i.e. 111- and 121-type), data operations only occur during the second set (between C1S5P1 and C1S6P2). A few more complex instructions also use the first set of data operations (between C1S2P1 and C1S3P2). As there are no branching single-cycle instructions, no address operations will take place. Overall, single-cycle instructions require between one and five data operations.

Double cycle instructions (i.e. 112- and 122-type) require between one and eight data operations, depending on their complexity. Conditional or unconditional branching instructions require either one or two address operations: either the first or second segment is used, and not both. The MUL and DIV (originally 114-type) instructions only require one
data operation (at C1S6P2) since these operations are carried out with dedicated hardware multipliers/divisors.

### H.2 TE-51 peripherals

The timers and interrupt system were developed by the author. A few minor changes to the processor’s core were necessary to form the SFR-based communication channel between the processor and the peripheral, and for the processor to handle interrupts. The SFR-based communication method is described in *Appendix L*. The schematic of the peripheral system is shown in *Figure H.1*.

![Schematic of TE-51's peripherals](image)

*Figure H.1: Schematic of TE-51's peripherals*

All connections between the core and peripherals are shown in *Figure H.1*. *Table H.2* describes each connection. The direction of the signals is with respect to the peripheral.
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rst</td>
<td>input</td>
<td>Reset signal</td>
</tr>
<tr>
<td>clk</td>
<td>input</td>
<td>Clock</td>
</tr>
<tr>
<td>csp</td>
<td>input</td>
<td>Cycle, state and phase (enumerated - e.g. C1S4P2)</td>
</tr>
<tr>
<td>mcode</td>
<td>input</td>
<td>Instruction information (enumerated)</td>
</tr>
<tr>
<td>Opl(7:0)</td>
<td>input</td>
<td>Value of instruction’s first byte</td>
</tr>
<tr>
<td>regi(7:0)</td>
<td>input</td>
<td>SFR address</td>
</tr>
<tr>
<td>sfri(7:0)</td>
<td>input</td>
<td>SFR value (to be written)</td>
</tr>
<tr>
<td>sfrwri</td>
<td>input</td>
<td>SFR write signal</td>
</tr>
<tr>
<td>sfro(7:0)</td>
<td>output</td>
<td>SFR value (to be read)</td>
</tr>
<tr>
<td>INT_LCALL</td>
<td>output</td>
<td>Signal to inform processor to perform int. vectoring</td>
</tr>
<tr>
<td>INT_VEC(23:0)</td>
<td>output</td>
<td>Instruction that is injected into the processor’s core</td>
</tr>
<tr>
<td>P3XR(7:0)</td>
<td>input</td>
<td>Port 3 pin values</td>
</tr>
</tbody>
</table>

Table H.2: Description of signals and buses to and from the peripheral module

H.2.1 SFR module
The SFR module (te51SFR – see Figure H.1) houses all the peripheral SFRs in a single location. If the peripheral SFRs had been directly tied to the core, some multiplexors within the core have to be redesigned whenever a peripheral is added or removed. Hence, the technique employed keeps processor core modifications to the minimum (i.e. only an additional input in the register address and value multiplexors in the core is necessary). The addition and removal of peripheral SFRs are carried out within te51SFR.

The processor core accesses the peripheral SFRs like other core SFRs (e.g. ACC, PSW), during the data operations, as shown in Table H.1. At S2P2, the SFR module updates Timer 0 and Timer 1’s timing registers and their interrupt flag (TF0 or TF1), if necessary. This update depends on the status of the TO_UPDATEi and TI_UPDATEi signal. If an update takes place, the new timing values are available by S3P1.

At S5P2, the external interrupt flags (IE0 and IE1) are updated based on the value presented by the TE51P3_state module. Both the timing register update and external interrupt reads occur at the same sub-cycle as the generic 8051’s [Atmel 1997, Siemens 1996a].

At S1P1, te51SFR would update the interrupt flags (in the TCON SFR) based on the values presented by te51int (via INTFLAGS_I(3:0)). This is necessary as the interrupt module may have changed the status of some flags after they have been handled. The update is
Techniques intended to reduce the impact of program-flow errors on embedded systems

chosen at S1P1 since no other processor accesses happens on this sub-cycle, and the correct values would be available in time for the current execution cycle.

The SFR module also sets a signal, \( \text{IP}_{\text{IEO}} \), for one clock-cycle whenever the IP or IE register is written. This signal is used by the interrupt system (te51INT) to prevent it from executing ISR vector immediately after IP or IE is written (there will at least be a one-cycle delay).

**H.2.2 Timer 0 and Timer 1**

The te51TMR and te51TMR\_INC modules handle both timers. te51TMR is connected to the external control inputs on Port 3\(^{103}\). With this inputs and the timer configuration bits (stored in TMOD), te51TMR determines – via \( \text{TO\_UPDATE}_{\text{i}} \) and \( \text{TI\_UPDATE}_{\text{i}} \) – if te51SFR should or should not update TL0, TH0, TL1 and TH1 with the values presented by te51TMR\_INC, at S2P2.

The second module, te51TMR\_INC, presents the incremented values of the timing registers, taking each timer’s mode into account to ensure the correct registers are incremented. If an overflow is detected, the respective interrupt request bits are set, which would be updated by te51SFR at S2P2 (if \( \text{TO\_UPDATE}_{\text{i}}/\text{TI\_UPDATE}_{\text{i}} \) is set).

**H.2.3 External interrupts**

te51P3\_STATE determines if an interrupt condition is met by considering the state of the external interrupt pins, and their control bits (edge- or level-triggered). To detect edge-triggered interrupts (high to low transition), the state of the external pins that are read at S5P1, are aged (i.e. their contents copied to a ‘history’ register) at S4P2 of the following cycle. Hence, when the latest state is read at S5P1, an edge-triggered interrupt is detected if the ‘aged’ value is high while the new value is low. The respective interrupt request signal is then set.

If an interrupt has been configured as level-triggered, the respective interrupt request signal will be set if the corresponding interrupt pin is low at S5P1. The interrupt request signal is read by te51SFR, which updates the interrupt request bits at S5P2.

---

\(^{103}\) Port 3’s pins are multifunctional, as shown in Table 2.2.
Techniques intended to reduce the impact of program-flow errors on embedded systems

H.2.4 Interrupt system

The interrupt system consists of three parts: 1) the interrupt controller (te51INT), 2) the interrupt polling mechanism (te51INT_POLL), and 3) the instruction boundary detector (te51NOT_END_INS).

te51NOT_END_INS decodes the instruction information (mcode – a VHDL record) from the processor to determine the instruction boundaries. This module then goes high between S1P2 of the first cycle of an instruction, and S6P1 of the last cycle. The signal is used by te51INT to prevent interrupt vectoring from happening at non-instruction boundaries.

te51INT_POLL reads the interrupt flags at S5P2 of each cycle. At S5P1 of the following cycle, the read values are aged (in the same manner as described for the external interrupts) before a new read occurs at the next sub-cycle. If an interrupt flag is set for two consecutive clock-cycles, the respective interrupt bit would be set, which is read by te51INT\textsuperscript{104}.

The interrupt controller is a complex module that determines if an interrupt should be handled based on the current interrupt level, the priority of an interrupt, and if an interrupt source is enabled. If an interrupt is handled, te51INT will start the execution of the LCALL instruction to vector program-flow to the correct ISR if, and only if, the correct conditions are met (discussed in Chapter 2).

ISR vectoring is done by injecting the equivalent LCALL instruction – via INT_VEC(23:0) – directly into the processor core’s instruction unit. The INT_LCALL signal informs the core that this instruction should be executed, and not that pointed by the PC. Once the vectoring cycle completes, the respective interrupt request bits may be reset (IE0 and IE1 depends on whether an interrupt is edge- or level-triggered) by informing te51SFR through the FLAGS_I(3:0) signal. The current interrupt level is also increased to reflect its new state.

The interrupt level can only be decreased when the RETI instruction is encountered. This is detected by the interrupt system through the Op1(7:0) signal, which is the first byte of an instruction: basically its mnemonic. If the value of this signal is 0x32 (the mnemonic of the RETI instruction) at C2S1P1, the interrupt level would be decreased.

\textsuperscript{104} There are seven interrupt sources that could be polled. This is for future expansion purposes.
Techniques intended to reduce the impact of program-flow errors on embedded systems

H.3 Peripheral source codes

library IEEE;
use IEEE.STD_LOGIC_1164.all;

package te51pak2 is
  type te51INT_STATE is (
    te51INT_STATE_NONE,
    te51INT_STATE_LO,
    te51INT_STATE_HI,
    te51INT_STATE_LOHI);
end package;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;

entity te51SFR is
  port(
    clk: in STD_LOGIC; -- clock
    rst: in STD_LOGIC; -- reset
    ram_reg: in STD_LOGIC_VECTOR(7 downto 0); -- ram address
    sfr: out STD_LOGIC_VECTOR(7 downto 0); -- dbus output
    sfrw: in STD_LOGIC; -- write enable
    csp: in te51stateT; -- cycle state information
    TCONo: out STD_LOGIC_VECTOR(7 downto 0); -- to make SFR's visible
    TMDo: out STD_LOGIC_VECTOR(7 downto 0);
    TL0o: out STD_LOGIC_VECTOR(7 downto 0);
    TLOi: out STD_LOGIC_VECTOR(7 downto 0);
    IP: out STD_LOGIC_VECTOR(7 downto 0);
    IBO: out STD_LOGIC_VECTOR(7 downto 0);
    TLOi._A: STD_LOGIC_VECTOR(7 downto 0); -- new values for SFRs
    THOi._A: STD_LOGIC_VECTOR(7 downto 0);
    TLli._A: STD_LOGIC_VECTOR(7 downto 0);
    THli._A: STD_LOGIC_VECTOR(7 downto 0);
    T0_UPDATEi: in STD_LOGIC; -- flags to update T0/1
    T1_UPDATEi: in STD_LOGIC;
    INT_FLAGS_Ii: in STD_LOGIC_VECTOR(3 downto 0); -- flags from INT module
    INT_FLAGS_Pi: in STD_LOGIC_VECTOR(3 downto 0); -- flags from peripherals
    IP_IEo: out STD_LOGIC -- IP/IE just written
  );
end entity te51SFR;

architecture bhv of te51SFR is
  signal TCON: STD_LOGIC_VECTOR(7 downto 0); -- declaring SFRs
  signal TMDo: STD_LOGIC_VECTOR(7 downto 0);
  signal TL0: STD_LOGIC_VECTOR(7 downto 0);
  signal TL1: STD_LOGIC_VECTOR(7 downto 0);
  signal TH0: STD_LOGIC_VECTOR(7 downto 0);
  signal TH1: STD_LOGIC_VECTOR(7 downto 0);
  signal IP: STD_LOGIC_VECTOR(7 downto 0);
  signal IE: STD_LOGIC_VECTOR(7 downto 0);
  signal TLOi: STD_LOGIC_VECTOR(7 downto 0);
  signal TLli: STD_LOGIC_VECTOR(7 downto 0);
  signal THOi: STD_LOGIC_VECTOR(7 downto 0);
  signal THli: STD_LOGIC_VECTOR(7 downto 0);
end architecture bhv;
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
signal TCON_A: STD_LOGIC_VECTOR(7 downto 0);  -- this connects to TCON
signal P32_edge: STD_LOGIC;  -- edge/level input
signal IP_IE: STD_LOGIC;  -- IP/IE written
constant TCON_adx: STD_LOGIC_VECTOR(7 downto 0):= x"88";  -- SFR addresses
constant TMOD_adx: STD_LOGIC_VECTOR(7 downto 0):= x"89";
constant TL0_adx: STD_LOGIC_VECTOR(7 downto 0):= x"8A";
constant TL1_adx: STD_LOGIC_VECTOR(7 downto 0):= x"8B";
constant TH0_adx: STD_LOGIC_VECTOR(7 downto 0):= x"8C";
constant TH1_adx: STD_LOGIC_VECTOR(7 downto 0):= x"8D";
constant IP_adx: STD_LOGIC_VECTOR(7 downto 0):= x"8E";
constant IE_adx: STD_LOGIC_VECTOR(7 downto 0):= x"8F";

begin
process(clk, rst)
begin
if rst = '1' then
TCON <= x"0000";  -- SFRs
TL0 <= x"0000";  TH1 <= x"0000";
end if;
elsif rising_edge(clk) then
if sfrwr = '1' then
-- on rising clock edge
if sfri = '1' then
-- if SFR to be written
end case;
end if;
case csp is
when C1S1P1 =>
IP_IE <= '0';  -- reset IP_IE flag
TCON <= TCON_C;
when C1S2P2 | C2S2P2 =>
TL0 <= TL0i_A;  TH1 <= TH1i_A;
end case;
end if;
if ((INT_FLAGS_Pi(2) = '1') and (TO_UPDATEi = '1')) then
TCON(5) <= '1';  -- TCON update: peripherals
end if;
if TMOD(1 downto 0) = "11" then
-- check mode of T0
if ((INT_FLAGS_Pi(3) = '1') and (TCON(6) = '1')) then
TCON(7) <= '1';  -- other mode TF1 from TMR1
end if;
else
-- other mode TFL from TMR1
if ((INT_FLAGS_Pi(3) = '1') and (TL_UPDATEi = '1')) then
TCON(7) <= '1';  -- T0_update: peripherals
end if;
end if;
when C1S5P2 | C2S5P2 =>
TCON <= TCON_C;
when others =>
nul;  -- do nothing
end case;
end if;
end process;
```

-- TLx/THx write from TMR
TL0i_A <= TL0i when TO_UPDATEi = '1' else TL0;  -- updated condition
Techniques intended to reduce the impact of program-flow errors on embedded systems

TH0i_A <= TH0i when T0_UPDATEi = '1' else TH0;
TL1i_A <= TL1i when T1_UPDATEi = '1' else TL1;
THli_A <= THli when Tl_UPDATEi = '1' else THl;

-- ISO/1, T0/P write from peripherals and interrupt system (reset)
TCON_A(0) <= TCON(0); -- modified by peripherals
TCON_A(1) <= (not INT_FLAGS_Ii(0)) and TCON(1); -- new flag
TCON_A(2) <= TCON(2); -- modified by peripherals
TCON_A(3) <= (not INT_FLAGS_Ii(1)) and TCON(3); -- new flag
TCON_A(4) <= TCON(4); -- modified by peripherals
TCON_A(5) <= (not INT_FLAGS_Ii(2)) and TCON(5); -- new flag
TCON_A(6) <= TCON(6); -- modified by peripherals
TCON_A(7) <= (not INT_FLAGS_Ii(3)) and TCON(7); -- new flag

P32_edge <= '1' when INT_FLAGS_Pi(0) = '1' else TCON(1); -- edge triggered
P33_edge <= '1' when INT_FLAGS_Pi(1) = '1' else TCON(3);
P32_level <= INT_FLAGS_Pi(0); -- level triggered
P33_level <= INT_FLAGS_Pi(1);

TCON_B(0) <= TCON(0);
TCON_B(1) <= P32_edge when TCON(0) = '1' else P32_level;
TCON_B(2) <= TCON(2);
TCON_B(3) <= P33_edge when TCON(2) = '1' else P33_level;
TCON_B(7 downto 4) <= TCON(7 downto 4); -- same as previous

with csp select
TCON_A when C1S6P2 | C2S6P2, -- INT connected to TCON
TCON_A when C1S1P1 | C2S1P1, -- INT connected to TCON
TCON_B when others; -- per. connected to TCON

with ram_reg select
sfro <= -- output the selected SFR
TCON when TCON_adx,
TMOD when TMOD_adx,
TL0 when TL0_adx,
TL1 when TL1_adx,
TH0 when TH0_adx,
TH1 when TH1_adx,
IP when IP_adx,
IE when IE_adx,
(others => '-' when others);

TCONo <= TCON;
TM0do <= TMOD;
TL0o <= TL0;
TH0o <= TH0;
TL1o <= TL1;
TH1o <= TH1;
IPo <= IP;
IEo <= IE;
IP_IEo <= IP_IE;
end architecture;

Listing H.2: Source of te51SFR

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.TE51pak.all;

tenity te51TMR is
port(
clk: in STD_LOGIC; -- clock
csp: in te51stateT; -- cycle state information
TMODi: in STD_LOGIC_VECTOR(7 downto 0); -- timer control values
TCONI: in STD_LOGIC_VECTOR(7 downto 0); -- TMRO update
T0_UPDATEx: out STD_LOGIC;
T1_UPDATEy: out STD_LOGIC;
P32_leveli: in STD_LOGIC; -- P3 inputs
P33_leveli: in STD_LOGIC;
P34_edgei: in STD_LOGIC;
P35_edgei: in STD_LOGIC);
end entity te51TMR;
Techniques intended to reduce the impact of program-flow errors on embedded systems

architecture bhv of te51TMR is
  alias MODE0: STD_LOGIC_VECTOR(1 downto 0) is TMODi(1 downto 0);
  alias CT0: STD_LOGIC is TMODi(2);
  alias GATE0: STD_LOGIC is TMODi(3);
  alias MODE1: STD_LOGIC_VECTOR(1 downto 0) is TMODi(5 downto 4);
  alias CT1: STD_LOGIC is TMODi(6);
  alias GATE1: STD_LOGIC is TMODi(7);
  alias TRO: STD_LOGIC is TCONi(4);
  alias TR1: STD_LOGIC is TCONi(6);
  signal T0_pulse: STD_LOGIC;  -- T0 receives pulse
  signal T1_pulse: STD_LOGIC;  -- T1 receives pulse
  signal T0_ctrl: STD_LOGIC;   -- T0 control
  signal T1_ctrl: STD_LOGIC;   -- T1 control
  signal T0_MODE3: STD_LOGIC;  -- signify T0 mode3
  signal T1_MODE3: STD_LOGIC;  -- signify T1 mode3
  signal UPDATE: STD_LOGIC;    -- high to update TLx/THx

begin
  UPDATE <= '1' when ((csp = C1S2P2) or (csp = C2S2P2)) else '0';
  T0_MODE3 <= '1' when MODE0 = '11' else '0';  -- mode 3 for TMR0
  T1_MODE3 <= '1' when MODE1 = '11' else '0';  -- mode 3 for TMR1
  T0_ctrl <= ((not GATE0) or P32_leveli) and TRO;  -- TMR0 increment control
  T0_pulse <= '1' when CT0 = '0' else P34_edgei;   -- suitable pulse control
  T0_UPDATEo <= T0_ctrl and T0_pulse and UPDATE;  -- enable TMR0 to update
  T1_ctrl <= ((not GATE1) or P33_leveli) and TR1;  -- TMR1 increment control
  T1_pulse <= '1' when CT1 = '0' else P35_edgei;   -- suitable pulse receive
  T1_UPDATEo <= ((not T0_MODE3) and (not T1_MODE3) and
                 and T1_ctrl and T1_pulse and UPDATE)
                or (T0_MODE3 and (not T1_MODE3) and UPDATE);
end architecture;

Listing H.3: Source of te51TMR

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.te51pak.all;

entity te51TMR_INC is
  port (clk: in STD_LOGIC;  -- clock
        rst: in STD_LOGIC;   -- reset
        TMODi: in STD_LOGIC_VECTOR(7 downto 0);  -- incrementor input
        TCONi: in STD_LOGIC_VECTOR(7 downto 0);
        TLOi: in STD_LOGIC_VECTOR(7 downto 0);
        THOi: in STD_LOGIC_VECTOR(7 downto 0);
        TLli: in STD_LOGIC_VECTOR(7 downto 0);
        THli: in STD_LOGIC_VECTOR(7 downto 0);
        TLOo: out STD_LOGIC_VECTOR(7 downto 0);  -- incrementor output
        THOo: out STD_LOGIC_VECTOR(7 downto 0);
        TLlo: out STD_LOGIC_VECTOR(7 downto 0);
        THlo: out STD_LOGIC_VECTOR(7 downto 0);
        TFOo: out STD_LOGIC;  -- TFO/1 interrupt flags
        TFlo: out STD_LOGIC);
end entity te51TMR_INC;

architecture bhv of te51TMR_INC is
  signal T0_MODE3: STD_LOGIC;  -- to determine T0 mode 3
  signal T1_MODE3: STD_LOGIC;  -- to determine T0 mode 2
  signal T1_MODE2: STD_LOGIC;  -- to determine T1 mode 2
  signal TMRO_13i: STD_LOGIC_VECTOR(15 downto 0);  -- arranging bits
  signal TMRO_16i: STD_LOGIC_VECTOR(15 downto 0);
  signal INC0i: STD_LOGIC_VECTOR(15 downto 0);  -- input to incrementor
  signal TMPO: STD_LOGIC_VECTOR(8 downto 0);  -- incrementor use
  signal CINO: STD_LOGIC;  -- carry in
  signal INC0o: STD_LOGIC_VECTOR(15 downto 0);  -- incremented value

H-10
Techniques intended to reduce the impact of program-flow errors on embedded systems

```
signal RELOAD0: STD_LOGIC;  -- signal for TMR0 reload
signal OUTPUT0: STD_LOGIC_VECTOR(15 downto 0);  -- actual output
signal TMR1_13o: STD_LOGIC_VECTOR(15 downto 0);
signal TMR1o: STD_LOGIC_VECTOR(15 downto 0);

signal TMR1_13i: STD_LOGIC_VECTOR(15 downto 0);
signal TMR1_16i: STD_LOGIC_VECTOR(15 downto 0);
signal INC0i: STD_LOGIC_VECTOR(15 downto 0);  -- input to incrementor
signal TMP1: STD_LOGIC_VECTOR(8 downto 0);  -- incremented use
signal CIN1: STD_LOGIC;  -- carry in
signal INC0o: STD_LOGIC_VECTOR(15 downto 0);  -- incremented value
signal RELOAD1: STD_LOGIC;  -- signal for TMR1 reload
signal OUTPUT1: STD_LOGIC_VECTOR(15 downto 0);  -- actual output
signal TMR0_13o: STD_LOGIC_VECTOR(15 downto 0);  -- for arranging bits
signal TMR0o: STD_LOGIC_VECTOR(15 downto 0);  -- arranged bits

signal TFO_M0: STD_LOGIC;  -- TFO/1 flag conditions
signal TFO_M1: STD_LOGIC;
signal TFO_M2: STD_LOGIC;
signal TFO_M3: STD_LOGIC;
signal TFI_M0: STD_LOGIC;
signal TFI_M1: STD_LOGIC;
signal TFI_M2: STD_LOGIC;
signal TFI_M3: STD_LOGIC;
signal TFI_A: STD_LOGIC;

alias MODE0: STD_LOGIC_VECTOR(1 downto 0) is TMODi(1 downto 0);
alias MODE1: STD_LOGIC_VECTOR(1 downto 0) is TMODi(5 downto 4);
alias TR1: STD_LOGIC is TCONi(6);

begin
T0_MODE3 <= '1' when MODE0 = '11' else '0';  -- setting mode flags
T0_MODE2 <= '1' when MODE0 = '10' else '0';
T1_MODE2 <= '1' when MODE1 = '10' else '0';

TMR0_13i(4 downto 0) <= TLOi(4 downto 0);  -- arranging bits
TMR0_13i(12 downto 5) <= THOi;
TMR0_13i(15 downto 13) <= "000";
TMR0i(7 downto 0) <= TLOi;
TMR0i(15 downto 8) <= THOi;

TMR1_13i(4 downto 0) <= TLIi(4 downto 0);  -- takes care of TMR1
TMR1i(12 downto 5) <= THii;
TMR1i(15 downto 13) <= "000";
TMR1i(7 downto 0) <= TLIi;
TMR1i(15 downto 8) <= THii;

with MODE0 select INC0i <=
  TMR0_13i when "00",  -- mode 0
  TMR0i when others;
with MODE1 select INC0i <=
  TMR1_13i when "00",  -- mode 1, 2 and 3
  TMR1i when others;

CIN0 <= '1' when ((T0_MODE3 = '1') and (TR1 = '1')) else
  '0' when ((T0_MODE3 = '1') and (TR1 = '0')) else
  '0' when T0_MODE2 = '1' else
  TMP0(8);
CIN1 <= '0' when T1_MODE2 = '1' else
  TMP1(8);

TMP0 <= CONV_STD_LOGIC_VECTOR(CONV_INTEGER(INC0i(7 downto 0)) + 1), 9);
INC0o(7 downto 0) <= TMP0(7 downto 0);
INC0o(15 downto 8) <= INC0i(15 downto 8) + CIN0;

TMP1 <= CONV_STD_LOGIC_VECTOR(CONV_INTEGER(INC1i(7 downto 0)) + 1), 9);
INC1o(7 downto 0) <= TMP1(7 downto 0);
INC1o(15 downto 8) <= INC1i(15 downto 8) + CIN1;

WITH MODE0 SELECT INC00 <=
  TMR0_13i when "00",  -- connection to TMR0
  TMR0i when others;
WITH MODE1 SELECT INC00 <=
  TMR1_13i when "00",  -- connection to TMR1
  TMR1i when others;

CIN0 <= '1' when ((T0_MODE3 = '1') and (TR1 = '1')) else
  '0' when ((T0_MODE3 = '1') and (TR1 = '0')) else
  '0' when T0_MODE2 = '1' else
  TMP0(8);
CIN1 <= '0' when T1_MODE2 = '1' else
  TMP1(8);

OUTPUT0 <= RELOAD0 when ((T0_MODE3 = '1') and (INC0o(7 downto 0) = x'00'))
  else '0';  -- condition for reload
OUTPUT1 <= INC00 when RELOAD0 = '1';
else INC0o(7 downto 0);  -- LO output based on
OUTPUT1 <= INC0o(15 downto 8);
```

H-11
Techniques intended to reduce the impact of program-flow errors on embedded systems

Listing H.4: Source for te51TMR_INC

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;

entity te51P3_state is
port(
    clk: in STD_LOGIC; -- clock
    rst: in STD_LOGIC; -- reset
    csp: in te51stateT; -- csp
    P3XRi: in STD_LOGIC_VECTOR(7 downto 0); -- pad input
    IT0i: in STD_LOGIC; -- for external interrupt
    IT1i: in STD_LOGIC;
    IE0o: out STD_LOGIC;
);

end architecture bhv;
```

---

Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
RELOAD1 <= '1' when (T1_MODE2 = '1') and (INC1o(7 downto 0) = x"00")
else '0'; -- condition for reload
OUTPUT1(7 downto 0) <= INC1i(15 downto 8) when RELOAD1 = '1'
else INC1o(7 downto 0); -- LO output based on
OUTPUT1(15 downto 8) <= INC1o(15 downto 8);

TMR0_13o(4 downto 0) <= OUTPUT0(4 downto 0);
TMR0_13o(7 downto 5) <= "000";
TMR0_13o(15 downto 8) <= OUTPUT0(12 downto 5);

TMR1_13o(4 downto 0) <= OUTPUT1(4 downto 0);
TMR1_13o(7 downto 5) <= "000";
TMR1_13o(15 downto 8) <= OUTPUT1(12 downto 5);

with MODE0 select TMR0o <=
    TMR0_13o when "00",
    OUTPUT0 when others; -- mode 0
with MODE1 select TMR1o <=
    TMR1_13o when "00",
    OUTPUT1 when others; -- mode 1, 2 and 3

TL0o <= TMR0o(7 downto 0);
TM0o <= TMR0o(15 downto 8);
TL1o <= TMR1o(7 downto 0);
TH1o <= TMR1o(15 downto 8);

TF0_M0 <= '1' when ((TMR0o(4 downto 0) = "00000") and
(TMROo(15 downto 8) = x'00') else '0';
TF0_M1 <= '1' when (TMROo = x'0000') else '0';
TF0_M2 <= '1' when ((RELOADO = '1') and
(TMROo(7 downto 0) = TMR0o(15 downto 8))) else '0';
TF0_M3 <= '1' when (TMROo(7 downto 0) = x'00') else '0';

TF1_M0 <= '1' when ((TMR1o(4 downto 0) = "00000") and
(TMROo(15 downto 8) = x'00') else '0';
TF1_M1 <= '1' when (TMR1o = x'0000') else '0';
TF1_M2 <= '1' when ((RELOAD1 = '1') and
(TMROo(7 downto 0) = TMR0o(15 downto 8))) else '0';
TF1_M3 <= '1' when (TMROo(15 downto 8) = x'00') else '0';

with MODE0 select TF0o <=
    TF0_M0 when "00",
    TF0_M1 when "01",
    TF0_M2 when "10",
    TF0_M3 when others;
with MODE1 select TF1_A <=
    TF1_M0 when "00",
    TF1_M1 when "01",
    TF1_M2 when others;
with MODE0 select TF1o <=
    TF1_M3 when "11", -- TMR0 in mode 3
    TF1_A when others; -- TMR0 not in mode 3
```

---

Techniques intended to reduce the impact of program-flow errors on embedded systems
Techniques intended to reduce the impact of program-flow errors on embedded systems

IElo: out STD_LOGIC;
P32_levelo: out STD_LOGIC; -- for timer module
P33_levelo: out STD_LOGIC;
P34_edgeo: out STD_LOGIC;
P35_edgeo: out STD_LOGIC;
end entity te51P3_state;

architecture bhv of te51P3_state is
signal P3_old: STD_LOGIC_VECTOR(3 downto 0); -- history buffer
signal P3_new: STD_LOGIC_VECTOR(3 downto 0);
signal P32_edge: STD_LOGIC; -- edge detection
signal P33_edge: STD_LOGIC;
begin
  process(clk, rst)
  begin
    if rst = '1' then
      P3_old <= "0000";
      P3_new <= "0000";
    elsif rising_edge(clk) then
      case csp is
        when C1S4P2 | C2S4P2 =>
          P3_old <= P3_new;
        when C1S5P1 | C2S5P1 =>
          P3_new <= P3XRi(5 downto 2);
        when others =>
          null;
      end case;
    end if;
  end process;
  P32_edge <= (not P3_new(0)) and P3_old(0);
P33_edge <= (not P3_new(1)) and P3_old(1);
with ITOi select
  IElo <= P32_edge when '1',
  (not P3XRi(2)) when others;
with ITli select
  IEOo <= P33_edge when '1',
  (not P3XRi(3)) when others;
P32_levelo <= P3XRi(2); -- P3.2/3 level outputs
P33_levelo <= P3XRi(3);
P34_edgeo <= (not P3_new(2)) and P3_old(2);
P35_edgeo <= (not P3_new(3)) and P3_old(3);
end architecture bhv;

Listing H.5: Source of te51P3_STATE

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;

entity te51not_end_ins is
  port(
    clk: in STD_LOGIC;
    rst: in STD_LOGIC;
    mcodei: in te51mcodeT;
    csp: in te51stateT;
    NOT_END_INSo: out STD_LOGIC;
  );
end te51not_end_ins;

architecture bhv of te51not_end_ins is
signal cyclesl: BOOLEAN;
signal NOT_END_INS: STD_LOGIC;
begin
  with mcodei.be select
    cyclesl <=
    true when te51bc_lblc | te51bc_2blc,
    false when others;
  process(clk, rst)
  begin
    if rst = '1' then
      NOT_END_INS <= '0';
  end process;
end architecture bhv;
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
elsif rising_edge(clk) then
  case csp is
  when C1S5P1 =>
    NOT_END_INS <= '1';  -- set signal high
  when C1S5P2 =>
    if cyclesl then NOT_END_INS <= '0'; end if;
  when C2S5P2 =>
    NOT_END_INS <= '0';  -- reset signal
  when others =>
    null;
  end case;
end if;
end process;

NOT_END_INSo <= NOT_END_INS;
end architecture;
```

Listing H.6: source of te51 NOT_END_INS

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;

entity te51INT_POLL is
  port(
    elk: in STD_LOGIC; -- clock
    rst: in STD_LOGIC; -- reset
    csp: in te51stateT;
    TCONi: in STD_LOGIC_VECTOR(7 downto 0); -- TCON
    INTO: out STD_LOGIC_VECTOR(6 downto 0)  )  ;-- polled interrupt
end entity te51INT_POLL;

architecture bhv of te51INT_POLL is
  signal INT_old: STD_LOGIC_VECTOR(6 downto 0 );
  signal INT_new: STD_LOGIC_VECTOR(6 downto 0 );

  alias IE0: STD_LOGIC is TCONi(1);  -- aliases for flags
  alias IE1: STD_LOGIC is TCONi(3);
  alias TF0: STD_LOGIC is TCONi(5);
  alias TF1: STD_LOGIC is TCONi(7);

begin
  process(clk, rst)
  begin
    if rst = '1' then
      INT_old <= "0000000"; INT_new <= "0000000"; -- reset history registers
    elsif rising_edge(clk) then
      case csp is
        when C1S5P1 | C2S5P1 =>
          INT_old <= INT_new;
        when C1S5P2 | C2S5P2 =>
          INT_new(0) <= IE0;
          INT_new(1) <= TF0;
          INT_new(2) <= IE1;
          INT_new(3) <= TF1;
        when others =>
          null;
      end case;
    end if;
  end process;

  INTO(0) <= INT_new(0) and INT_old(0);  -- if flags high two reads,
  INTO(1) <= INT_new(1) and INT_old(1);
  INTO(2) <= INT_new(2) and INT_old(2);
  INTO(3) <= INT_new(3) and INT_old(3);
end architecture bhv;
```

Listing H.7: Source of te51INT_POLL
Techniques intended to reduce the impact of program-flow errors on embedded systems

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;
use work.te51pak2.all;

entity te51INT is
  port(
    clk: in STD_LOGIC; -- clock
    rst: in STD_LOGIC; -- reset
    csp: in te51stateT; -- cycle state information
    IT0i: in STD_LOGIC; -- SFR inputs
    IT1i: in STD_LOGIC;
    IP1i: in STD_LOGIC_VECTOR(7 downto 0);
    IP2i: in STD_LOGIC_VECTOR(7 downto 0);
    INT1i: in STD_LOGIC_VECTOR(6 downto 0); -- polled interrupt flags
    NOT_END_INSi: in STD_LOGIC;
    IP_IEi: in STD_LOGIC;
    Op1i: in STD_LOGIC_VECTOR(7 downto 0); -- 1st byte of instruction
    IE0o: out STD_LOGIC; -- flag output to SFR
    IE1o: out STD_LOGIC;
    TF0o: out STD_LOGIC;
    TF1o: out STD_LOGIC;
    INT_VECo: out STD_LOGIC_VECTOR(23 downto 0); -- interrupt vector
    INT_LCALLo: out STD_LOGIC); -- signal to the processor
end entity te51INT;

architecture bhv of te51INT is
  alias EX0: STD_LOGIC is IEi(0); -- SFR bit alias
  alias ET0: STD_LOGIC is IEi(1);
  alias EX1: STD_LOGIC is IEi(2);
  alias ET1: STD_LOGIC is IEi(3);
  alias ES: STD_LOGIC is IEi(4);
  alias EA: STD_LOGIC is IEi(7);
  alias PX0: STD_LOGIC is IPi(0);
  alias PT0: STD_LOGIC is IPi(1);
  alias PX1: STD_LOGIC is IPi(2);
  alias PT1: STD_LOGIC is IPi(3);
  alias PS: STD_LOGIC is IPi(4);

  signal BLOCKING: STD_LOGIC; -- blocking condition
  signal INT_IE: STD_LOGIC_VECTOR(6 downto 0); -- output of IE
  signal INT_IP: STD_LOGIC_VECTOR(13 downto 0); -- output of IP
  signal PRI0_HI: STD_LOGIC_VECTOR(2 downto 0); -- output of encoder
  signal PRI0_LO: STD_LOGIC_VECTOR(23 downto 0); -- output of encoder
  signal INT_LO_DET: STD_LOGIC; -- lo priority interrupt
  signal INT_HI_DET: STD_LOGIC; -- hi priority interrupt
  signal INT_ANY_HDL: STD_LOGIC;
  signal INT_ALL_HDL: STD_LOGIC;
  signal INT_DET: STD_LOGIC; -- interrupt detected
  signal INT_VEC: STD_LOGIC_VECTOR(23 downto 0); -- interrupt vector
  signal INT_VEC_LC: STD_LOGIC_VECTOR(23 downto 0); -- actual interrupt vector
  signal INT_LCALL: STD_LOGIC; -- handle int. LCALL
  signal RETI_INS: STD_LOGIC;
  signal INT_STATE: te51INT_STATE; -- interrupt state
  signal INT_HDL: te51INT_HDL; -- interrupt handling state

begin
  process(clk, rst) -- process SFRs
  begin
    if rst = '1' then
      INT_STATE <= te51INT_STATE_NONE; -- reset int. state
      INT_HDL <= te51INT_HDL_NONE; -- reset int. handling
      INT_LCALL <= '0';
      INT_VEC_LC <= x"00000001";
      IE0o <= '0'; IE1o <= '0'; TF0o <= '0'; TF1o <= '0';
    elsif rising_edge(clk) then
      case csp is
        when C2S1P1 =>
          if RETI_INS = '1' then -- check for RETI
            if INT_STATE = te51INT_STATE_LOHI then
              INT_STATE <= te51INT_STATE_LO; -- reduce to lo priority
            end if;
          end if;
        when others =>
      end case;
    end if;
  end process;
end architecture;
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
else
    INT_STATE <= te51INT_STATE_NONE; -- reduce to no interrupts
end if;
end if;
when C1S1P2 | C2S1P2 => -- reset "flags for TCON"
    IE00 <= '0'; IE10 <= '0'; TF00 <= '0'; TF10 <= '0';
when C2S3P1 =>
    INT_LCALL <= '0';
when C1S5P2 | C2S5P2 =>
    if INT_HDL = te51INT_HDL_LCl then
        INT_VEC_LC(7 downto 0) <= x"12";
        INT_VEC_LC(15 downto 8) <= x"00";
        INT_VEC_LC(23 downto 22) <= "00";
        INT_VEC_LC(21 downto 19) <= INT_VEC;
        INT_VEC_LC(18 downto 16) <= "011"; -- the 3 LSB of the LCALL
    end if;
when C1S6P2 | C2S6P2 =>
    case INT_HDL is
        when te51INT_HDL_NONE =>
            if INT_DET = '1' then
                INT_LCALL <= '1';
            end if;
        when te51INT_HDL_LCl =>
            INT_HDL <= te51INT_HDL_LC2;
        when te51INT_HDL_LC2 =>
            if INT_VEC_LC(5 downto 3) = "000" then
                ITOi <= '1';
            elsif INT_VEC_LC(5 downto 3) = "010" then
                TF0o <= '1';
            elsif INT_VEC_LC(5 downto 3) = "011" then
                TF1o <= '1';
            end if;
        when others =>
            INT_HDL <= te51INT_HDL_NONE;
    end case;
end when others =>
    null;
end case;
end if;
end process;
```
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
INT_IP(3) <= INT_IE(3) when PT1 = '0' else '0';
INT_IP(9) <= INT_IE(4) when PS = '1' else '0'; -- ES interrupt
INT_IP(2) <= INT_IE(4) when PS = '0' else '0';
INT_IP(8) <= '0';
INT_IP(1) <= '0';
INT_IP(7) <= '0';
INT_IP(0) <= '0';

PRIO_HI <= "000' when INT_IP(13) = '1' else
            '001' when INT_IP(12) = '1' else
            '011' when INT_IP(10) = '1' else
            '101' when INT_IP(9) = '1' else
            '111' when INT_IP(7) = '1' else
            '111';

PRIO_LO <= '000' when INT_IP(6) = '1' else
            '001' when INT_IP(5) = '1' else
            '011' when INT_IP(3) = '1' else
            '110' when INT_IP(2) = '1' else
            '111';

INT_HI_DET <= '0 when PRIO_HI = "111" else '1'; -- interrupt detected
INT_LO_DET <= '0 when PRIO_LO = "111" else '1'; -- interrupt detected
INT_VEC <= PRIO_HI when INT_HI_DET = '1' else -- selected interrupt
            PRIO_LO;

with INT_STATE select INT_ANY_HDL <=
    '1' when te51INT_STATE_NONE,
    '1' when te51INT_STATE_HI,
    '0' when others;

with INT_STATE select INT_ALL_HDL <=
    '1' when te51INT_STATE_NONE,
    '0' when others;

INT_DET <= ((INT_ANY_HDL and INT_ALL_HDL and (INT_HI_DET or INT_LO_DET)) or
             (INT_ANY_HDL and (not INT_ALL_HDL) and INT_HI_DET)) and
            (not BLOCKING);

INT_LCALLo <= INT_LCALL;
INT_VECo <= INT_VEC_LC;
end architecture bhv;
```

Listing H.8: Source of te51INT

Note: the source files can be found on the CD-ROM (see Appendix R). The source of the structural connections (shown in Figure H.1) is also found on the CD-ROM.
Appendix I  MCU Testbench commands and scripting interface

This appendix describes the MCU Testbench and MCU Host commands (see Chapter 14), scripting variables and scripting functions.

The commands to the testbench can be grouped into three classes: control commands, error injection commands and observer commands.

I.1  MCU Host to MCU Testbench command format

All commands from the host to the testbench are six bytes wide and sent in the format shown in Table I.1. This common format makes it easy to design the COMINT mode. Note: data is transferred MSB first from byte 0 to byte 5.

<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Module Command</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>32-bit operand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Checksum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table I.1: Host to MCU Testbench command format

Each command from the host to the testbench starts with a 1 and 0 (byte 0 bit 7 and 6). The module selection bits – shown in Table I.2 – (Byte 0 bit 5 and bit 4) selects the module that is going to receive the command.

<table>
<thead>
<tr>
<th>Module</th>
<th>Bit 5</th>
<th>Bit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ERR</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OBS</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table I.2: Module selection bit format
As the command field is four bits wide, each module can have up to 16 commands. The commands and the numbers for all modules are shown in Table I.3.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Operand Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>Stop TE-51</td>
<td>-</td>
</tr>
<tr>
<td>RUN</td>
<td>Run TE-51</td>
<td>-</td>
</tr>
<tr>
<td>STEP</td>
<td>Step execute TE-51</td>
<td>-</td>
</tr>
<tr>
<td>MICROSTEP</td>
<td>Microstep execute TE-51</td>
<td>-</td>
</tr>
<tr>
<td>RUNTO</td>
<td>Run until RUNTO clock-cycle</td>
<td>-</td>
</tr>
<tr>
<td>RESETMCU</td>
<td>Reset TE-51</td>
<td>-</td>
</tr>
<tr>
<td>SETUPRES</td>
<td>Set prescaler value</td>
<td>8</td>
</tr>
<tr>
<td>SETUPRUNTO</td>
<td>Set RUNTO value</td>
<td>32</td>
</tr>
<tr>
<td>SETUPECY</td>
<td>Set cycle to inject error value</td>
<td>32</td>
</tr>
<tr>
<td>SETUPESC</td>
<td>Set sub-cycle to inject error value</td>
<td>4</td>
</tr>
<tr>
<td>SETUPNPC</td>
<td>Set PC error injection value</td>
<td>16</td>
</tr>
<tr>
<td>ERREN</td>
<td>Enable module</td>
<td>-</td>
</tr>
<tr>
<td>ERRDIS</td>
<td>Disable module</td>
<td>-</td>
</tr>
<tr>
<td>RESETERR</td>
<td>Reset module</td>
<td>-</td>
</tr>
<tr>
<td>OBS</td>
<td>Observer register/status</td>
<td>32</td>
</tr>
</tbody>
</table>

Table I.3: MCU Testbench Commands

The operand field is 32-bits wide. This enables even the longest parameter to be transferred in a single command.

The check command of the OBS module has a different operand format as shown in Table I.4. The 32-bit operand is considered as two 16-bit operands: byte 1 and byte 2 forms the parameter to observe while byte 3 and byte 4 are the address field. The latter is only necessary when RAM or SFR is selected for observation.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Table 1.4: Format of OBSERVE command (OBS module)

<table>
<thead>
<tr>
<th>Byte</th>
<th>Value</th>
<th>Parameter to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 and 2</td>
<td>1</td>
<td>RAM (address in byte 2/3)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>SFR (address in byte 2/3)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Clock-cycles executed</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Sub-cycle state of MCU (SC)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Program Counter value</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>TE-51 running state</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Error injection state</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Cycle to inject error value</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Sub-cycle to inject error value</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Erroneous PC value</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>MCU prescaler value (PRES)</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>Breakpoint value (RUNTO)</td>
</tr>
<tr>
<td>3 and 4</td>
<td>0 to 255</td>
<td>RAM/SFR address</td>
</tr>
</tbody>
</table>

The checksum (byte 5 – see Table 1.1) is the XORed value of all the previous bytes.

I.2 MCU Testbench to MCU Host command format

The testbench only sends data to the host system on two occasions:
- When the OBS module receives an OBSERVE command from the host.
- When TE-51’s execution stops.

The value of the observed parameter is transmitted according to the format shown in Table 1.5. Note: data is transmitted MSB first from byte 0 to byte 5.

Table 1.5: MCU Testbench to host command format

<table>
<thead>
<tr>
<th>Byte</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

0xA5 or 0xA0
Operand
Checksum
Techniques intended to reduce the impact of program-flow errors on embedded systems

Byte 0 of the MCU Testbench to the MCU Host command is a header that must either be 0xA5 or 0xA0. If the MCU Testbench transmits a command as a reply to an OBSERVE command, 0xA5 is sent as the header (first situation in the previous list). The next four bytes are the value of the parameter selected by the OBSERVE command, followed by the checksum. The checksum is the XORed value of all the previous bytes.

If the MCU Testbench transmits a command when TE-51’s execution is stopped, 0xA0 is used as the command header instead. Bit 0 and 1 of byte 2 is set if the MCU is running or an error has been injected, respectively. As before, the checksum is the XORed value of all the previous bytes.

1.3 Scripting variables and commands

The scripting variables and commands are separated into four categories:

- Testbench variables
- TE-51 SFRs
- Testbench commands
- Scripting functions

All of these categories are described in their respective sub-sections.
Techniques intended to reduce the impact of program-flow errors on embedded systems

### 1.3.1 Testbench variables

Testbench variables are those that keep track of TE-51's execution, and the error injection system. They must be prefixed with `tc`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>R/W</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CY</td>
<td>Long</td>
<td>R</td>
<td>Instruction cycles executed</td>
<td>0 - 2147483647</td>
</tr>
<tr>
<td>SC</td>
<td>Integer</td>
<td>R</td>
<td>Sub-cycle state</td>
<td>0 - 11</td>
</tr>
<tr>
<td>PC</td>
<td>Long</td>
<td>R</td>
<td>Program Counter value</td>
<td>0 - 65535</td>
</tr>
<tr>
<td>PRES</td>
<td>Integer</td>
<td>R/W</td>
<td>Prescaler setting</td>
<td>0 - 255</td>
</tr>
<tr>
<td>RUNTO</td>
<td>Long</td>
<td>R/W</td>
<td>Hardware breakpoint cycle</td>
<td>0 - 2147483647</td>
</tr>
<tr>
<td>ECY</td>
<td>Long</td>
<td>R/W</td>
<td>Instruction cycle to inject error</td>
<td>0 - 2147483647</td>
</tr>
<tr>
<td>ESC</td>
<td>Integer</td>
<td>R/W</td>
<td>Sub-cycle to inject error</td>
<td>0 - 11</td>
</tr>
<tr>
<td>EPC</td>
<td>Long</td>
<td>R</td>
<td>Erroneous PC value</td>
<td>0 - 65535</td>
</tr>
<tr>
<td>MCUnin</td>
<td>Integer</td>
<td>R</td>
<td>MCU run status</td>
<td>0, 1</td>
</tr>
<tr>
<td>ERRinj</td>
<td>Integer</td>
<td>R</td>
<td>Error injection status</td>
<td>0, 1</td>
</tr>
</tbody>
</table>

Table 1.6: MCU Testbench variables and constants accessible by scripts

Note: * 'R' for read-only variables, 'R/W' for read/write variables.

### 1.3.2 MCU Testbench special function registers

All 8051-generic SFRs apart from SBUF and SCON can be read. These SFRs must be prefixed with `tc` (e.g. reading the Accumulator would be written as `x = tc.ACC`).

<table>
<thead>
<tr>
<th>SFR</th>
<th>Description</th>
<th>SFR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Accumulator</td>
<td>PCON</td>
<td>Power Control</td>
</tr>
<tr>
<td>B</td>
<td>B register</td>
<td>PSW</td>
<td>Program Status Word</td>
</tr>
<tr>
<td>DPH</td>
<td>Data pointer (low byte)</td>
<td>SP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>DPL</td>
<td>Data pointer (high byte)</td>
<td>TCON</td>
<td>Timer Control</td>
</tr>
<tr>
<td>IE</td>
<td>Interrupt Enable</td>
<td>TH0</td>
<td>T0 high register</td>
</tr>
<tr>
<td>IP</td>
<td>Interrupt Priority</td>
<td>TH1</td>
<td>T1 high register</td>
</tr>
<tr>
<td>P0</td>
<td>Port 0 latch</td>
<td>TL0</td>
<td>T0 low register</td>
</tr>
<tr>
<td>P1</td>
<td>Port 1 latch</td>
<td>TL1</td>
<td>T1 low register</td>
</tr>
<tr>
<td>P2</td>
<td>Port 2 latch</td>
<td>TMOD</td>
<td>Timer Mode</td>
</tr>
<tr>
<td>P3</td>
<td>Port 3 latch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1.7: SFRs accessible by scripts (type 'byte')
Techniques intended to reduce the impact of program-flow errors on embedded systems

1.3.3 Testbench commands
The Testbench commands are encapsulated in the same way as scripting functions and called like so. Unless explicitly stated, these commands return a –1 when execution fails. Successfully executed commands return a non-negative value. Testbench commands must be prefix by ‘tc’.

**cmdRUNO() As Integer**
- Description: Runs MCU
- Parameters: none
- Return value: 1 - when successful

**cmdSTEP() As Integer**
- Description: Runs MCU for one clock-cycle
- Parameters: none
- Return value: 1 - when successful

**cmdMICROSTEP() As Integer**
- Description: Runs MCU for one sub-cycle
- Parameters: none
- Return value: 1 - when successful

**cmdRUNTO() As Integer**
- Description: Runs MCU until CY = RUNTO
- Parameters: none
- Return value: 1 - when successful

**cmdSTOP() As Integer**
- Description: Stops MCU
- Parameters: none
- Return value: 1 - when successful

**cmdRESETMCU() As Integer**
- Description: Reset MCU
- Parameters: none
- Return value: 1 - when successful

**cmdERREN(state As Integer) As Integer**
- Description: Enable Error Injection System
- Parameters: state 0 for no MCU stop after error injection,
- Parameters: state 1 for MCU stop after error injection
- Return value: 1 - when successful

**cmdERRDIS() As Integer**
- Description: Disables Error Injection System
- Parameters: none
- Return value: 1 - when successful
Techniques intended to reduce the impact of program-flow errors on embedded systems

**cmdRESETERR() As Integer**
Description: Reset Error Injection System
Parameters: none
Return value: 1 – when successful

### 1.3.4 Scripting functions

The scripting functions are support functions to enhance the usefulness of the Testbench.

Unless explicitly stated, these functions return a –1 when execution fails. Successfully executed functions return a non-negative value. Scripting function must be prefixed by 'tc'.

**CloseFile(file_num As Integer) As Integer**
Description: Closes the file pointed by file_num
Parameters: file_num file number to close
Return value: 1 - when successful

**Complement(value As Byte) As Integer**
Description: Complements a single byte value
Parameters: value value to complement
Return value: Complemented value

**CreateFile(filename As String) As Integer**
Description: Create a text file
Parameters: filename name of file to create
Return value: +ve value - file number, -1 - file cannot be created

**Echo(file_num As Integer, str As String) As Integer**
Description: Write a line into a text file
Parameters: file_num file number to write
str string to write
Return value: 1 - on successful writing

**GetSFRVal(address As Integer) As Integer**
Description: Reads an SFR location
Parameters: address location of SFR to read (only generic 8051 SFRs)
Return value: SFR value

**GetUserInput(prompt As String, title As String) As String**
Description: Prompts the user to enter a value into the dialog box
Parameters: prompt text message to display for user input
title Dialog Box title
Return value: user input as string
Techniques intended to reduce the impact of program-flow errors on embedded systems

Hex2Dec(hex As String, length As Integer) As Long
Description: Converts a hexadecimal string into a decimal equivalent number
Parameters: hex hexadecimal string
length number of characters to convert (starting from 1st character)
Return value: decimal equivalent of the string

InsByteCycle_I (ins As Byte) As Integer
Description: Returns the instruction length and cycle time.
Parameters: ins Instruction hex code.
Return: 11 - 1B 1C, 12 - 1B 2C, 14 - 1B 4C,

InsSize_I(ins As Byte) As Integer
Description: Length of instruction
Parameters: ins instruction hex code
Return value: instruction length

OpenFile(filename As String, mode As Integer) As Integer
Description: Opens a file for reading/appending
Parameters: filename filename of the file to open/append
mode 0 for writing, 1 for appending
Return value: +ve value - file number, -1 - file cannot be created

ReadFile(file_num As Integer) As String
Description: Reads a line from a text file
Parameters: file_num file number to read
Return value: read line, "" if read not successful

WriteFile(file_num As Integer, str As String) As Integer
Description: Write a line into a text file
Parameters: file_num file number to write
str string to write
Return value: 1 - on successful writing
Appendix J  Description of STATS

STATS is a command-line program written in Microsoft’s Visual C++ 6.0. It requires one parameter: the filename of the batch file. This batch file lists the filename of the programs (in Intel Hex8 format) to process, along with the relevant arguments. STATS is executed by invoking the following command:

\[ \text{STATS} \ <\text{batch\_filename}> \]

Example executions are shown below. The format for the batch file is described towards the end of this appendix.

\begin{verbatim}
STATS input.txt
STATS f:\research\experiments\NF_n_FT\files.txt
\end{verbatim}

STATS sequentially processes each program listed in the batch file. STATS terminates if a listed program is not found, or if the harddrive is full (it creates other files).

Since the same processing occurs for all programs, only one iteration is described.

STATS clears its memory buffer before loading the program. It then goes through each instruction in the buffer, categorising it according to type S, DN, D2, TN, T2 and T23. The number of conditional and unconditional branch instructions where the branching location can and cannot be ascertained (in advance), is also listed\(^{105}\). All these statistics are saved in the results file: ‘stats results.txt’ (generated automatically). In addition, all code memory locations that are not on the instruction cycle boundary (potential MIT errors) are saved in files with the following filename format:

\[ <\text{program\_filename}>.\text{mit.\_txt} \]

where \text{program\_filename} is the program’s filename (excluding the ‘.hex’ extension). The file generated is in a format that can immediately be used in the dScope scripts such as the experiments carried out in Chapter 7 (see Listing N.1).

\(^{105}\) The branching address of the JMP, RET and RETI instructions cannot be determined at compile time.
Two other files (‘.jmp’ and ‘.par’ extensions) are subsequently created. The former lists the ‘take-off’ and ‘landing’ addresses of all conditional and unconditional branches (except when the ‘landing’ address cannot be determined). Both files are in a format originally intended for some futile error detection techniques, and can be ignored.

The batch file must be written in the following format:

- Each line must be for only one program.
- Each program can have the following switches: /EX0=1, /EX1=1, /ET0=1, /ET1=1, /ET2=1 or /ES=1. These switches are only used by the ‘.jmp’ files.
- The batch file must be in pure text format (i.e. created with 'Notepad').

A sample listing of the batch file is shown in Listing J.1. With this batch file, STATS would process four files in the same order described.

```
alarm_a.hex
alarm_b.hex /ET0=1
alarm_c.hex /ET0=1 /EX0=1
alarm_d.hex
```

Listing J.1: Example of batch file entry

The source codes for STATS are located on the CD-ROM (see Appendix R).
Appendix K  Field Programmable Gate Arrays

FPGAs are integrated circuits that are made up of many configurable logic blocks arranged in a 2-dimensional array. Each CLB comprises of look-up-tables, multiplexors and D-type flip-flop/latches, as shown in Figure K.1 that can be configured in many different ways. One or more CLBs would be configured to form the desired logic function (e.g. binary counter, register).

All the connections between CLBs are programmable to allow communications between them. Each CLB is connected with all the adjacent CLBs. For non-adjacent CLBs to be connected, the buses that run along the CLBs can be programmed in a specific manner to link two or more distant CLBs together. This is carried out by the programmable switching matrix (PSM), as shown in Figure K.2.

The PSM can link horizontal and vertical buses, or every second or forth PSM together. This connection hierarchy avoids routing signals between CLBs through too many switching points [Van den Bout 1999].
Techniques intended to reduce the impact of program-flow errors on embedded systems

![Block diagram of an FPGA showing arrays of CLBs and PSMs](image)

Figure K.2: Block diagram of an FPGA showing arrays of CLBs and PSMs [Van den Bout 1999]

At the sides of the FPGA die are input/output blocks (IOBs) that allow connections between the CLBs/PSMs with the integrated circuit pins. The IOBs are also programmable to different logic standards (e.g. TTL, LVTTL, ECL). Hence, by programming the CLBs, PSMs and IOBs correctly, very complicated logic structures (e.g. processors) can be created.

Note: newer FPGAs include additional components such as delay-locked loops for clock multiplication and real RAM cells (as opposed to using the CLBs as RAM) [Xilinx 2001].
Appendix L  Integrating peripherals with processors

Peripherals are hardware modules performing specific tasks that are controllable by the processor. Peripherals that are found on MCUs include timers, USARTs, SPI controllers, pulse width modulators and LCD (liquid crystal display) controllers.

All peripherals have one or more registers that can be accessed by the processor core. When a program needs to configure a peripheral, it writes certain values into registers that are specific to each peripheral. These registers – commonly known as special function registers – are read by the respective peripherals at specific clock-cycles to determine its course of action. On specific cycles, or due to certain stimulus, the peripherals can also change certain SFRs to inform the processor of its status. This basic form of message passing, illustrated in Figure L.1, constitutes the peripheral-processor communication link, which makes it easy to integrate peripherals with the processor [Cruttenden 1987].

The SFR accesses by the processor core and peripherals must occur at different times to prevent contention, which may corrupt the SFR, or cause electrical problems. Alternatively, these SFR locations can be created from dual-access memory, which is used in the Motorola MCU families (e.g. M68HC16) [Melear 1995].

![Figure L.1: Peripheral control via SFRs](image-url)
There are several advantages of implementing error detection and/or correction modules as peripherals (e.g. CBG, HWFT, SP) instead of the chip-based (hard-wired) approach (e.g. ST, ST2 and BCT). These advantages are as follows:

- **Portability** – Peripherals are self-contained and have common interfaces [Bursky 1994]. Hence, they can be reused by different processor families with little or no modifications, especially if they adhere to certain interface protocols (e.g. ARMs’ AMBA – Advance Microcontroller Bus Architecture [Jones 2002]).

- **Cost** – The reusability [Melear 1995, Bursky 1994] of peripheral usually translates to reduce implementation time. This will directly affect a product’s cost.

- **Controllability** – Peripherals can be actively controlled to behave differently at different times. This means modules such as filters can have their parameters changed under different conditions (adaptive filtering).

- **Multi-level error detection coverage** – Implementing peripheral-based techniques allow errors detected at the chip (or higher) level to be handled at the software level. Hence, simple circuit-level techniques can be complemented with complex software-based schemes and vice versa.
Techniques intended to reduce the impact of program-flow errors on embedded systems

Appendix M  NF and FT’s simulation script

NF and FT’s simulation script for the experiments carried out in Chapter 4 is written for dScope. This script is made up of four files. The MIT checking functions (Listing M.4) have been shortened (the complete scripts are on the CD-ROM – see Appendix R). Only Prog_A’s script is described, the rest are similar.

load 8051.dll /* driver */
load prog_a /* program */
include init.ini /* initialisation file */
include comfunc.ini /* common functions */
include mit.ini /* MIT location list */

func void ERROR_TEST(){
  uint loop;
  exec("log > dlog.txt"); /* makes log file */
  exec("log off");
  printf("IP ERROR Simulation... by HLR Ong.

");
  for(loop=0;loop<NUM_TEST;loop++)
  {
    ERROR_COUNT++;
    exec("bk *");
    GEN_ERROR(); /* MIT check (broken into multiple files due to dScope’s limit */
    if((ERROR_POS>=0)&&(ERROR_POS<0x0500)) LO_CHECK();
    if((ERROR_POS>=0x0500)&&(ERROR_POS<START_NF)) HI_CHECK();
    if((ERROR_POS>=START_NF)&&(ERROR_POS<=END_NF)) NF_ERROR();
    if((ERROR_POS>END_NF)&&(ERROR_POS<=END_CODE)) END_CHECK();
  }
  ERROR_SUMMARY();
}

Listing M.1: Main file

xtal=20000000
define button "GOTO Main", "u main" /* defining buttons */
define button "GO", "g"
declare button "STOP", ".break_=1"
declare button "START TEST", "ERROR_TEST()"
declare ulong END_CODE /* end of code memory */
declare ulong START_NF /* start of NF region */
declare ulong END_NF /* end of NF region */
declare ulong NUM_TEST /* number of test cycles */
declare ulong MAX_CYCLES /* max. cycles (ECY) */
declare ulong CON_CYCLES /* max. extra cycles */

END_CODE=0fffffff
START_NF=07dffH
END_NF=0fffffff
NUM_TEST=200
MAX_CYCLES=10000000
CON_CYCLES=50000

define uint NF_ERROR /* NF counter */
define uint FT_ERROR /* FT counter */
define uint MIT_ERROR /* MIT counter */
define uint CON_ERROR /* Con counter */
define uint ERROR_COUNT /* Error counter */
Techniques intended to reduce the impact of program-flow errors on embedded systems

```c
#define uint ERROR_POS /* Error jump location */
#define ulong ERROR_CYCLE

NF_ERROR=0; /* initialising */
FT_ERROR=0;
MIT_ERROR=0;
CON_ERROR=0;
ERROR_COUNT=0;

Listing M.2: Initialising file (init.ini)

func void GEN_ERROR(){ /* generates error */
    uint loop;
    do {
        ERROR_CYCLE=(rand(0)/32768.0)*10000000.0;
    } while( ERROR_CYCLE>MAX_CYCLES);
    do {
        ERROR_POS=rand(0)/8;
    } while(ERROR_POS>END_CODE);
    /* generates error */
    /* generate ECY */
    printf("ERROR_CYCLE #%d = %lu ERROR_COUNT, ERROR_CYCLE) ;
    printf("ERROR_POS = %lu\n", ERROR_POS);
    exec("bs cycles==ERROR_CYCLE") ;
    exec("bs cycles==ERROR_CYCLE+l")
    exec("bs cycles==ERROR_CYCLE+2")
    exec("bs cycles==ERROR_CYCLE+3")
    exec("g");
    $=ERROR_POS;
}

func void PRINT_ERROR(int error_type){
    exec("log » dlog.txt");
    switch(error_type)
    {
        case 1: printf("NF") ;  break;
        case 2: printf("FT") ;  break;
        case 3: printf("MIT") ; break;
        case 4: printf("CON") ; break;
    }
    printf(" ERROR #%d at $=%#x", ERROR_COUNT, $);
    printf(" vectoring to $=%#x", ERROR_POS);
    printf(" at cycle %lu\n \n", ERROR_CYCLE);
    exec("log off");
}

func void NF_ERROR(){
    NF_ERROR++;
    PRINT_ERROR(1);
    exec("reset");
}

func void MIT_ERROR(){
    MIT_ERROR++;
    PRINT_ERROR(3);
    exec("reset");
}

func void CON_ERROR(){
    ulong loop;
    int PT;
    int MIT;
    FT=0;
    MIT=0;
    for(loop=0;loop<CON_CYCLES;loop++) {
        exec("t");
        if((internal_timer_counter!=0x001e))||((&$&0xffff)>END_CODE)||
            (timer_counter>0x001e) {"}
```

M-2
Techniques intended to reduce the impact of program-flow errors on embedded systems

```c
printf("MIT BREAK\n") ;
MIT=1;
break;
}

if(P2<0x80) { /* check FT is activated */
    printf("FT BREAK\n") ;
    FT=1;
    break;
}

if(FT==1)  {
    FT_ERROR++;
    PRINT_ERROR(2);
}  
else {
    if(MIT==1)  {
        MIT_ERROR++;
        PRINT_ERROR(3);
    } 
    else {
        CON_ERROR+ +;
        PRINT_ERROR(4);
    } 
    exec("reset");  
}

func void ERROR_SUMMARY(){
    exec("log >> diag.txt");
    printf("\n\nNF_ERROR :  %u\n",  NF_ERROR);
    printf("FT_ERROR :  %u\n", FT_ERROR);
    printf("MIT_ERROR :  %u\n",  MIT_ERROR);
    printf("CON_ERROR :  %u\n", CON_ERROR);
    exec("log off");
}
```

Listing M.3: Common function file (comfunc.ini)

```c
func void LO_CHECK(){
    switch(ERROR_POS){
        case 0x0001: MIT_ERROR(); break;
        case 0x0002: MIT_ERROR(); break;
        case 0x04FF: MIT_ERROR(); break;
        default: CON_ERROR(); break;
    }
}
```

Listing M.4: Abridged version of the MIT checking routines (mit.ini)

```c
func void HI_CHECK(){
    switch(ERROR_POS){
        case 0x0505: MIT_ERROR(); break;
        case 0x0509: MIT_ERROR(); break;
        case 0x07DB: MIT_ERROR(); break;
        default: CON_ERROR(); break;
    }
}
```

```c
func void END_CHECK(){
    switch(ERROR_POS){
        case 0xFF7: MIT_ERROR(); break;
        case 0xFF9: MIT_ERROR(); break;
        case 0xFFF: MIT_ERROR(); break;
        default: CON_ERROR(); break;
    }
}
```

Listing M.4: Abridged version of the MIT checking routines (mit.ini)
Appendix N  MIT simulation scripts

The scripts of the simulations carried out in Chapter 7 are shown in this appendix. Only the script for Prog_A is shown. The script for other simulations is identical apart from a few parameter changes and the MIT_CHECK() function. This function has been shortened to fit this thesis. The complete scripts for all test programs can be found on the CD-ROM (please refer to Appendix R).

```
Const END_CODE = &H1FFF 'Simulation constants
Const START_NF = &H761
Const END_NF = &H1FFF
Const MAX_CYCLES = 1000000
Const CON_CYCLES = 50000
Const NF_PRESENT = FALSE

Const HEX_FILENAME = "prog_a.hex"
Const LOG_FILENAME = "EIP_Error_Prog_A.txt"
Const INIT_FILENAME = "script_prog_a.ini"

Const eiLMIT = 0 'Enumerations (do not edit)
Const EMIT = 1
Const LIMIT = 2
Const NOR = 3
Const UNK1 = 4
Const EMP = 10
Const NF = 11
Const FT = 12
Const UE = 20
Const UNK2 = 21

Const S = 1
Const DN = 2
Const D2 = 3
Const TN = 4
Const T3 = 5
Const T3 = 6

Dim eiLMIT_ERR, EMIT_ERR, LIMIT_ERR, NOR_ERR 'Global variables
Dim EMP_ERR, UNK1_ERR, NF_ERR, FT_ERR, UE_ERR, UNK2_ERR
Dim ERROR_COUNT, ERROR_POS, ERROR_CYCLE, ERROR_SUB_CYCLE
Dim LAST_GOOD_PC, LAST_GOOD_SUB_CYCLE
Dim NUM_TEST, PC_temp, sim_start_time, sim_start_date
Dim log_file_num, init_file_num, Lbound_to, Hbound_to, Lbound_ld, Hbound_ld

Public Sub Main()
    Dim loop1, loop2
    SF.ResetSimulation 'reset simulator
    SV.SIM_ERROR_PROMPT = False 'disable error prompting
    SV.PC_ANTIALIAS_ENABLE = True 'enable PC aliasing
    If SF.LoadHexFile(HEX_FILENAME) = 1 Then 'load hex file
        NUM_TEST = CInt(SF.GetUserlnput("Number of Cycles: ", "Run time"))
        SF.Echo "IP Error Simulator" 'welcome message
        SF.Echo "NUM_TEST = " & CStr(NUM_TEST)
        SF.Echo ""
        SF.Echo "Simulation start..."
        SV.XTAL = 12000000 'clock frequency
        Randomize 'randomize seed

        If ERROR_COUNT=0 Then
            log_file_num=SF.CreateFile(LOG_FILENAME) 'first run, create log
        Else
            log_file_num=SF.OpenFile(LOG_FILENAME, 1) 'not first run, append log
        End If
```

N-1
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vbnet
sim_start_time = Time 'start time and date
sim_start_date = Date

For loop1 = 0 To NUM_TEST - 1 'body of simulation
ERROR_COUNT = ERROR_COUNT + 1 'increment error counter
SF.PutRAMByte &H30, 0 'Reset FT detected flag
GEN_ERROR() 'generates ECY, ESC, EIF
SF.AddBreakpoint 0, 0, 1, CLng(ERROR_CYCLE) 'add breakpoint
SF.RunSimulation SV.cFULLRUN 'run to breakpoint
SF.RemoveAllBreakpoint 'remove breakpoint

If SV.SCRIPT_STATUS = SV.cSCRIPTSTOP Then 'if sim. Stopped before BP.
    SF.Echo("Script stopped")
    ERROR_COUNT = ERROR_COUNT - 1 'reduce error count
    ERROR_SUMMARY() 'print sumary
    SF.CloseFile Cint(log_file_num) 'close log file
    SF.ResetSimulation 'reset simulator
    Exit Sub
Else
    Do Until (SV.CYCLE_STATE = ERROR_SUB_CYCLE) 'microsteps
        SF.MicroStepSimulation
    Loop
    LAST_GOOD_PC = SV.PC 'last "good" position
    LAST_GOOD_SUB_CYCLE = SV.CYCLE_STATE 'last state

    If SV.MULTIBYTE = true Then 'Checking for EMIT
        EMIT_ERROR()
    Else
        NO_EMIT_ERROR()
    End If
End If

EDC() 'Evaluate NF/FT
End If
SF.WriteFile Cint(log_file_num), "***" SF.WriteFile Cint(log_file_num), " " SF.ResetSimulation
Next

ERROR_SUMMARY() 'print summary
SF.CloseFile Cint(log_file_num) 'close log file
SF.ResetSimulation 'reset simulator
End If
SF.Echo("HEX file error. Script Terminating")
End If
End Sub

Private Sub EMIT_ERROR()
    Lbound_to = Cint(SF.InsLBoundary(CLng(SV.PC)))
    Hbound_to = Cint(SF.InsHBoundary(CLng(SV.PC)))
    Lbound_ld = Cint(SF.InsLBoundary(CLng(ERROR_POS))
    Hbound_ld = Cint(SF.InsHBoundary(CLng(ERROR_POS))

    If Lbound_to = 1 Then 'checking take-off
        Select Case Cint(Hbound_to)
            Case 1: If DEMIT_ERROR_CHECK() = 1 Then 'DEMIT
                eilMIT_ERR = eilMIT_ERR + 1
                PRINT_ERROR(eilMIT)
            Else
                EMIT_ERR = EMIT_ERR + 1
                PRINT_ERROR(EMIT)
            End If
            Case 2: If TEMIT1_ERROR_CHECK() = 1 Then 'TEMIT1
                eilMIT_ERR = eilMIT_ERR + 1
                PRINT_ERROR(eilMIT)
            Else
                EMIT_ERR = EMIT_ERR + 1
                PRINT_ERROR(EMIT)
            End If
        Case Else:
            SF.EchoAll Cint(log_file_num), _
            " !!! ERROR: EMIT_ERROR case not found. Lbound_to=' & _
            Cstr(Lbound_to) & ", Hbound_to=' & Cstr(Hbound_to)
            UNK1_ERR = UNK1_ERR + 1
            PRINT_ERROR(UNK1) 'instruction size > 3
        End Select
    End If
End Sub
```

N-2
Techniques intended to reduce the impact of program-flow errors on embedded systems

Else
  If Hbound_to = 1 Then
    DIMIT_ERROR_CHECK() = 1 Then
      DEMIT = 'take off pos. 2
      eILIMIT_ERR = eILIMIT_ERR + 1
      PRINT_ERROR(eILIMIT)
    Else
      EMIT_ERR = EMIT_ERR + 1
      PRINT_ERROR(EMIT)
    End If
  Else
    SF.EchoAll Cint(log_file_num), _
      " !!! ERROR: EMIT_ERROR case not found. Lbound_to=" & _
      Cstr(Lbound_to) & ", Hbound_to=" & Cstr(Hbound_to)
    UNK1_ERR = UNK1_ERR + 1
    PRINT_ERROR(UNK1) 'instruction size
  End If
End If
End Sub

Private Function DEMIT_ERROR_CHECK()
Select Case (SF.InsSize_A(CLng(ERROR_POS - Lbound_ld))) 'size of landing
Case 1: DEMIT_ERROR_CHECK = 0 'landing single byte
Case 2: If Lbound_ld = 0 Then
  If SF.InsType(CLng(ERROR_POS)) = DN Then 'checking instruction type
    DEMIT_ERROR_CHECK = 1 'Type DN, eILIMIT
  Else
    DEMIT_ERROR_CHECK = 0 'Type D2, NO eILIMIT
  End If
Else
    DEMIT_ERROR_CHECK = 0 'land at position 1
End If
Case 3: Select Case Lbound_ld
Case 0: If SF.InsType(CLng(ERROR_POS)) <> T23 Then 'checking ins.
  DEMIT_ERROR_CHECK = 1 'Type TN or T3, eILIMIT
Else
  DEMIT_ERROR_CHECK = 0 'Type T23, NO eILIMIT
End If
Case 1: 'landing position 1
If SF.InsType(CLng(ERROR_POS - 1)) = TN Then 'checking ins.
  DEMIT_ERROR_CHECK = 1 'Type TN, eILIMIT
Else
  DEMIT_ERROR_CHECK = 0 'Type T23 or T3, NO eILIMIT
End If
Case 2: 'landing position 2
End Select
End Select
End Function

Private Function TEMIT1_ERROR_CHECK()
Dim this_ins, next_ins
this_ins = CLng(ERROR_POS - Lbound_ld)
ext_ins = CLng(ERROR_POS + Hbound_ld)
Select Case (SF.InsSize_A(CLng(this_ins))) 'size of landing ins.
Case 1: 'single-byte instruction
  Select Case (SF.InsType(CLng(next_ins))) 'check landing+1 ins.
    Case S, D2, T23:
      TEMIT1_ERROR_CHECK = 0 'Type S, D2, T23
    Case Else:
      TEMIT1_ERROR_CHECK = 1 'NO eILIMIT
  End Select
Case 2: 'double-byte instruction
  If Lbound_ld = 0 Then 'checking landing position
    TEMIT1_ERROR_CHECK = 0 'landing 0, NO eILIMIT
Else
  Select Case (SF.InsType(CLng(next_ins))) 'check landing+1 ins.
    Case S, D2, T23:
      TEMIT1_ERROR_CHECK = 0 'Type S, D2, T23
    Case Else:
      TEMIT1_ERROR_CHECK = 1 'NO eILIMIT
  End Select
End Select
Case 3: 'triple-byte instruction
  Select Case Lbound_ld 'checking landing position
    Case 0: 'landing position 0
End Select
End Function
Techniques intended to reduce the impact of program-flow errors on embedded systems

If SF.InsType(CLng(this_ins)) <= 4 Then 'check landing ins.
   TEMIT1_ERROR_CHECK = 1 'S, DN, D2 or TN, eilMIT
Else
   TEMIT1_ERROR_CHECK = 0 'T23 or T3, NO eilMIT
End If

Case 1:
   TEMIT1_ERROR_CHECK = 0 'landing position 1
Case 2:
   TEMIT1_ERROR_CHECK = 0 'landing position 2
Select Case (SF.InsType(CLng(next_ins))
   Case S, D2, T23:
      TEMIT1_ERROR_CHECK = 0 'Type S, D2, T23
   Case Else:
      TEMIT1_ERROR_CHECK = 1 'eilMIT
End Select
End Select
End Function

Private Sub NO_EMIT_ERROR()
   If M1T_CHECK(CLng(ERROR_POS)) = 1 Then 'Checking for LMIT
      LMIT_ERR = LMIT_ERR + 1
      PRINT_ERROR(LMIT) 'LMIT_ERR detected
   Else
      NOR_ERR = NOR_ERR + 1
      PRINT_ERROR(NOR)
   End If
End Sub

Private Sub EDC()
   Dim count, no_err 'NF/FT detected error?
   SF.EchoAll Cint(log_file_num), "10 cycle RUN"
   For count = 1 To 10 'additional 10 cycles
      SF.StepSimulation 'step simulation
      SF.EchoAll Cint(log_file_num), "CY=" & Cstr(SV.CYCLES) & 
      " , PC=" & hex(SV.PC)
   Next
   SF.EchoAll Cint(log_file_num), "Trying to determine NF, FT or UE-type error"
   If ((ERROR_POS >= START_NF) And (ERROR_POS <= END_NF)) Then 'Check NF/EMPTY
      If NF_PRESENT = True Then
         NF_ERR = NF_ERR + 1
         PRINT_ERROR(NF) 'NF_ERR detected
      Else
         EMP_ERR = EMP_ERR + 1
         PRINT_ERROR(EMP) 'EMP_ERR detected
      End If
   Else
      SV.PC = ERROR_POS
      no_err = 1 'making IP=error value
      For count = 0 To CON_CYCLES - 1
         SF.StepSimulation 'cycle stepping
         If SF.GetRAMByte(&H30) = &H33 Then 'testing FT flag
            FT_ERR = FT_ERR + 1
            PRINT_ERROR(FT) 'FT_ERR detected
            no_err = 0 'exit
         End If
      Next
      If SV.SIM_ERROR <> SV.eNOERROR Then 'testing simulator error
         SF.EchoAll Cint(log_file_num), "SIMULATOR ERROR " & 
         Cstr(SV.SIM_ERROR)
         UNK2_ERR = UNK2_ERR + 1
         PRINT_ERROR(UNK2) 'UNK2_ERR detected
         no_err = 0 'exit
      End If
   End If
   If SV.CYCLES > (MAX_CYCLES + CON_CYCLES) Then 'testing simulator error
      SF.EchoAll Cint(log_file_num), "RUN ERROR - running above maximum cycle limit"
      UNK2_ERR = UNK2_ERR + 1
      PRINT_ERROR(UNK2) 'UNKOWN_ERR detected

N-4
Techniques intended to reduce the impact of program-flow errors on embedded systems

count = CON_CYCLES + 10
no_err = 0
End If

If no_err = 0 Then Exit For

Next

If no_err = 1 Then
UE_ERR = UE_ERR + 1
PRINT_ERROR(UE)
End If
End If

Private Sub GEN_ERROR()
ERROR_CYCLE = Clng(Rnd() * MAX_CYCLES)
ERROR_POS = Clng(Rnd() * END_CODE)
ERROR_SUB_CYCLE = Cint(Rnd() * 12)
If ERROR_SUB_CYCLE = 0 Then ERROR_SUB_CYCLE = 1
End Sub

Private Sub PRINT_ERROR(error_type)
Dim STR
Select Case Cint(error_type)
  Case eiLMIT: STR = "ERROR # " & Cstr(ERROR_COUNT) & " eiLMIT"
  Case EMIIT: STR = "ERROR # " & Cstr(ERROR_COUNT) & " EMIIT"
  Case LMIT: STR = "ERROR # " & Cstr(ERROR_COUNT) & " LMIT"
  Case NOR: STR = "ERROR # " & Cstr(ERROR_COUNT) & " NOR"
  Case UNK1: STR = "ERROR # " & Cstr(ERROR_COUNT) & " UNK1"
  Case EMP: STR = "ERROR # " & Cstr(ERROR_COUNT) & " EMP"
  Case NF: STR = "ERROR # " & Cstr(ERROR_COUNT) & " NF"
  Case FT: STR = "ERROR # " & Cstr(ERROR_COUNT) & " FT"
  Case UE: STR = "ERROR # " & Cstr(ERROR_COUNT) & " UE"
  Case UNK2: STR = "ERROR # " & Cstr(ERROR_COUNT) & " UNK2"
End Select
STR = STR & " at PC=" & hex(SV.PC) & " CY=" & Cstr(SV.CYCLES) & " CYCLES_STATE"
STR = STR & " ( " & Cstr(NOR_ERR) & " LMIT_ERR) & & Cstr(eiLMIT_ERR) & & Cstr(EMIT_ERR) & & Cstr(UNK1_ERR) & & Cstr(SV.Date) & & Cstr(SV.Time) & & Cstr(SV.Sim_start_time) & & Cstr(systime)
SF.EchoAll Cint(log_file_num), Cstr(STR)
End Sub

Private Sub ERROR_SUMMARY()
SF.EchoAll Cint(log_file_num), " Simulation start time: " & Cstr(systime)
SF.EchoAll Cint(log_file_num), " Simulation end time: " & Cstr(SV.Time) & & Cstr(SV.Sim_end_time) & & Cstr(SV.Sim_end_date)
End Sub

Private Function MIT_CHECK(value)
Select Case Clng(value)
  Case &H0001: MIT_CHECK = 1
  Case &H075F: MIT_CHECK = 1
  Case Else: MIT_CHECK = 0
End Select
End Function

Listing N.1: Abridged version of the MIT simulation script for Prog_A
Appendix O  ST, ST2 and BCT’s simulation script

The simulation scripts for each technique running with Prog_A are presented in this appendix. The scripts for other variants (for each technique) are identical apart from a few parameters. Note: the CHECK_COR() function has been shortened.

The complete scripts for all test programs can be found on the CD-ROM (please refer to Appendix R).

0.1 Simulation script for ST

```vba
Const CODE_SIZE = &H1FFF 'Simulation parameters
Const HEX_FILENAME = "Prog_A.hex" 'program file name
Const LOG_FILENAME = "Prog_A" 'log file name
Const MAX_CYCLES = 1000

Dim ECY, ESC, EQC, EPC 'Global variables
Dim CYCLE, SUBCYCLE, QUANCYCLE 'internal counters
Dim ERROR_COUNT, NUM_TEST 'errors injected
Dim sim_start_time, sim_start_date 'injection results
Dim log_file_num 'start date and time

Dim SPC, CREG, STORE, CHECK 'ST1 variables and signals
Dim S_PC, S_PC_PRE 'S-PC (+ edge detection)
Dim PC_S, PC_S_PRE 'PC-S (+ edge detection)
Dim CMP_X 'CMP output (CMP)
Dim ERROR 'error flag

Public Sub Main()
    Dim filename, FIRST_CY, SIM_ERR
    SF.ResetSimulation() 'reset simulator
    SV.SIM_ERROR_PROMPT = False 'disable prompts on errors
    SV.PC_ANTIALIAS_ENABLE = True 'enable ROM antialiasing
    SV.XTAL = 12000000 'clock frequency

    If SF.LoadHexFile(HEX_FILENAME) = 1 Then 'load hex file
        ERROR_COUNT = CInt(SF.GetUserInput("Number of Cycles: ", "Run time"))
        filename = LOG_FILENAME & "  ( ERR=" & CStr(ERROR_COUNT) & ").txt"
        log_file_num = SF.CreateFile(Cstr(filename)) 'create log file
        SF.Echo "State Tracking 1 simulation" 'welcome message
        SF.Echo "Simulation start..."
        Randomize 'randomize RNG seed
        sim_start_time = Time 'start time and date
        sim_start_date = Date

        For NUM_TEST = 1 To ERROR_COUNT 'main loop
            GEN_ERROR() 'generates ECY, ESC and EPC
            SF.EchoAll Cint(log_file_num), "NUMBER OF ERRORS=" & Cstr(ERROR_COUNT)
            SF.EchoAll Cint(log_file_num), "ERROR #=" & Cstr(NUM_TEST) & " : Error cycle=" & Cstr(ECY) & ", ESC=" & Cstr(ESC) & ", EQC=" & Cstr(EQC) & ", EPC=" & hex(EPC)
            SF.ResetSimulation 'reset MCU
            ST1_RESET() 'reset ST1
            FIRST_CY = True 'reset 1st cycle flag
            SIM_ERR = False
```

O-1
Techniques intended to reduce the impact of program-flow errors on embedded systems

For CYCLE = 0 To ECY + 3  '2 additional clock-cycles
For SUBCYCLE = 1 To 12
For QUANCYCLE = 0 To 3
If QUANCYCLE = 0 Then 'single step on QCY 0
If SV.SIM_ERROR <> SV.eNOERROR Then 'simulator is OK?
QUANCYCLE = 4  'If not, exit
SUBCYCLE = 1
CYCLE = ECY + 4
SIM_ERR = True
Else
SF.MicroStepSimulation() 'If OK, simulate
End If
End If
If FIRST_CY = True Then 'to align counter with MCU
SUBCYCLE = 2
QUANCYCLE = QUANCYCLE - 1
FIRST_CY = False
End If
INJECT_ERROR()
ST1_UPDATE()
Next
Next
Next
If SIM_ERR = True Then 'determine If simulator
ERR_UNK = ERR_UNK + 1
SF.EchoAll Cint(log_file_num), "+++ SIMULATOR ERROR " & _
Cstr(SV.SIM_ERROR) & " & Cstr(ERR_DET) & ":" & _
Cstr(ERR_UNDET) & ":" & Cstr(ERR_UNK) & ":" & Cstr(ERR_COR) & _
& Cstr(ERR_UNCOR)
PRINT_SIM_INFO()
Else 'no errors encountered
If ERROR <> 1 Then 'check If ST detected error
ERR_UNDET = ERR_UNDET + 1 'increment UNDET counter
SF.EchoAll Cint(log_file_num), "$$$ PC ERROR NOT CORRECTED " & _
Cstr(ERR_DET) & ":" & Cstr(ERR_UNDET) & ":" & Cstr(ERR_UNK) & ":" & Cstr(ERR_COR) & _
& Cstr(ERR_UNCOR)
PRINT_SIM_INFO()
End If
End If
If CHECK_COR_ERR() = 1 Then 'see If error is corrected
ERR_COR = ERR_COR + 1
SF.EchoAll Cint(log_file_num), "&&& CORRECTED ERROR " & _
Cstr(ERR_DET) & ":" & Cstr(ERR_UNDET) & ":" & Cstr(ERR_UNK) & ":" & Cstr(ERR_COR) & _
& Cstr(ERR_UNCOR)
Else
SF.EchoAll Cint(log_file_num), "%%% UNCORRECTED ERROR " & _
Cstr(ERR_DET) & ":" & Cstr(ERR_UNDET) & ":" & Cstr(ERR_UNK) & _
& Cstr(ERR_COR) & ":" & Cstr(ERR_UNCOR)
End If
SF.EchoAll Cint(log_file_num), "" 'test loop
Next
SF.EchoAll Cint(log_file_num), "Detectable errors (DET) = " & _
Cstr(ERR_DET)
SF.EchoAll Cint(log_file_num), "Undetectable errors (UNDET) = " & _
Cstr(ERR_UNDET)
SF.EchoAll Cint(log_file_num), "SIMULATOR errors (UNK) = " & _
Cstr(ERR_UNK)
SF.EchoAll Cint(log_file_num), "Corrected errors (COR) = " & _
Cstr(ERR_COR)
SF.EchoAll Cint(log_file_num), "Uncorrected errors (UNCOR) = " & _
Cstr(ERR_UNCOR)
SF.EchoAll Cint(log_file_num), "Start date:" & Cstr(sim_start_date) & _
"Start time:" & Cstr(sim_start_time)
SF.CloseFile Cint(log_file_num) 'close log file
SF.ResetSimulation() 'reset simulator
Techniques intended to reduce the impact of program-flow errors on embedded systems

Else
    SF.Echo("HEX file error. Script Terminating")
End If

SF.Echo("End of Script")

Private Sub PRINT_SIM_INFO()
    SF.EchoAll Cint(log_file_num), "CY=" & Cstr(SV.CYCLES) & ";" & _
               Cstr(SV.CYCLES_STATE) & ";" & Cstr(QUANCYCLE) & ";PC=" & hex(SV.PC)
End Sub

Private Sub GEN_ERROR()
    ECY = CLng(Rnd() * MAX_CYCLES)
    ESC = CInt(Rnd() * 12)
    If ESC = 0 Then ESC = 1
    EQC = CInt(Rnd() * 4)
    If EQC = 0 Then EQC = 1
    EPC = CLng(Rnd() * CODE_SIZE)
End Sub

Private Sub INJECT_ERROR() 'when cycle is correct
    If CYCLE = ECY Then
        If SUBCYCLE = ESC Then
            If QUANCYCLE = EQC Then
                SF.EchoAll Cint(log_file_num), "*** Error reached. Vectoring PC=" & hex(SV.PC) & " to PC=" & hex(EPC)
                SV.PC = EPC
            End If
        End If
    End If
End Sub

***** STATE TRACKING MODULES *****

Private Sub ST1_RESET()
    SPC = 0
    CREG = SV.CYCLE_STATE + 1
    STORE = 0
    CHECK = 0
    S_PC = 0
    PC_S = 0
    ERROR = 0
End Sub

Private Sub ST1_UPDATE()
    UPDATE_CREG()
    UPDATE_CON()
    UPDATE_CMP()
    UPDATE_PC_N_SPC()
End Sub

Private Sub UPDATE_CREG()
    If QUANCYCLE = 0 Then
        If CREG = 12 Then
            CREG = 1
        Else
            CREG = CREG + 1
        End If
    End If

    Select Case CREG
        Case 2: CHECK = 1
            STORE = 0
        Case 4: CHECK = 0
            'PCINC cycle
            STORE = 1
        Case 5: STORE = 0
        Case 8: CHECK = 1
            'check cycle
            STORE = 1
        Case 10: CHECK = 0
            'PCINC cycle
            STORE = 1
        Case 11: STORE = 0
        Case 12: CHECK = 1
            'PCINC cycle
            STORE = 1
        Case 1: CHECK = 0
    End Select
End If

Private Sub UPDATE_CON()
Techniques intended to reduce the impact of program-flow errors on embedded systems

If CHECK = 1 Then
  'error check cycle
  If QUANCYCLE = 3 Then
    If CMP_X = 0 Then
      S_PC = 1
    End If
  End If
End If

If STORE = 1 Then
  If QUANCYCLE = 1 Then
    PC_S = 1
  Else
    PC_S = 0
  End If
End If

If QUANCYCLE = 0 Then
  S_PC = 0
End If

Private Sub UPDATE_CMP
  If QUANCYCLE = 2 Then
    If SV.PC = SPC Then
      CMP_X = 1
    Else
      CMP_X = 0
    End If
  End If
End Sub

Private Sub UPDATE_PC_N_SPC
  If (PC_S = 1) And (PC_S_PRE = 0) Then
    SPC = SV.PC
  End If

  If (S_PC = 1) And (S_PC_PRE = 0) Then
    SV.PC = SPC
    ERROR = 1
    ERR_DET = ERR_DET + 1
    SF.WriteFile Cint(log_file_num), "!!! PC ERROR CORRECTED " & _
    Cstr(ERR_DET) & " & " & Cstr(ERR_UNDET) & " & " & Cstr(ERR_UNK) & _
    Cstr(ERR_COR) & " & " & Cstr(ERR_UNCOR)
    PRINT_SIM_INFO()
  End If

  PC_S_PRE = PC_S
  S_PC_PRE = S_PC
End Sub

***** CHECKING PROFILE *****

Private Function CHECK_COR_ERR()
  Dim temp, CY, CHECK_OK
  For temp = 0 To (12 - SV.CYCLE_STATE)
    SF.MicroStepSimulation
  Next

  SF.WriteFile Cint(log_file_num), "{{ Executing 10 CY"
  CHECK_OK = true
  For temp = 1 To 10
    SF.StepSimulation
    PRINT_SIM_INFO
    CY = Cstr(SV.CYCLES) & & Cstr(SV.CYCLE_STATE)
    If Clng(CHECK_COR(CY)) <> Clng(SV.PC) Then
      CHECK_OK = false
      temp = 11
    End If
  Next

  If CHECK_OK = true Then
    CHECK_COR_ERR = 1
  Else
    CHECK_COR_ERR = 0
End If
Techniques intended to reduce the impact of program-flow errors on embedded systems

End Function

Private Function CHECK_COR(CY)
    Select Case CLng(CY)
    Case 01: CHECK_COR = 0
    Case 11: CHECK_COR = 2
    Case 9991: CHECK_COR = 1174
    Case 10001: CHECK_COR = 1174
    End Select
End Function

Listing 0.1: Abridged version of ST's simulation script for Prog_A

0.2 Simulation script for ST2

Const CODE_SIZE = &H1FFF  'Simulation parameters
Const HEX_FILENAME = "Prog_A.hex"  'program file name
Const LOG_FILENAME = "Prog_A"  'log file name
Const MAX_CYCLES = 1000

Dim ECY, ESC, EQC, EPC  'Global variables
Dim CYCLE, SUBCYCLE, QUANCYCLE  'internal cycle counters
Dim ERROR_COUNT, NUM_TEST  'number of errors
Dim NUM_TEST
Dim ERR_DET, ERR_DET, ERR_UNDET, ERR_COR, ERR_UNK  'error injection results
Dim sim_start_time, sim_start_date  'start date and time
Dim log_file_num  'log file number
Dim SPC, CREG, STORE, CHECK, CHECKSTORE  'ST1 variables and signals
Dim S_PC, S_PC_PRE  'S-PC (+ edge detection)
Dim PC_S, PC_S_PRE  'PC-S (+ edge detection)
Dim CMP_X0, CMP_X1  'CMP output X0, X1
Dim ERROR  'error flag
Dim DETNOTCOR  'flag to denote DET error

Public Sub Main()
    Dim filename, FIRST_CY, SIM_ERR
    SF.ResetSimulation()  'reset simulator
    SV.SIM_ERROR_PROMPT = False  'disable error prompting
    SV.PC_ANTIALIAS_ENABLE = True  'enable antialiased ROM
    SV.XTAL = 12000000  'clock frequency
    If SF.LoadHexFile(HEX_FILENAME) = 1 Then  'load hex file
        ER0R_COUNT = Cint(SF.GetUserlnput("Number of Cycles: ", "Run time"))
        filename = LOG_FILENAME & "  ( ERR= " & Cstr(ERROR_COUNT) & ").txt"
        log_file_num = SF.CreateFile(Cstr(filename))  'create log file
        SF.Echo "State Tracking 1 simulation"  'welcome message
        SF.Echo "Welcome to State Tracking 1 simulation..."
        Randomize  'randomize RNG seed
        sim_start_time = Time  'start time and date
        sim_start_date = Date
        For NUM_TEST = 1 To ERROR_COUNT  'main loop
            GEN_ERROR()  'generates error cycle
            SF.EchoAll Cint(log_file_num), "ERROR #" & Cstr(NUM_TEST) & _
            " : Error cycle=" & Cstr(ECY) & ", & Cstr(ESC) & ", & Cstr(EPC) & ", & _
            Cstr(EPC) & ", & & hex(EPC)
            SF.ResetSimulation  'reset MCU
            ST2_RESET()  'reset ST1
            FIRST_CY = True  'reset 1st cycle
            SIM_ERR = False
        Next
    End If
End Sub
Techniques intended to reduce the impact of program-flow errors on embedded systems

For CYCLE = 0 To ECY + 3
For SUBCYCLE = 1 To 12
For QUANCYCLE = 0 To 3
If QUANCYCLE = 0 Then 'single step simulation
If SV.SIM_ERROR <> SV.eNOERROR Then 'simulator OK?
QUANCYCLE = 4 'if not, exit
SUBCYCLE = 13
CYCLE = ECY + 4
SIM_ERR = True
Else
SF.MicroStepSimulation() 'if OK, simulate
End If
End If

If FIRST_CY = True Then 'to align counter with MCU
SUBCYCLE = 2
QUANCYCLE = QUANCYCLE - 1
FIRST_CY = False
End If

INJECT_ERROR() 'inject error
ST2_UPDATE() 'execute ST1
Next 'quantum loop
Next 'subcycle loop
Next 'cycle loop
If SIM_ERR = True Then 'simulator OK?
ERRJ3NK = ERRJ3NK + 1
SF.EchoAll Cint(log_file_num), " +++ SIMULATOR ERROR ( " & _
Cstr(SV.SIM_ERROR) & " ) " & Cstr(ERR_DET) & "," & _
Cstr(ERR_DETCOR) & "," & Cstr(ERR_DETUNK) & _
",:" & Cstr(ERR_COR)
Else 'simulator OK
If ERROR = 0 Then ' ST2 detected error?
ERR_DET = ERR_DET + 1
SF.EchoAll Cint(log_file_num), " $$$ PC ERROR NOT CORRECTED " & _
Cstr(ERR_DET) & "," & Cstr(ERR_DETCOR) & "," & _
Cstr(ERR_DETUNK) & ",:" & Cstr(ERR_COR)
Else
ERR_DETCOR = ERR_DETCOR + 1
SF.EchoAll Cint(log_file_num), " !!! PC ERROR CORRECTED " & _
& Cstr(ERR_DET) & "," & Cstr(ERR_DETCOR) & "," & _
Cstr(ERR_DETUNK) & ",:" & Cstr(ERR_COR)
End If
End If
End If
End If

PRINT SIM_INFO()

If CHECK_COR_ERR() = 1 Then 'checks if error corrected
ERR_COR = ERR_COR + 1
SF.EchoAll Cint(log_file_num), " &&& CORRECTED ERROR " & _
Cstr(ERR_DET) & "," & Cstr(ERR_DETCOR) & "," & _
Cstr(ERR_DETUNK) & ",:" & Cstr(ERR_COR)
Else
SF.EchoAll Cint(log_file_num), " $$$ UNCORRECTED ERROR " & _
& Cstr(ERR_DET) & "," & Cstr(ERR_DETCOR) & "," & _
Cstr(ERR_DETUNK) & ",:" & Cstr(ERR_COR)
End If

SF.EchoAll Cint(log_file_num), " test loop
SF.EchoAll Cint(log_file_num), "Detectable And Correctable errors _
(DETCOR) = " & Cstr(ERR_DETCOR)
SF.EchoAll Cint(log_file_num), "Detectable errors _
(DET) = " & Cstr(ERR_DET)
SF.EchoAll Cint(log_file_num), "Undetectable errors _
(UNDET) = " & Cstr(ERR_DETUNK)
SF.EchoAll Cint(log_file_num), "SIMULATOR errors _
Techniques intended to reduce the impact of program-flow errors on embedded systems

(UNK) = " & CSB(UNK)
SF.EchoAll Cint(log_file_num), "
SF.EchoAll Cint(log_file_num), 'Corrected errors _
(COR) = " & CSB(ERR_COR)
SF.EchoAll Cint(log_file_num), 'Uncorrected errors _
(UNCOR) = " & CSB(ERROR_COUNT - ERR_COR)
SF.EchoAll Cint(log_file_num), "Start date:" & CSB(sim_start_date) & " _
Start time:" & CSB(sim_start_time)
SF.EchoAll Cint(log_file_num), "End date:" & Date() & "  End time:" & time()
SF.CloseFile Cint(log_file_num) 'close log file
SF.ResetSimulation() 'reset simulator
Else
SF.Echo('HEX file error. Script Terminating')
End If

Private Sub PRINT_SIM_INFO()
SF.EchoAll Cint(log_file_num), "CY=" & CSB(SV.CYCLES) & "." & _
CSB(SV.CYCLE_STATE) & ":PC=1 & hex(SV.PC)
End Sub

Private Sub GEN_ERROR()
ECY = CLng(Rnd()) * MAX_CYCLES 'generate ECY
Do
ESC = Cint(Rnd()) * 12 'generate ESC
Loop While ESC = 0
Do
EQC = Cint(Rnd()) * 4 'generate EQC
Loop While EQC = 4
EPC = CLng(Rnd()) * CODE_SIZE 'generate EPC
End Sub

Private Sub INJECT_ERROR() 'inject errors
If CYCLE = ECY Then
If SUBCYCLE = ESC Then
If QUANCYCLE = EQC Then
SF.EchoAll Cint(log_file_num), "  *** Error reached. Vectoring PC=" & hex(SV.PC) & "  to PC=" & hex(EPC)
PRINT_SIM_INFO()
SV.PC = EPC
End If
End If
End If
End Sub

****** STATE TRACKING 2 MODULES *****
Private Sub ST2_RESET()
SPC = 0
CREG = SV.CYCLE_STATE + 1
STORE = 0
CHECK = 0
CHECKSTORE = 0
S_PC = 0
S_PC_PRE = 0
PC_S = 0
PC_S_PRE = 0
ERROR = 0
DETNOTCOR = 0 'for simulator used only
End Sub

Private Sub ST2_UPDATE()
UPDATE_CREG()
UPDATE_CMP()
UPDATE_CON()
UPDATE_PC_N_SPC()
End Sub

Private Sub UPDATE_CREG()
If QUANCYCLE = 0 Then 'update only on SYNC
If CREG = 12 Then 'if CREG = 12 then reset
CREG = 1
End If
End Sub
Else
  CREG = CREG + 1  'else increment CREG
End If

Select Case CREG
  Case 2: CHECK = 1  'check cycle
    STORE = 0
  Case 3:
    CHECK = 0
  Case 4: CHECKSTORE = 1  'PCINC cycle
    STORE = 0
  Case 5: CHECKSTORE = 0
  Case 6: CHECK = 1
  Case 7: CHECK = 0
  Case 10: CHECKSTORE = 1  'PCINC cycle
  Case 11: CHECKSTORE = 0
  Case 12: CHECK = 1
  Case 1: CHECK = 0
    STORE = 1  'PCCHG cycle
End Select
End If
End Sub

Private Sub UPDATE_CMP
  Dim PC_temp
  PC_temp = SV.PC
  If PC_temp = SPC Then  'combinational logic
    CMP_X0 = 1  
    If PC_temp = SPC + 1 Or (SPC = PC_temp + 1) Then
      CMP_X1 = 1  'if equal then CMP_X0 = 1
      Else
        CMP_X0 = 0  'else CMP_X0 = 0
      End If
    If Not ((CMP_X0 = 1) And (CMP_X1 = 0)) Then
      ERROR = 1  
      If QUANCYCLE = 3 Then
        If CMP_X0 = CMP_X1 Then
          DETNOTCOR = 1  'something's wrong here!
        Else
          PC_S = 1  'ST2 cannot correct this
        End If
      End If
      Else
        PC_S = 0  'error check cycle
      End If
    End If
  End If
End Sub

Private Sub UPDATE_CON()
  If CHECK = 1 Then  'error check cycle
    If QUANCYCLE = 1 Then
      If Not ((CMP_X0 = 1) And (CMP_X1 = 0)) Then  'PC = SPC comparison fail
        ERROR = 1  
        S_PC = 1  'PC = SPC value
      End If
    End If
  End If
  If CHECKSTORE = 1 Then  'check for diff
    If QUANCYCLE = 1 Then
      If CMP_X0 = CMP_X1 Then
        DETNOTCOR = 1  'something's wrong here!
      Else
        PC_S = 1  'ST2 cannot correct this
      End If
    Else
      PC_S = 0  'error check cycle
    End If
  End If
  If STORE = 1 Then  'PCCHG or PCINC cycles
    If QUANCYCLE = 1 Then
      PC_S = 1  
    Else
      PC_S = 0  'enable SPC = PC value
    End If
  End If
  If QUANCYCLE = 0 Then  'PCCHG or PCINC cycles
    S_PC = 0  'disable for other cycles
  End If
End Sub

Private Sub UPDATE_PC_N_SPC
  If (PC_S = 1) And (PC_S_PRE = 0) Then  
    S_PC = 1  'reset S-PC
    PC_S = 0  'enable SPC
    End If
End Sub

Techniques intended to reduce the impact of program-flow errors on embedded systems
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vbnet
End If
If (S_PC = 1) And (S_PC_PRE = 0) Then
  S_PC = SPC
End If
' age signals
PC_S_PRE = PC_S 'age signals
S_PC_PRE = S_PC
End Sub

***** CHECKING PROFILE *****
Private Function CHECK_COR_ERR()
  Dim temp, CY, CHECK_OK
  For temp = 0 To (12 - SV.CYCLE_STATE) 'execute till cycle boundary
    SF.MicroStepSimulation
  Next
  SF.WriteFile Cint(log_file_num), " {{{ Executing 10 CY"
  CHECK_OK = true
  For temp = 1 To 10
    SF.StepSimulation
    PRINT_SIM_INFO
    CY = Cstr(SV.CYCLES) & Cstr(SV.CYCLE_STATE)
    If Clng(CHECK_COR(CY)) <> Clng(SV.PC) Then
      CHECK_OK = false
      temp = 11
    End If
  Next
  If CHECK_OK = true Then
    CHECK_COR_ERR = 1
  Else
    CHECK_COR_ERR = 0
  End If
End Function

Private Function CHECK_COR(CY)
  Select Case Clng(CY)
    Case 01: CHECK_COR = &h0
    Case 11: CHECK_COR = &h2
    Case 10191: CHECK_COR = &h49A
    Case 10201: CHECK_COR = &h49B
  End Select
End Function

Listing 0.2: Abridged version of ST2's simulation script for Prog_A

O.3 Simulation script for BCT

Const CODE_SIZE = &H1FFF 'Simulation parameters
Const HEX_FILENAME = "Prog_a.hex" 'program file name
Const LOG_FILENAME = "Prog_A" 'log file name
Const MAX_CYCLES = 1000

Dim ECY, ESC, EQC, EQC, EPC 'Global variables
Dim CYCLE, SUBCYCLE, QUANCYCLE 'internal cycle counters
Dim ERROR_COUNT, NUM_TEST 'number of errors injected
Dim NUM_TEST 'error injection results
Dim ERR_DET, ERR_UNDET, ERR_COR, ERR_UNK
Dim sim_start_time, sim_start_date 'start date and time
Dim log_file_num 'log file number
Dim SPC, BREG, CREG 'BCT variables and signals
Dim ROM_temp, INT_HANDLED, INSTYPE
Dim BREG_val, CREG_val, BREG_en
Dim INC_val, INC_valo

O-9
Techniques intended to reduce the impact ofprogram-flow errors on embedded systems

Dim
Dim
Dim
Dim
Dim

CHECK, SPCINC, CHECKSTORE, STORE
S_PC, S_PC_PRE
PC_S, PC_S_PRE
CMP
ERROR

'SPC to PC (CON)
'PC to SPC (CON)
'CMP output
'error flag

Public Sub Main()
Dim filename, FIRST_CY, SIM_ERR
SF.ResetSimulation()
SV.SIM_ERROR_PROMPT = False
SV.PC_ANTIALIAS_ENABLE = True
SV.XTAL = 12000000

'reset simulator
'disable error prompting
'enable ROM antialiasing
'clock frequency

If S F .LoadHexFile (HEX_F ILENAME) = 1 Then
'load hex file
ERROR_COUNT = Cint(S F .GetUserInput("Number of Cycles: ", "Run time"))
filename = LOG_FILENAME & " (ERR=" & Cstr(ERROR_COUNT) & ").txt"
log_file_num = SF.CreateFile(Cstr(filename))
'create log file
SF.Echo "State Tracking 1 simulation"
'welcome message
S F .Echo ""
SF.EchoAll Cint(log_file_num), "NUMBER OF ERRORS=" & CStr(ERROR_COUNT)
SF.EchoAll Cint(log_file_num), ""
SF.Echo "Simulation start..."
Randomize
sim_start_time = Time
sim_start date = Date

'randomize RNG seed
'start time and date

For NUM_TEST = 1 To ERROR_COUNT
'main loop
GEN_ERROR()
'generates error
SF.EchoAll Cint(log_file_num), "ERROR #" & Cstr(NUM_TEST) &
": Error cycle=" & Cstr(ECY) & "." & Cstr(ESC) & "." & _
Cstr(EQC) & ", EPC=" & hex(EPC)
SF.ResetSimulation
'reset MCU
BCT_RESET()
'reset BCT
FIRST_CY = True
'reset 1st cycle
SIM ERR = False
For CYCLE = 0 To ECY + 3
For SUBCYCLE = 1 To 12
For QUANCYCLE = 0 To 3
If QUANCYCLE = 0 Then
'single
If S V .SIM_ERROR <> SV.eNOERROR Then
QUANCYCLE = 4
'If not,
SUBCYCLE = 1 3
CYCLE = ECY + 4
SIM_ERR = True
Else
SF.MicroStepSimulation(
If OK,
End If
End If
If FIRST_CY = True Then
SUBCYCLE = 2
QUANCYCLE = QUANCYCLE
FIRST_CY = False
End If
INJECT_ERROR()
BCT_UPDATE()
Next
Next
Next

step simulation
'simulator OK?
exit

simulate

'to align counter with MCU

'inject error
'execute BCT
'quantum loop
'subcycle loop
'cycle loop

If SIM_ERR = True Then
'simulator OK?
ERR_UNK = ERR_UNK + 1
SF.EchoAll Cint(log_file_num),
+++ SIMULATOR ERROR (" &
Cstr(S V .SIM_ERROR)
’) " Sc Cstr (ERR_DETCOR) & "," &
Cstr(ERR_UNDET) &
& Cstr(ERR_UNK) &
& Cstr(ERR_COR)
Else
'simulator OK
If ERROR = 0 Then
'BCT detected error?
ERR_UNDET = ERR UNDET
1
'increment UNDET
SF.EchoAll Cint(log_file_num), " $$$ PC ERROR NOT DETECTABLE
& Cstr(ERR_DETCOR) & "," & Cstr(ERR_UNDET) &
& _
Cstr(ERR_UNK) &
& Cstr(ERR_COR)
Else

0 -1 0


Techniques intended to reduce the impact of program-flow errors on embedded systems

```vbnet
ERR_DETDCOR = ERR_DETDCOR + 1
SF.EchoAll Cint(log_file_num), " !!! PC ERROR CORRECTABLE " & _
Cstr(ERR_DETDCOR) & "," & Cstr(ERR_UNDET) & "," & _
Cstr(ERR_UNK) & ":" & Cstr(ERR_COR)

End If
End If
PRINT_SIM_INFO()

If CHECK_COR_ERR() = 1 Then 'checks if error corrected
ERR_COR = ERR_COR + 1
SF.EchoAll Cint(log_file_num), " & & CORRECTED ERROR " & _
Cstr(ERR_DETDCOR) & "," & Cstr(ERR_UNDET) & "," & _
Cstr(ERR_UNK) & ":" & Cstr(ERR_COR)
Else
SF.EchoAll Cint(log_file_num), " & & UNCORRECTED ERROR " & _
Cstr(ERR_DETDCOR) & "," & Cstr(ERR_UNDET) & "," & _
Cstr(ERR_UNK) & ":" & Cstr(ERR_COR)
End If

SF.EchoAll Cint(log_file_num), " "

Next 'test loop
SF.EchoAll Cint(log_file_num), ""
SF.EchoAll Cint(log_file_num), "Detectable And Correctable errors _
(DETDCOR) = " & Cstr(ERR_DETDCOR)
SF.EchoAll Cint(log_file_num), "Undetectable errors _
(UNDET) = " & Cstr(ERR_UNDET)
SF.EchoAll Cint(log_file_num), "SIMULATOR errors _
(UNK) = " & Cstr(ERR_UNK)
SF.EchoAll Cint(log_file_num), "Corrected errors _
(COR) = " & Cstr(ERR_COR)
SF.EchoAll Cint(log_file_num), "Uncorrected errors _
(UNCOR) = " & Cstr(ERR_UNCOR) - ERR_COR
SF.EchoAll Cint(log_file_num), ""
SF.EchoAll Cint(log_file_num), "Start date:" & Cstr(sim_start_date) & "  _
End date:" & Date( ) & "  End time:" & 

SF.EchoAll Cint(log_file_num), "Corrected errors _
(COR) = " & Cstr(ERR_COR)
SF.EchoAll Cint(log_file_num), "Uncorrected errors _
(UNCOR) = " & Cstr(ERROR_COUNT - ERR_COR)
SF.EchoAll Cint(log_file_num), " "

SF.EchoAll Cint(log_file_num), "Start date:" & Cstr(sim_start_date) & "  _
End date:" & Date( ) & "  End time:" & 

SF.CloseFile Cint(log_file_num) 'close log file
SF.ResetSimulation() 'reset simulator
Else
SF.Echo("HEX file error. Script Terminating")
End If
SF.Echo("End of Script")
End Sub

Private Sub PRINT_SIM_INFO()  
SF.EchoAll Cint(log_file_num), "CY=" & Cstr(SV.CYCLES) & "." & _
Cstr(SV.CYCLEJSTATE) & ":PC=" & hex(SV.PC)
End Sub

Private Sub GEN_ERROR()  
ECY = CLng(Rnd() * MAX_CYCLES) 'generate ECY
ESC = Cint(Rnd() * 12) 'generate ESC
If ESC = 0 Then ESC = 1
EQC = Cint(Rnd() * 4) 'generate EQC
If EQC = 0 Then EQC = 1
EPC = CLng(Rnd() * CODE_SIZE) 'generate EPC
End Sub

Private Sub INJECT_ERROR()  'inject errors
If CYCLE = ECY Then
If SUBCYCLE = ESC Then  
If QUANCYCLE = EQC Then
SF.EchoAll Cint(log_file_num), " *** Error reached. _
Vectoring PC=" & hex(SV.PC) & " to PC=" & hex(EPC)
SV.PC = EPC
End If
End If
End If

**** BYTE-CYCLE TRACKING MODULES ****
Private Sub BCT_RESET()  'reset BCT
Techniques intended to reduce the impact of program-flow errors on embedded systems

```plaintext
CHECK = 1
SPCIINC = 0
CHECKSTORE = 0
STORE = 0
PC_S = 0
PC_S_PRE = 0
S_PC = 0
S_PC_PRE = 0
BREG_en = 1
ERROR = 0
SPC = 0

BREG_val = 0
CREG_val = 0
INC_vali = 0
INC_valo = 0
INT_HANDLED = 0

ROM_temp = SF.InsByteCycle_A(&HO)
INSTYPE = ROM_temp
BREG = ROM_temp \ 10
CREG = ((ROM_temp mod 10) * 12) - 1

End Sub

Private Sub BCT_UPDATE()

UPDATE_IDEC()
UPDATE_BREG()
UPDATE_CREG()
UPDATE_CMP()
UPDATE_CON()
UPDATE_PC_N_SPC()
UPDATE_INC()

End Sub

Private Sub UPDATE_IDEC()
If STORE = 1 Then
Select Case QUANCYCLE
Case 1: ROM_temp = SF.GetROMByte(SV.PC)
Case 2: If SV.INT_LCALL = 1 Then
   If INT_HANDLED = 0 Then
      INSTYPE = 99
      BREG_val = 0
      CREG_val = 25
      INT_HANDLED = 1
   End If
Else
   ROM_temp = SF.InsByteCycle_I(CByte(ROM_temp))
   INSTYPE = ROM_temp
   BREG_val = ROM_temp \ 10
   CREG_val = ((ROM_temp mod 10) * 12)
End If
End Select
Else
   INT_HANDLED = 0
End If

End Sub

Private Sub UPDATE_BREG()
If STORE = 1 Then
If QUANCYCLE = 3 Then
   If CHECKSTORE = 1 Then
      BREG = BREG_val
      BREG_en = 1
   End If
End If
End If

If BREG = 0 Then
   BREG_en = 0
End If

End Sub

Private Sub UPDATE_CREG()

'update Byte Register
'when STORE = 1
'ASY3: reload BREG
'enable BREG_en flag

'when CHECKSTORE = 1
'ASY3: decrement BREG

End Sub
```

reset signals
reset buses
preload BREG and CREG
main update module
'updating IDEC
execute when STORE = 1
at specific cycle
ASY0: read ROM
'value for INT_LCALL
'0 bytes
'2 cycles (25)
'flag to denote INT_LCALL
ASY1: decode
put byte value on bus
put cycle value on bus
reset INT_LCALL
'update Cycle Register
'If BREG = 0, BREG_en = 0

O-12
Techniques intended to reduce the impact of program-flow errors on embedded systems

Dim CHK_t, CHKSTR_t, SPCINC_t

If QUANCYCLE = 0 then
    CREG = CREG - 1
    Select Case CREG
        Case 47: STORE = 0
            CHK_t = 1
        Case 46: CHK_t = 0
            SPCINC_t = 1
        Case 45: SPCINC_t = 0
            CHKSTR_t = 1
        Case 44: CHKSTR_t = 0
        Case 23: STORE = 0
            CHK_t = 1
        Case 22: CHK_t = 0
            SPCINC_t = 1
        Case 21: SPCINC_t = 0
            CHKSTR_t = 1
        Case 20: CHKSTR_t = 0
        Case 17: CHK_t = 1
        Case 16: CHK_t = 0
            SPCINC_t = 1
        Case 15: SPCINC_t = 0
            CHKSTR_t = 1
        Case 14: CHKSTR_t = 0
        Case 11: STORE = 0
            CHK_t = 1
        Case 10: CHK_t = 0
            SPCINC_t = 1
        Case 9: SPCINC_t = 0
            CHKSTR_t = 1
        Case 8: CHKSTR_t = 0
        Case 5: CHK_t = 1
        Case 4: CHK_t = 0
            SPCINC_t = 1
        Case 3: SPCINC_t = 0
            CHKSTR_t = 1
        Case 2: CHKSTR_t = 0
        Case 1: CHECK = 1
        Case 0: CHECK = 0
            STORE = 1
    End Select
    If CREG > 1 Then
        CHECK = CHK_t And BREG_en
        SPCINC = SPCINC_t And BREG_en
        CHECKSTORE = CHKSTR_t And BREG_en
    End If
End If

Private Sub UPDATE_CMP()
    If SV.PC = SPC Then
        CMP = 1
    Else
        CMP = 0
    End If
End Sub

Private Sub UPDATE_CON()
    If CHECK = 1 Then
        If QUANCYCLE = 3 Then
            If CMP <> 1 Then
                ERROR = 1
                SPC = 1
            End If
        End If
    End If
End Sub
Techniques intended to reduce the impact of program-flow errors on embedded systems

End If
End If

If CHECKSTORE = 1 Then
If QUANCYCLE = 1 Then
  If CMP <> 1 Then
    ERROR = 1
    S_PC = 1
  End If
Else
  If QUANCYCLE = 2 Then
    S_PC = 0
    PC_S = 0
  End If
End If
End If

If STORE = 1 Then
If QUANCYCLE = 1 Then
  PC_S = 1
Else
  If QUANCYCLE = 2 Then
    PC_S = 0
  End If
End If
End If

If QUANCYCLE = 0 Then ' ASY2:  S_PC = 0
S_PC = 0
End If

Private Sub UPDATE_PC_N_SPC() 'update PC and SPC module
If SPCINC = 1 Then
  If QUANCYCLE = 1 Then
    INC_vali = SPC
  End If
  If QUANCYCLE = 2 Then
    SPC = INC_valo
  End If
End If

If (PC_S = 1) And (PC_S_PRE = 0) Then 'when PC_S is set
  SPC = SV.PC
End If

If (S_PC = 1) And (S_PC_PRE = 0) Then 'when S_PC is set
  ERROR = 1
  SV.PC = SPC
End If

PC_S_PRE = PC_S 'age signals
S_PC_PRE = S_PC

End Sub

Private Sub UPDATE_INC() 'execute additional cycles
INC_valo = INC_vali + 1
End Sub

'***** CHECKING PROFILE *****
Private Function CHECK_COR_ERR() 'execute 1 cycle
Dim temp, CY, CHECK_OK

For temp = 0 to (12 - SV.CYCLE_STATE) 'execute until boundary
  SF.MicroStepSimulation
Next

SF.WriteFile Cint(log_file_num), "  {{{ Executing 10 CY" 'execute 1 cycle
CHECK_OK = true

For temp = 1 To 10
  SF.StepSimulation 'execute additional cycles
  PRINT_SIM_INFO 'and print information
  CY = Cstr(SV.CYCLES) & Cstr(SV.CYCLE_STATE)
  If Clng(CHECK_COR(CY)) <> Clng(SV.PC) Then 'checking PC with list
    'execute additional cycles
    SF.StepSimulation
    PRINT_SIM_INFO
    CY = Cstr(SV.CYCLES) & Cstr(SV.CYCLE_STATE)
    If Clng(CHECK_COR(CY)) <> Clng(SV.PC) Then
      'checking PC with list
    End If
  End If
End For

End Function
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vbnet
CHECK_OK = false  'PC <> list
temp = 11  'exit loop and report error
End If
Next
If CHECK_OK = true Then
    CHECK_COR_ERR = 1
Else
    CHECK_COR_ERR = 0
End If
End Function
Private Function CHECK_COR(CY)
    Select Case CLng(CY)
        Case 01: CHECK_COR = &h0
        Case 11: CHECK_COR = &h2
        Case 10191: CHECK_COR = &h49A
        Case 10201: CHECK_COR = &h49B
    End Select
End Function
Listing O.3: Abridged version of BCT's simulation script for Prog_A
Appendix P  CBG, HWFT and SP simulation script

The simulation scripts for each technique running with Prog_A are presented in this appendix. The scripts for other variants (for each technique) are identical apart from a few parameters. These scripts can be found in the electronic form on the CD-ROM (please refer to Appendix R).

P.1 Simulation script for CBG

```vba
Const HEX_FILENAME = "Prog_A.hex"  'Simulation parameters
Const LOG_FILENAME = "Prog_A"      'log file name
Const END_CODE = &h76B             'last programmed location+2
Const NF_START_CY = 720           'cycle that CBG starts
Const MAX_CYCLES = 1720           'upper ECY value
Const EXTRA_CYCLES = 10           'additional cycles

Dim EYC, ESC, EPC                 'Global variables
Dim CYCLE, SUBCYCLE               'internal counters
Dim ERROR_COUNT, NUM_TEST         'number of errors
Dim ERR_DET, ERR_UNDET, ERR_POVR, ERR_UNK 'error injection results
Dim INJ, LATENCY                  'error injection parameters
Dim log_file_num                  'log file number
Dim NF, NF_EN, NF_ERR_DET        'NF variables and signals
Dim CMP                             'comparator output

Public Sub Main( )
    Dim filename, FIRST_CY, SIM_ERR
    SF.ResetSimulation()      'reset simulator
    SV.SIM_ERROR_PROMPT = False  'disable error prompting
    SV.PC_ANTIALIAS_ENABLE = True  'enable ROM antialiasing
    SV.XTAL = 12000000  'clock frequency

    If SF.LoadHexFile(HEXFILENAME) = 1 Then  'load hex file
        ERROR_COUNT = Cint(SF.GetUserInput("Number of Cycles: ", Run time'))
        filename = LOGFILENAME & " (ERR=" & Cstr(ERRORCOUNT) & ").txt"
        log_file_num = SF.CreateFile(Cstr(filename))  'create log file
    SF.Echo "Hardware NF simulation"  'welcome message
    SF.Echo "NUMBER OF ERRORS=" & CStr(ERRORCOUNT)
    SF.Echo "Simulation start..."
    Randomize  'randomize RNG seed
    sim_start_time = Time  'start time and date
    sim_start_date = Date
    LATENCY = 0

    For NUM_TEST = 1 To ERROR_COUNT
        GEN_ERROR()  'main loop
    SF.EchoAll Cint(log_file_num), "ERROR #" & Cstr(NUM_TEST) & 
        ": Error cycle=" & Cstr(ECY) & ", EPC=" & hex(EPC)
    SF.ResetSimulation
        'reset MCU
    SF.RESET()  'reset NF
    FIRST_CY = True
    SIM_ERR = False  'reset 1st cycle

    For CYCLE = 0 To ECY
        For SUBCYCLE = 1 To 12
            If SV.SIM_ERROR <> SV.eNOERROR Then  'simulator OK?
                SF.EchoAll Cint(log_file_num), "Error cycle=" & Cstr(ECY)
                SF.ResetSimulation
                SF.RESET()  'reset NF
        End If
        SF.EchoAll Cint(log_file_num), "Error cycle=" & Cstr(ECY)
        SF.ResetSimulation
        SF.RESET()  'reset NF

    Next SUBCYCLE
    Next CYCLE

    SF.Echo "Simulation end..."

Next NUM_TEST
```

P.1 Simulation script for CBG

```vba
Const HEX_FILENAME = "Prog_A.hex"  'Simulation parameters
Const LOG_FILENAME = "Prog_A"      'log file name
Const END_CODE = &h76B             'last programmed location+2
Const NF_START_CY = 720           'cycle that CBG starts
Const MAX_CYCLES = 1720           'upper ECY value
Const EXTRA_CYCLES = 10           'additional cycles

Dim EYC, ESC, EPC                 'Global variables
Dim CYCLE, SUBCYCLE               'internal counters
Dim ERROR_COUNT, NUM_TEST         'number of errors
Dim ERR_DET, ERR_UNDET, ERR_POVR, ERR_UNK 'error injection results
Dim INJ, LATENCY                  'error injection parameters
Dim log_file_num                  'log file number
Dim NF, NF_EN, NF_ERR_DET        'NF variables and signals
Dim CMP                             'comparator output

Public Sub Main( )
    Dim filename, FIRST_CY, SIM_ERR
    SF.ResetSimulation()      'reset simulator
    SV.SIM_ERROR_PROMPT = False  'disable error prompting
    SV.PC_ANTIALIAS_ENABLE = True  'enable ROM antialiasing
    SV.XTAL = 12000000  'clock frequency

    If SF.LoadHexFile(HEXFILENAME) = 1 Then  'load hex file
        ERROR_COUNT = Cint(SF.GetUserInput("Number of Cycles: ", Run time'))
        filename = LOGFILENAME & " (ERR=" & Cstr(ERRORCOUNT) & ").txt"
        log_file_num = SF.CreateFile(Cstr(filename))  'create log file
    SF.Echo "Hardware NF simulation"  'welcome message
    SF.Echo "NUMBER OF ERRORS=" & CStr(ERRORCOUNT)
    SF.Echo "Simulation start..."
    Randomize  'randomize RNG seed
    sim_start_time = Time  'start time and date
    sim_start_date = Date
    LATENCY = 0

    For NUM_TEST = 1 To ERROR_COUNT
        GEN_ERROR()  'main loop
    SF.EchoAll Cint(log_file_num), "ERROR #" & Cstr(NUM_TEST) & 
        ": Error cycle=" & Cstr(ECY) & ", EPC=" & hex(EPC)
    SF.ResetSimulation
        'reset MCU
    SF.RESET()  'reset NF
    FIRST_CY = True
    SIM_ERR = False  'reset 1st cycle

    For CYCLE = 0 To ECY
        For SUBCYCLE = 1 To 12
            If SV.SIM_ERROR <> SV.eNOERROR Then  'simulator OK?
                SF.EchoAll Cint(log_file_num), "Error cycle=" & Cstr(ECY)
                SF.ResetSimulation
                SF.RESET()  'reset NF
        End If
        SF.EchoAll Cint(log_file_num), "Error cycle=" & Cstr(ECY)
        SF.ResetSimulation
        SF.RESET()  'reset NF

    Next SUBCYCLE
    Next CYCLE

    SF.Echo "Simulation end..."

Next NUM_TEST
```
Techniques intended to reduce the impact of program-flow errors on embedded systems

```
CYCLE = ECY + 4
SIM_ERR = True
Else
SF.MicroStepSimulation()
End If

If FIRST_CY = True Then
SUBCYCLE = 2
FIRST_CY = False
End If

NF_UPDATE()
Next
Next

If SIM_ERR = false Then
For SUBCYCLE = 1 To ESC
If SV.SIM_ERROR <> SV.eNOERROR Then
SUBCYCLE = 13
SIM_ERR = True
Else
SF.MicroStepSimulation()
End If
Next
End If

If SIM_ERR = false Then
'Subcycle loop
If FIRST_CY = True Then
SUBCYCLE = 2
FIRST_CY = False
End If
NF_UPDATE()
Next
Next

If SIM_ERR = false Then
'Cycle loop
If FIRST_CY = True Then
SUBCYCLE = 2
FIRST_CY = False
End If
NF_UPDATE()
Next
Next

If (SIM_ERR = False) And (NF_ERR_DET <> 1) Then
If SV.PC < END_CODE
ERR_POVR = ERR_POVR + 1
SF.EchoAll Cint(log_file_num), " \nPC ERROR NOT DETECTED ", & Cstr(ERR_DET) & "," & Cstr(ERR_POVR) & "," & Cstr(ERR_UNDET) & "," & Cstr(ERR_UNK)
End If
Else
ERR_UNDET = ERR_UNDET + 1
'Increment UNDET counter
```

P-2
Techniques intended to reduce the impact of program-flow errors on embedded systems

```plaintext
SF.EchoAll Cint(log_file_num), " $$$ PC ERROR NOT DETECTED " & Cstr(ERR_DET) & "," & Cstr(ERR_POVR) & "," & Cstr(ERR_UNK)
End If
End If
End If
END IF
PRINT_SIM_INFO()
SF.Echo ""
Next
'test loop

SF.EchoAll Cint(log_file_num), "Detectable errors _ (DET) = " & Cstr(ERR_DET)
SF.EchoAll Cint(log_file_num), "Possible overridden errors _ (POVR) = " & Cstr(ERR_POVR)
SF.EchoAll Cint(log_file_num), "Undetectable errors _ (UNDET) = " & Cstr(ERR_UNDET)
SF.EchoAll Cint(log_file_num), "SIMULATOR errors _ (UNK) = " & Cstr(ERR_UNK)
SF.EchoAll Cint(log_file_num), " LATENCY = " & Cstr(LATENCY / ERR_DET)
SF.EchoAll Cint(log_file_num), " Start date:" & Cstr(sim_start_date) & " Start time:" & Cstr(sim_start_time)
SF.EchoAll Cint(log_file_num), " End date:" & Date() & " End time:" & time()
SF.CloseFile Cint(log_file_num)
SF.ResetSimulation()
Else
SF.Echo("HEX file error. Script Terminating")
End If
SF.Echo("End of Script")
End Sub

Private Sub PRINT_SIM_INFO()
SF.EchoAll Cint(log_file_num), "CY=" & Cstr(SV.CYCLES) & " :PC=" & hex(SV.PC)
End Sub

Private Sub GEN_ERROR()
Do
ECY = CLng(Rnd()) * MAX_CYCLES
Loop While (ECY < NF_START_CY)

Do
ESC = Cint(Rnd()) * 12
Loop While (ESC = 0)
EPC = CLng(Rnd()) * 65535

End Sub

Private Sub INJECT_ERROR()
If CYCLE = ECY Then
If SUBCYCLE = ESC Then
SF.EchoAll Cint(log_file_num), " *** Error reached. Vectoring PC=" & hex(SV.PC) & " to PC=" & hex(EPC)
SV.PC = EPC
End If
End If
End Sub

***** NOP Fills MODULES *****

Private Sub NF_RESET()
SF.PutSFRVal &hC0, &h01
NF_EN = 0
NF_ERR_DET = false
End Sub

Private Sub NF_UPDATE()
If (SF.CheckBit(SV.CSFR, &hC0, 4) = 1) And (SF.CheckBit(SV.CSFR, &hC0, 0) = 0) Then NF_EN = 1
End If
If (SF.CheckBit(SV.CSFR, &hC0, 4) = 0) And _
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vbnet
(SF.CheckBit(SV.cSFR, &hCO, 0) = 1) Then
    NF_EN = 0
End If

NF = (SF.GetSFRVal(&hC2) * 256) + SF.GetSFRVal(&hCl) 'NF equivalent value
If SV.PC > NF Then 'comparator
    CMP = 1
Else
    CMP = 0
End If

If SV.Cycle_state = 1 Then
    If NF_EN = 1 Then
        If CMP = 1 Then
            NF_ERR_DET = 1
        End If
    End If
End If
End Sub
```

Listing P.1: CBG's simulation script for Prog_A

### P.2 Simulation script for HWFT

```vbnet
Const HEX_FILENAME = "Prog_A.hex" 'Simulation parameters
Const LOG_FILENAME = "Prog_A" 'log file name
Const END_CODE = &h0810 'last code position + 1
Const FT_START_CY = 710 'cycle that FT starts
Const MAX_CYCLES = 1710 'upper ECY value
Const EXTRA_CYCLES = 500 'maximum cycles to detect

Dim ECY, ESC, EPC 'Global variables
Dim CYCLE, SUBCYCLE 'internal counters
Dim ERROR_COUNT, NUM_TEST 'number of errors
Dim ERR_DET, ERR_UNDET, ERR_POVR, ERR_UNK 'error injection results
Dim sim_start_time, sim_start_date 'start date and time
Dim log_file_num 'log file number
Dim FT_EN, FT_ERR_DET 'FT variables and signals
Dim CMP, CMP1, CMP_A, CMP_B 'comparator output/inputs

Public Sub Main()
    Dim filename, FIRST_CY, SIM_ERR 'reset simulator
    SF.ResetSimulation()
    SV.SIM_ERROR_PROMPT = False 'disable error prompting
    SV.PC_ANTIALIAS_ENABLE = True 'enable ROM antialiasing
    SV.XTAL = 12000000 'clock frequency

    If SF.LoadHexFile(HEX_FILENAME) = 1 Then 'load hex file
        ERROR_COUNT = CInt(SF.GetUserInput("Number of Cycles: ", "Run time"))
        filename = LOG_FILENAME & " ( ERR= " & CStr(ERROR_COUNT) & ").txt"
        log_file_num = SF.CreateFile(Cstr(filename)) 'create log file

        SF.Echo "Hardware FT simulation" 'welcome message
        SF.Echo "NUMBER OF ERRORS=" & CStr(ERROR_COUNT) 'log file number
        SF.Echo "Simulation start..." 'wait time and date

        Randomize
        sim_start_time = Time
        sim_start_date = Date
        LATENCY = 0

        For NUM_TEST = 1 To ERROR_COUNT 'main loop
            GEN_ERROR() 'generates error
            SF.EchoAll CInt(log_file_num), "ERROR #" & CStr(NUM_TEST) & ": Error cycle=" & Cstr(ECY) & ": Error cycle=" & Cstr(ESC) & ": EPC=" & hex(EPC)
            SF.ResetSimulation 'reset MCU
        Next
    End If
End Sub
```

P-4
Techniques intended to reduce the impact of program-flow errors on embedded systems

```
FT_RESET()  'reset FT
FIRST_CY = True  'reset 1st cycle
SIM_ERR = False

For CYCLE = 0 To ECY  'run to ECY
    For SUBCYCLE = 1 To 12
        If SV.SIM_ERROR <> SV.eNOERROR Then  'making simulator is OK
            SUBCYCLE = 13
            CYCLE = ECY + 4
            SIM_ERR = True
        Else
            SF.MicroStepSimulation()  'if OK, simulate
        End If
        If FIRST_CY = True Then
            SUBCYCLE = 2
            FIRST_CY = False
        End If
    Next
    FT_UPDATE()  'execute FT
Next  'cycle loop

If SIM_ERR = false Then
    For SUBCYCLE = 1 To ESC
        If SV.SIM_ERROR <> SV.eNOERROR Then  'making simulator is OK
            SUBCYCLE = 13
            SIM_ERR = True
        Else
            SF.MicroStepSimulation()  'if OK, simulate
        End If
    Next
    FT_UPDATE()  'execute FT
Next  'cycle loop

If SIM_ERR = false Then
    SV.PC = EPC  'inject error
    INJ = SV.CYCLES
    SF.EchoAll Cint(log_file_num),  "  ERROR INJECTED, CY=" & _
        Cstr(SV.CYCLES) & ",", & Cstr(SV.CYCLE_STATE) & ",", & Cstr(SV.PC)
End If

If SIM_ERR = True Then
    ERRJNJK = ERRJNJK + 1
    SF.EchoAll Cint(log_file_num),  "  +++ SIMULATOR ERROR ("  & _
        Cstr(SV.SIM_ERROR) & ")  "," & _
        Cstr(ERRPOVR) & "," & Cstr(ERRNDET) & "," & _
        Cstr(ERRJNJK)
Else
    For SUBCYCLE = 1 To (EXTRA_CYCLES * 12)
        If SV.SIM_ERROR <> SV.eNOERROR Then  'making simulator is OK
            SIM_ERR = True
            SUBCYCLE = (EXTRA_CYCLES * 12) + 1
            ERUJN = ERR_UNK + 1
            SF.EchoAll Cint(log_file_num),  "  +++ SIMULATOR ERROR ("  & _
                Cstr(SV.SIM_ERROR) & ")  "," & _
                Cstr(ERRPOVR) & "," & Cstr(ERRNDET) & "," & _
                Cstr(ERRJNJK)
        Else
            SF.MicroStepSimulation()  'if OK, simulate
        End If
    End If
End If

If SIM_ERR = False Then
    FT_UPDATE()  'execute FT
End If
```

P-5
Techniques intended to reduce the impact of program-flow errors on embedded systems

If (SIM_ERR = False) And (FT_ERR_DET = False) Then
If SV.PC < END_CODE Then
   ERR_POVR = ERR_POVR + 1  ' increment POVR counter
   SF.EchoAll Cint(log_file_num),  "PC ERROR POSSIBLY OVERRIDDEN " & Cstr(ERR_DET) & ", " & Cstr(ERR_POVR) & ", " & Cstr(ERR_UNDET) & ", " & Cstr(ERR_UNK)
Else
   ERR_UNDET = ERR_UNDET + 1  ' increment UNDET counter
   SF.EchoAll Cint(log_file_num),  "PC ERROR NOT DETECTED " & Cstr(ERR_DET) & ", " & Cstr(ERR_POVR) & ", " & Cstr(ERR_UNDET) & ", " & Cstr(ERR_UNK)
End If
End If
End If
PRINT_SIM_INFO()
SF.Echo ""
Next
' test loop
SF.EchoAll Cint(log_file_num),  "Detectable errors (DET) = " & Cstr(ERR_DET)
SF.EchoAll Cint(log_file_num),  "Possible overridden errors (POVR) = " & Cstr(ERR_POVR)
SF.EchoAll Cint(log_file_num),  "Undetectable errors (UNDET) = " & Cstr(ERR_UNDET)
SF.EchoAll Cint(log_file_num),  "SIMULATOR errors (UNK) = " & Cstr(ERR_UNK)
SF.EchoAll Cint(log_file_num),  ""
If ERR_DET > 0 Then
   SF.EchoAll Cint(log_file_num),  "LATENCY = " & Cstr(LATENCY / ERR_DET)
End if
SF.EchoAll Cint(log_file_num),  ""
SF.EchoAll Cint(log_file_num),  "Start date: " & Cstr(sim_start_date) & ", Start time: " & Cstr(sim_start_time)
SF.EchoAll Cint(log_file_num),  "End date: " & Date() & ", End time: " & time()
SF.CloseFile Cint(log_file_num)  ' close log file
SF.ResetSimulation()  ' reset simulator
Else
SF.Echo("HEX file error. Script Terminating")
End if
SF.Echo("End of Script")
End Sub

Private Sub PRINT_SIM_INFO()
SF.EchoAll Cint(log_file_num),  "CY= " & Cstr(SV.CYCLES) & ": PC= " & hex(SV.PC)
End Sub

Private Sub GEN_ERROR()
    DO
       ECY = CLng(Rnd() * MAX_CYCLES) ' generate ECY
       Loop While (ECY < FT_START_CY)
       ESC = CInt(Rnd() * 12) ' generate ESC (1 to 12)
       Loop While (ESC = 0)
       EPC = CLng(Rnd() * 4095) ' generate EPC
    End Sub

Private Sub INJECT_ERROR()
' inject errors
If CYCLE = ECY Then
   IF CYCLE = ECY Then
      SV.PC = EPC
   End If
End If
End Sub

'***** Hardware Function Token MODULES *****
Private Sub FT_RESET()
SF.PutSFRVal &hC0, &h02  ' reset PT
End Sub
Techniques intended to reduce the impact of program-flow errors on embedded systems

Private Sub FT_UPDATE()
    If (SF.CheckBit(SV.cSFR, &hCO, 5) = 1) And _
        (SF.CheckBit(SV.cSFR, &hC0, 1) = 0) Then
        FT_EN = 1
    End If

    If (SF.CheckBit(SV.cSFR, &hCO, 5) = 0) And _
        (SF.CheckBit(SV.cSFR, &hC0, 1) = 1) Then
        FT_EN = 0
    End If

    CMP_A = SF.GetSFRVal(&hC3)
    CMP_B = SF.GetSFRVal(&hC4)
    If CMP_A <> CMP_B Then
        CMP1 = 1
    Else
        CMP1 = 0
    End If

Select Case SV.Cycle_state
    Case 1: If (SV.SFR_CHANGE = True) And _
        (SV.SFR.CHG_VAL = &hC4) Then
        CMP = CMP1
    End If

    Case 2: If (FT_EN = 1) And (CMP = 1) Then
        FT_ERR_DET = True
    End If
End Select
End Sub

Listing P.2: HWFT's simulation script for Prog_A
Techniques intended to reduce the impact of program-flow errors on embedded systems

P.3 Simulation script for SecurePorts

Const CODE_SIZE = &H1FFF 'Simulation parameters

Dim RUN_CYCLES 'Global variables
Dim SUBCYCLE 'sub cycle executed
Dim sim_start_time, sim_start_date 'start date and time
Dim log_filename, log_filenum 'file number of log file
Dim P1_WRITTEN 'Statistics
Dim SPI_WRITTEN 'SPI written
Dim UPDtoDate 'RP is updated
Dim NU_NIT 'RP not update, not in time
Dim NU_SPI_ONLY 'RP not update, 1 wr. only
Dim NU_SPI_DIFF 'RP not update, diff.
Dim CNT 'SP variables and signals
Dim CNT_update_flag, dec_flag, test_flag
Dim run_flag, busy_flag
Dim REAL_PORT
Dim CMPort
Dim enCMP

Public Sub Main()
    Dim cycle_loop, run_cycles, FILENAME, hex_filename
    SF. ResetSimulation() 'reset simulator
    SV.SIM_ERROR_PROMPT = False 'disable error prompting
    SV.PC_antialias_enable = True 'antialias ROM
    SV.Serial_enable = False 'disable USART

    FILENAME = Cstr(SF.GetUserInput("Program name (exc. extension)", _
    "Program details")")
    hex_filename = FILENAME + ".hex" 'hex and log filenames

    If SF.LoadHexFile(Cstr(hex_filename)) = 1 Then 'load hex file
        run_cycles = Clng(SF.GetUserInput("Number of cycles to execute: ", _
        "Run time")
        log_filename = "SP simulation " & FILENAME & 
        " (CY= " + Cstr(run_cycles) + ").txt"
        log_filenum = SF.CreateFile(Cstr(log_filename) )
        SP_RESET() 'reset SP
        sim_start_time = Time 'start time and date
        sim_start_date = Date

        For cycle_loop = 0 To run_cycles - 1 'cycle loop
            For SUBCYCLE = 1 To 12 'subcycle loop
                PRINT_SIM_INFO() 'single step
                SF.MicroStepSimulation() 'update SecurePort
            Next
        Next

        SF.EchoAll Cint(log_filenum), "" 'cycle loop
        SF.EchoAll Cint(log_filenum), "P1 writes = " & Cstr(P1_WRITTEN) 'cycle loop
        SF.EchoAll Cint(log_filenum), "SPI writes = " & Cstr(SPI_WRITTEN) 'cycle loop
        SF.EchoAll Cint(log_filenum), "" 'cycle loop
        SF.EchoAll Cint(log_filenum), "RP UPDATE = " & Cstr(UPDtoDate) 'cycle loop
        SF.EchoAll Cint(log_filenum), "RP NOT UPDATE (NOT IN TIME) = " & Cstr(NU_NIT) 'cycle loop
        SF.EchoAll Cint(log_filenum), "RP NOT UPDATE (SPI only written) = " & _
        Cstr(NU_SPI_ONLY) 'cycle loop
        SF.EchoAll Cint(log_filenum), "" 'cycle loop
        SF.EchoAll Cint(log_filenum), "Start date: " & Cstr(sim_start_date) & 
        " Start time: " & Cstr(sim_start_time) 'cycle loop
        SF.EchoAll Cint(log_filenum), "End date:" & Date() & " End time:" & time() 'cycle loop

        SF.CloseFile Cint(log_filenum) 'close log file
        SF.ResetSimulation() 'reset simulator
    End If

End Sub
Techniques intended to reduce the impact of program-flow errors on embedded systems

SF.Echo("End of Script")

Private Sub PRINT_SIM_INFO()
    SF.WriteFile Cint(log_filenum),  "CY=" & Cstr(SV.CYCLES) & ":PC=" & hex(SV.PC) &":XT=" & Cstr(SF.InsByteCycle_A(SV.PC)) &":DF=" & Cstr(dec_flag) &":tft=": & Cstr(test_flag) &":RF=" & Cstr(run_flag) &":CNT=" & Cstr(CNT) &":Pl=" & hex(SV.PI) &":SP1=" & hex(SF.GetSFRVal(&H91)) &":RP=" & REAL_PORT & ":SFR_CHANGE=" & Cstr(SV.SFR_CHANGE) & ":SFR_CHG_VAL=" & hex(SV.SFR_chg_val)
End Sub

****** SecurePort MODULES: *****
Private Sub SP_RESET()
    CNT = 4 'default CNT value
    CNT_update_flag = 0
    dec_flag = 0
    test_flag = 0
    run_flag = 0
    busy_flag = 0
    CMPort = 0
    enCMP = 0
    REAL_PORT = 0
End Sub

Private Sub SP_UPDATE()
    UPDATE_CNTR()
    UPDATE_CMP()
    UPDATE_SPcnt()
End Sub

Private Sub UPDATE_CNTR()
    DetectWrite()
    Select Case(SV.CYCLE_STATE)
        Case 1: If test_flag = 1 Then
            If dec_flag = 1 Then
                run_flag = 1
            Else
                NU_SP1_ONLY = NU_SP1_ONLY + 1
            End If
        Else
            If dec_flag = 1 Then
                If CNT <> 0 Then
                    CNT = CNT - 1
                Else
                    dec_flag = 0
                    busy_flag = 0
                    CNT_update_flag = 1
                    NU_NIT = NU_NIT + 1
                End If
            End If
        Case 2: test_flag = 0
            If run_flag = 1 Then
                dec_flag = 0
                CNT = CNT - 1
                busy_flag = 0
                CNT_update_flag = 1
                NU_NIT = NU_NIT + 1
            End If
        Case 3: If run_flag = 1 Then
            If CNT_update_flag = 1 Then
                If busy_flag <> 1 Then
                    CNT_update_flag = 0
                    CNT = SF.GetSFRVal(&H92) And &H3F
                End If
            End If
        Case 4: If run_flag = 1 Then
            If run_flag = 1 Then
                run_flag = 0
                busy_flag = 0
            End If
    End Select
End Sub

Private Sub UPDATE_CMP()
    'update Compare

Private Sub UPDATE_SPcnt()
    'start checking PI and SP1
    'only SP1 written
    'reset decrement flag
    'reset decrement flag
    'reset counter value
    'reset all signals
    'SP1 written in time
    'reset SP
    'SP1 not written in time
    'update Compare

P-9
Techniques intended to reduce the impact of program-flow errors on embedded systems

If enCMP = 1 Then
    If SF.GetSFRVal(&H90) = SF.GetSFRVal(&H91) Then
        REAL_PORT = SF.GetSFRVal(&H90)
        ' if SP1 = Pl, show value
        SF.WriteFile Cint(log_filename), " ***** REAL_PORT updated *****"
        UPDATED = UPDATED + 1
    Else
        NU_SP1_DIFF = NU_SP1_DIFF + 1
    End If
End If
End Sub

Private Sub UPDATE_SPcnt()
    If SV.SFR_CHANGE = True Then
        If SV.SFR_CHG_VAL = &H92 Then
            If (SF.GetSFRVal(&H92) And &HC0) = &H80 Then CNT_update_flag = 1
        End If
    End If
End Sub

Private Sub DetectWrite()  'detect Pl/SP1 writes
    If SV.SFR_CHANGE = True Then
        If SV.SFR_CHG_VAL = &H90 Then
            dec_flag = 1
            busy_flag = 1
            P1_WRITTEN = P1_WRITTEN + 1
            SF.WriteFile Cint(log_filename), _
            " $ P1 CHANGE (" & Cstr(P1_WRITTEN) & ")"
        End If
    End If

    If SV.SFR_CHANGE = True Then
        If SV.SFR_CHG_VAL = &H91 Then
            test_flag = 1
            SP1_WRITTEN = SP1_WRITTEN + 1
            SF.WriteFile Cint(log_filename), _
            " $ SP1 CHANGE (" & Cstr(SP1_WRITTEN) & ")"
        End If
    End If
End Sub

Listing P.3: SP's simulation script for Prog_A
Appendix Q  VHDL source for ST2, CBG, HWFT and SP

The VHDL source code for ST2, as presented in Chapter 12, is shown in the first half of this appendix. The source code of CBG, HWFT and SP is given in its integrated form (i.e. all three techniques are implemented together), as implemented in Chapter 18.

Q.1 VHDL source for ST2

ST2’s schematic is shown in Figure Q.1. The VHDL source code for each block is listed accordingly.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity PC is
    port (
        rst: in STD_LOGIC; -- reset
        clk: in STD_LOGIC; -- clock
        PCi: in STD_LOGIC_VECTOR (15 downto 0); -- PC input value (from bus)
        PCo: out STD_LOGIC_VECTOR (15 downto 0); -- PC output value (to bus)
        PCWRi: in STD_LOGIC; -- signal to write PC
        SPCo: out STD_LOGIC_VECTOR (15 downto 0); -- to SPC
        SPCI: in STD_LOGIC_VECTOR (15 downto 0); -- from SPC
        S_PCI: in STD_LOGIC; -- signal to load PC with SPC
        INJi: in STD_LOGIC; -- signal to inject error
        INJvi: in STD_LOGIC_VECTOR (15 downto 0) -- error injection value
    );
end PC;

architecture bhv of PC is
    signal PCreg: STD_LOGIC_VECTOR(15 downto 0); -- PC itself
    signal PCval: STD_LOGIC_VECTOR(15 downto 0);
    signal reset: STD_LOGIC;

Figure Q.1: Schematic of ST2's implementation
```
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
begin
  process(reset, clk, PCWRi)
  begin
    if reset = '1' then
      PCreg <= PCval;
    elsif rising_edge(clk) then
      if PCWRi = '1' then
        PCreg <= PCi;
      end if;
    end if;
  end process;

  reset <= rst or S_PCi or INJi;
  PCval <= x'0000' when rst = '1' else
           SPCi when S_PCi = '1' else
           INJvi when INJi = '1' else
           (others => '-');
  PCo <= PCreg;
  SPCo <= PCreg;
end bhv;

Listing Q.1: VHDL source code for the Program Counter (PC)
```

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity SPC is
  port(
    rst: in STD_LOGIC;
    PCi: in STD_LOGIC_VECTOR(15 downto 0);
    SPCo: out STD_LOGIC_VECTOR(15 downto 0);
    PC_Si: in STD_LOGIC
  );
end SPC;

architecture bhv of SPC is
  signal SPCreg: STD_LOGIC_VECTOR(15 downto 0); -- register
begin
  process(rst, PC_Si)
  begin
    if rst = '1' then
      SPCreg <= x'0000';
    elsif rising_edge(PC_Si) then
      SPCreg <= PCi;
    end if;
  end process;
  SPCo <= SPCreg;
end bhv;

Listing Q.2: VHDL source code for the Shadow Program Counter (SPC)
```

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity CREG is
  port(
    rst: in STD_LOGIC; -- reset
    clk: in STD_LOGIC; -- clock (four times MCU clock)
    CHKo: out STD_LOGIC; -- CHK output signal
    STRo: out STD_LOGIC; -- STR output signal
    CHKSTRo: out STD_LOGIC; -- CHKSTR output signal
    QCYo: out STD_LOGIC_VECTOR(1 downto 0) -- QCY output
  );
end CREG;

architecture bhv of CREG is
  signal CREGreg: STD_LOGIC_VECTOR(3 downto 0); -- register
  signal QCYreg: STD_LOGIC_VECTOR(1 downto 0);
  signal CHK: STD_LOGIC;
begin
end CREG;

Q-2
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
signal CHKSTR: STD_LOGIC;
signal STR: STD_LOGIC;
signal QCY123: STD_LOGIC;
signal QCY012: STD_LOGIC;

begin
  process(rst, clk4)
  begin
    if rst = '1' then
      CREGreg <= "0000";
    elsif rising_edge(clk4) then
      if QCYreg = 3 then
        if CREGreg = 11 then
          CREGreg <= "0000";
        else
          CREGreg <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER( CREGreg) + 1), 4) ;
        end if;
      else
        QCYreg <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(QCYreg) + 1), 2);
      end if;
    end if;
  end process;

  CHK <= '1' when CREGreg = "0000" else '1' when CREGreg = "0110" else
         '1' when CREGreg = "1010" else 'O';
  STR <= '1' when CREGreg = "1011" else 'O';
  CHKSTR <= '1' when CREGreg = "0010" else '1' when CREGreg = "1000" else 'O';
  QCY123 <= 'O' when QCYreg = "00" else '1';
  QCY012 <= 'O' when QCYreg = "11" else '1';
  CHKo <= CHK and QCY123;
  CHKSTRo <= CHKSTR and QCY012;
  STRo <= STR and QCY012;
  QCYo <= QCYreg;
end bhv;
```

Listing Q.3: VHDL source code for the Cycle Register (CREG) module

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity COMP is
  port (
    PCi: in STD_LOGIC_VECTOR(15 downto 0);
    SPCI: in STD_LOGIC_VECTOR(15 downto 0);
    CMPOo: out STD_LOGIC;
    CMPlo: out STD_LOGIC);
end COMP;

architecture bhv of COMP is
begin
  CMPOo <= 111 when PCi = SPCI else 'O';
  CMPlo <= '1' when PCi = CONV_STD_LOGIC_VECTOR((CONV_INTEGER(SPCI) + 1), 16) _
         else 'O';
end bhv;
```

Listing Q.4: VHDL source code for the Comparator (COMP) module

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity CTRL is
  port (    
end;
```

Q.3
Techniques intended to reduce the impact of program-flow errors on embedded systems

```
rst:  in STD_LOGIC;            -- reset
clk4: in STD_LOGIC;            -- clock (four times MCU clock)
CHKi: in STD_LOGIC;            -- control signal inputs
STRI: in STD_LOGIC;
CHKSTRI: in STD_LOGIC;
QCYi: in STD_LOGIC_VECTOR(1 downto 0);  -- quan-cycle input
CMP0i: in STD_LOGIC;          -- COMP inputs
CMP1i: in STD_LOGIC;
S_PCo: out STD_LOGIC;          -- PC and SPC signal outputs
PC_So: out STD_LOGIC;
ERRORo: out STD_LOGIC;        -- error signal
```

end CTRL;

architecture bhv of CTRL is

signal QCY0: STD_LOGIC;
signal QCY1: STD_LOGIC;
signal QCY3: STD_LOGIC;
signal ERRO: STD_LOGIC;
signal ERR1: STD_LOGIC;
signal ERRCHK: STD_LOGIC;
signal ERROR: STD_LOGIC := '0';

begin

process (rst, clk4)
begin
  if rst = '1' then
    ERRO <= 10' when QCYi = "00" else '0';
  elsif rising_edge(clk4) then
    ERRO <= ERRO;
  end if;
end process;

QCY0 <= '1' when QCYi = "00" else '0';
QCY1 <= '1' when QCYi = "01" else '0';
QCY3 <= '1' when QCYi = "11" else '0';

ERR0 <= (not CMP0i) or CMP1i;
ERR1 <= CMP0i xnor CMP1i;
ERRCHK <= CHK1 and QCY3 and ERRO;
S_PCo <= ERRCHK;
PC_So <= (STRI and QCY1) or (CHKSTRI and QCY1 and (not ERR1));
ERROR <= '1' when ERRCHK = '1' else '1' when ((CHKSTRI = '1') and (QCY1 = '1') and (ERR1 = '1')) else
       ERROR;
end bhv;

Listing Q.5: VHDL source code for the Control (CTRL) module

Q.2 VHDL source for CBG, HWFT and SP (integrated)
The integrated schematic for CBG, HWFT and SP is shown in Figure Q.2. The
NOT_END_INS module prevents the control module from vectoring to the ISR on non-
instruction boundaries to prevent EMIT errors. The immediate vectoring technique is
implemented here (see Chapter 15).

The PULSE_STRETCH module is necessary to lengthen P1's write pulse in order for SP to
detect it. The VHDL source code for each block is listed accordingly.
Techniques intended to reduce the impact of program-flow errors on embedded systems

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;

entity CTRL is
  port:
    clk: in STD_LOGIC;
    rst: in STD_LOGIC;
    csp: in te51state;
    EVECL: in STD_LOGIC_VECTOR(7 downto 0);  -- Error vector
    EVECH: in STD_LOGIC_VECTOR(7 downto 0);
    EDINT: in STD_LOGIC_VECTOR(7 downto 0);
    CBGerr: in STD_LOGIC;
    FTerri: in STD_LOGIC;
    SPerri: in STD_LOGIC;
    NOT_END_INS: in STD_LOGIC;
    irq_edco: out STD_LOGIC;
    irq_vec_edco: out STD_LOGIC_VECTOR(23 downto 0);
    int_state: out STD_LOGIC_VECTOR(1 downto 0);
    EDINTo: out STD_LOGIC_VECTOR(7 downto 0);
  end CTRL;

architecture bhv of CTRL is
  signal int_state: STD_LOGIC_VECTOR(1 downto 0);
  signal flag_pre: STD_LOGIC_VECTOR(2 downto 0);
  signal flag_cur: STD_LOGIC_VECTOR(2 downto 0);
  signal REDC_pre: STD_LOGIC;
  signal REDC_cur: STD_LOGIC;
  signal irq_edc: STD_LOGIC;
  signal irq_vec_edco: STD_LOGIC_VECTOR(23 downto 0);
begin
  process(clk, rst)
  begin
    if rst='1' then
      int_state <= "00";
      flag_pre <= "000"; flag_cur <= "000";
      REDC_pre <= '0'; REDC_cur <= '0';
  end process;
end bhv;

Figure Q.2: Schematic of CBG, HWFT and SP's implementation
Techniques intended to reduce the impact of program-flow errors on embedded systems

Listing Q.6: VHDL source code for the Control (CTRL) module

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;

entity te51not_end_ins is
  port(
    elk: in STD_LOGIC;
    rst: in STD_LOGIC;
    mcodei:  in te51mcodeT;
    csp: in te51stateT;
    NOT_END_INSo:  out STD_LOGIC);
end entity;

architecture bhv of te51not_end_ins is
  signal int_state: string(2 downto 0);
  signal flag_cur: te51stateT;
  signal REDC_cur: te51stateT;
  signal REDC_pre: te51stateT;
  signal EDINTo: te51stateT(7 downto 3);
  signal irq_edc: STD_LOGIC;
  signal irq_vec_edc: te51stateT(16 downto 0);

  process (elk)
  begin
    if rising_edge(clk) then
      case csp is
        when C2S3P2 =>
          if int_state = "10" then
            irq_edc <= '0';
            int_state <= "11";
          end if;
        when C1S5P1 | C2S5P1 =>
          if int_state = "00" then
            flag_pre <= flag_cur;
          end if;
        when C1S6P1 | C2S6P1 =>
          if int_state = "00" then
            flag_cur(0) <= CBGerri;
            flag_cur(1) <= FTerri;
            flag_cur(2) <= SPerri;
            REDC_CUr <= EDINTi(7) ;
          end if;
        when C1S6P2 | C2S6P2 =>
          if int_state = "01" then
            if flag_cur > flag_pre then
              int_state <= "01";
            end if;
          end if;
        when others =>
          null;
      end case;
      end if;
      end process;

      EDINTo(7 downto 3) <= EDINTi(7 downto 3); -- EDINT output
      EDINTo(2) <= SPerri when int_state = "00" else flag_cur(2);
      EDINTo(1) <= FTerri when int_state = "00" else flag_cur(1);
      EDINTo(0) <= CBGerri when int_state = "00" else flag_cur(0);
      irq_edco <= irq_edc;
      irq_vec_edco <= irq_vec_edc;
      int_stateo <= int_state;
  end architecture;
```

Q-6
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
begin
  with mcodei.bc select
  cyclesl<-
  true when te51bc_lblc | te51bc_2blc,
  false when others;
process(clk, rst)
begin
  if rst= '1' then
    NOT_END_INS <= 'O';
  elsif rising_edge(clk) then
    case csp is
    when ClSIPI=>
      NOT_END_INS <= '1'; -- set signal high
    when C1S5P2=>
      if cyclesl then NOT_END_INS <= 'O'; end if; -- reset signal
    when C2S5P2=>
      NOT_END_INS <= 'O'; -- reset signal
    when others=>
      null;
    end case;
  end if;
end process;
NOT_END_INSo <= NOT_END_INS;
end architecture;
```

Listing Q.7: VHDL source code for the NOT_END_INS module

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;

entity CBG is
  port(
    elk: in STD_LOGIC;
    rst: in STD_LOGIC;
    csp: in te51stateT;
    EDCi: in STD_LOGIC_VECTOR(7 downto 0);
    CBGLi: in STD_LOGIC_VECTOR(7 downto 0);
    CBGHi: in STD_LOGIC_VECTOR(7 downto 0);
    PCi: in STD_LOGIC_VECTOR(15 downto 0);
    CBGerro: out STD_LOGIC;
  );
end CBG;

architecture bhv of CBG is
signal CBG: STD_LOGIC_VECTOR(15 downto 0); -- cycle, state, phase input
signal CBGen: STD_LOGIC; -- CBG enable flag
signal COMP_A: STD_LOGIC; -- compare (to be ANDed)
signal COMP: STD_LOGIC; -- compare (after ANDing)
signal CBGerr: STD_LOGIC; -- error flag

alias CBGen0: STD_LOGIC is EDCi(O);
alias CBGen1: STD_LOGIC is EDCi(4);

process(clk, rst)
begin
  if rst= '1' then
    CBGen <= 'O'; CBGerr <= 'O'; -- reset flags
  elsif rising_edge(clk) then
    case csp is
    when C1S6P1 | C2S6P1 =>
      CBGerr <= '0'; -- reset error flag
    when C1S6P2 | C2S6P2 =>
      CBGerr <= COMP and CBGen; -- compare
    when others =>
      null; -- do nothing
    end case;
  end if;
end process;
```

Q-7
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
if (CBGenO = '0') and (CBGenl = '1') then  -- CBG enabled if
  CBGen <= '1';
elsif (CBGenO = '1') and (CBGenl = '0') then  -- and disabled if
  CBGen <= '0';
end if;
end if;
end process;
```

```vhdl
CBG(7 downto 0) <= CBGLi;  -- arranging CBG bits
CBG(15 downto 8) <= CBGHi;
COMP <= '1' when (PCi > CBG) else '0';  -- compare flag
CBGerro <= CBGerr;  -- error flag output
end architecture;
```

Listing Q.8: VHDL source code for the Code Boundary Guard module

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.te51pak.all;

entity HWFT is
  port(
    elk: in STD_LOGIC;
    rst: in STD_LOGIC;
    csp: in te51stateT;
    EDCi: in STD_LOGIC_VECTOR 7 downto 0);
  FTvali: in STD_LOGIC_VECTOR 7 downto 0;
  FTcDkwri in STD_LOGIC;
  FTerro: out STD_LOGIC);
end HWFT;

architecture bhv of HWFT is
  signal FTen: STD_LOGIC;
  signal CMP1: STD_LOGIC;
  signal CMP: STD_LOGIC;
  signal FTerr: STD_LOGIC;
  alias FTenO: STD_LOGIC is EDCi(1);
  alias FTenl: STD_LOGIC is EDCi(5);
begin
  process(elk, rst)
  begin
    if rst = '1' then
      FTen <= '0'; FTerr <= '1';
      CMP <= '0';
    elsif rising_edge(elk) then
      case csp is
        when C1S1P2 | C2S1P2 =>
          FTerr <= FTen and CMP;
        when C1S6P1 | C2S6P1 =>
          FTerr <= '0';
        when others =>
          null;
      end case;
    end if;
    if FTckwri = '1' then
      CMP <= CMP1;
    end if;
    if (FTenO = '0') and (FTenl = '1') then
      FTen <= '1';
    elsif (FTenO = '1') and (FTenl = '0') then
      FTen <= '0';
    end if;
  end process;
  CMPl <= '1' when (FTvali /= FTchki) else '0';  -- CMP latch input
  FTerro <= FTerr;  -- error flag output
end architecture;
```

Listing Q.9: VHDL source code for the Hardware Function Token module
Techniques intended to reduce the impact of program-flow errors on embedded systems

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.te51pak.all;

description SecurePort is
  port (  
    Clk: in STD_LOGIC; -- clock
    rst: in STD_LOGIC; -- reset
    csp: in te51stateT; -- cycle, state, input
    EDCi: in STD_LOGIC_VECTOR(7 downto 0); -- SFR input
    SPCnti: in STD_LOGIC_VECTOR(7 downto 0); -- Pi input
    Plwri: in STD_LOGIC; -- PI write enable input
    Plo: out STD_LOGIC_VECTOR(7 downto 0); -- Pi output
    Plwro: out STD_LOGIC; -- Pi write enable output
    SPerro: out STD_LOGIC); -- error flag
end description SecurePort;

description bhv of SecurePort is
  signal PIA: STD_LOGIC_VECTOR(7 downto 0); -- Pi first write
  signal PI_B: STD_LOGIC_VECTOR(7 downto 0); -- Pi second write
  signal PATH: STD_LOGIC_VECTOR(2 downto 0); -- PATH enumeration
  signal CNT: STD_LOGIC_VECTOR(7 downto 0); -- 4 bit counter
  signal CMP: STD_LOGIC; -- compare flag (comb)
  signal CNTzr: STD_LOGIC; -- CNT zero flag (comb)
  signal SPen: STD_LOGIC; -- SP enable flag (ff)
  signal Idle: STD_LOGIC; -- idle flag (FF)
  signal Awr: STD_LOGIC; -- Pi first write flag (FF)
  signal Bwr: STD_LOGIC; -- Pi second write flag (FF)
  signal SPerr: STD_LOGIC; -- error flag (FF)
  alias SPenO: STD_LOGIC is EDCi(2); -- to make life easier
  alias SPenl: STD_LOGIC is EDCi(6);

begin
  process(clk, rst)
  begin
    if rst = '1' then
      CNT <= SPCNTi;
      Idle <= '0'; Awr <= '0'; Bwr <= '0';
      Plwr <= '0'; SPerro <= '0'; Plwro <= '0';
      SPrst <= '0'; PIA <= x"00";
    elsif rising_edge(clk) then
      caspe CSP is
        when C1S1P1 | C2S1P1 =>
          if Plwri = '1' then
            if Awr = '1' then
              Bwr <= '1';
              Plwri <= not (Pli);
            else
              Awr <= '1';
              Plwri <= Pli;
            end if;
          end if;
        when C1S1P2 | C2S1P2 =>
          if PATH = "000" then
            SPrst <= '1';
          elsif PATH = "001" then
            CNT <= SPCNTi;
          elsif PATH = "010" then
            Idle <= '0';
          elsif PATH = "011" then
            SPrst <= '1';
            SPerro <= '1';
            SPrst <= '1';
          elsif PATH = "100" then
            CNT <= (CONV_STD_LOGIC_VECTOR((CONV_INTEGER(CNT(7 downto 0)) - 1), 8));
          elsif PATH = "101" then
            SPerro <= '1';
          end if;
    end if;
  end process;
end;
Techniques intended to reduce the impact of program-flow errors on embedded systems

SPrst <= '1';
elsif PATH = "110" then
  Plupd <= '1';
  SPrst <= '1';
end if;
when C1S2P1 | C2S2P1 =>
if SPrst = '1' then
  SPrst <= '0';
  CNT <= SPCNTi;
  Idle <= '1';
  Awr <= '0';
  Bwr <= '0';
end if;
if Plupd = '1' then
  Plupd <= '0';
  Plwr <= '1';
end if;
when C1S2P2 | C2S2P2 =>
  Plwr <= '0';
when C1S6P1 | C2S6P1 =>
  SPerr <= '0';
when others => null;
end case;
-- no processing, reset SP
-- update PI
-- reset SP
-- S2P1: resetting flags
-- reset SPrst flag
-- reload CNT
-- SP idle mode
-- reset PlA write flag
-- reset PlB write flag
if (SPenO = '0') and (SPen1 = '1') then
  SPen <= 11' ;
elsif (SPenO = '1') and (SPen1 = '0') then
  SPen <= '0';
end if;
end if;
end process control;

PATH <= '000' when SPen = '0' else
      '001' when SPen = '1' and Idle = '1' and Awr = '0' else
      '010' when SPen = '1' and Idle = '1' and Awr = '1' else
      '011' when SPen = '1' and Idle = '0' and Bwr = '0' and CNTzr = '1' else
      '100' when SPen = '1' and Idle = '0' and Bwr = '0' and CNTzr = '0' else
      '101' when SPen = '1' and Idle = '0' and Bwr = '1' and CMP = '1' else
      '110';

CMP <= '1' when PlA /= PlB else '0';
CNTzr <= '1' when CNT = "0000" else '0';
SPerro <= SPerr;
Plo <= PlA when SPen = 11 else Pli;
Plwro <= Plwr when SPen = 11 else Plwi;

Listing Q.10: VHDL source code for the SecurePorts module

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.te51pak.all;

entity pulse_stretch is
  port(
    clk: in STD_LOGIC;
    rst: in STD_LOGIC;
    csp: in te51stateT;
    WRi: in STD_LOGIC;
    DATAi: in STD_LOGIC_VECTOR(7 downto 0);
    WRo: out STD_LOGIC;
    DATAo: out STD_LOGIC_VECTOR(7 downto 0);
  );
end entity pulse_stretch;

architecture bhv of pulse_stretch is
  signal WR: STD_LOGIC;
  signal DATA: STD_LOGIC_VECTOR(7 downto 0);

Q-10
Techniques intended to reduce the impact of program-flow errors on embedded systems

```vhdl
signal stretch: STD_LOGIC;
signal CNT: STD_LOGIC_VECTOR(3 downto 0);

begin
process(clk, rst)
begin
  if rst= '1' then
    WR <= '0'; DATA <= x"00";
    stretch <= '0'; CNT <= x"b";
  elsif rising_edge(clk) then
    -- on rising clock edge
    if WRi = '1' then
      WR <= '1';
      DATA <= DATAi;
      stretch <= '1';
    end if;

    if stretch = '1' then
      if CNT = x"0" then
        CNT <= x"b";
        stretch <= '0';
        WR <= '0';
      else
        CNT <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(CNT(3 downto 0)) - 1), 4);
      end if;
    end if;
  end if;
end process;

WRo <= WR;
DATAo <= DATA;
end architecture bhv;
```

Listing Q.11: VHDL source code for the PULSE_STRETCH module
Appendix R  CD-ROM listing

The CD-ROM layout is presented in this appendix.

The abbreviation of the development tools used to develop the program/firmware or the hardware, shown in Table R.2 (in brackets), is described in Table R.1. Those without the abbreviations (e.g. CBG_HWFT_SP) are either complete programs (i.e. executable), or only given in the source file format (not as a complete project).

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Development program</th>
</tr>
</thead>
<tbody>
<tr>
<td>C51</td>
<td>Keil C51 v5.50 or v6.0</td>
</tr>
<tr>
<td>HDL</td>
<td>Aldec Active-HDL 4.2</td>
</tr>
<tr>
<td>OrCAD</td>
<td>OrCAD Capture for Windows 7.0</td>
</tr>
<tr>
<td>PCB</td>
<td>PADS PCB</td>
</tr>
<tr>
<td>VB</td>
<td>Microsoft Visual Basic 6.0</td>
</tr>
<tr>
<td>VC</td>
<td>Microsoft Visual C++ 6.0</td>
</tr>
<tr>
<td>XST</td>
<td>Xilinx ISE 4.1i</td>
</tr>
</tbody>
</table>

Table R.1: Abbreviations of development tools

The tools used in this project, as shown in Table R.2, are located in the ‘tools’ directory. The ‘8051Sim_xkB’ directory holds variants of 8051Sim and 8051Sim-NG with 4kB, 16kB, 32kB and 64kB (8051Sim-NG only) of physically implemented code memory (the default is 8kB). Please read the ‘readme_first.txt’ file for the correct program to use (especially for 8051sim-NG).

The MCU Testbench project comes in three formats: 1) on its own, 2) integrated with TE-51, and 3) integrated with TE-51 with error injection into an SP register (used in Chapter 18 only). All version interfaces with the same MCU Host. TE-51 is in the EDIF (Electronic Design Interchange Format) format.

HTPulse is a computer-controllable MCU-based variable duty-cycle/frequency high-voltage generator. This project was initially used to generate sparks as the source of EMI, mimicking the behaviour of a vehicle’s electrical distributor. It was not used in any experiment as the sparks generated were of low current, and the amount of noise generated was inconsistent and immeasurable (we lack suitable equipment).
Techniques intended to reduce the impact of program-flow errors on embedded systems

Espresso is a Boolean minimisation program developed by Berkeley University of California that can be downloaded from http://www.fke.utm.my/downloads/espresso/.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Sub-directory</th>
<th>No. of files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATS</td>
<td></td>
<td>17</td>
<td>Source files (VC)</td>
</tr>
<tr>
<td>8051sim</td>
<td>GUI</td>
<td>21</td>
<td>Graphical front end (VB)</td>
</tr>
<tr>
<td></td>
<td>8051DLL</td>
<td>22</td>
<td>Simulation engine (VC)</td>
</tr>
<tr>
<td>8051Sim_xkB</td>
<td></td>
<td>9</td>
<td>8051Sim-NG with various ROM sizes</td>
</tr>
<tr>
<td>TE-51</td>
<td></td>
<td>1</td>
<td>TE-51 only in EDIF format</td>
</tr>
<tr>
<td>MCUtestbench</td>
<td>Testbench_only</td>
<td>38</td>
<td>Testbench only (HDL/XST)</td>
</tr>
<tr>
<td></td>
<td>Testbench_TE-51</td>
<td>21</td>
<td>Testbench integrated with TE-51 (HDL/XST)</td>
</tr>
<tr>
<td></td>
<td>Testbench_TE-51_SP</td>
<td>23</td>
<td>Testbench for SP testing only (HDL/XST)</td>
</tr>
<tr>
<td></td>
<td>Host</td>
<td>21</td>
<td>Graphical front end (VB)</td>
</tr>
<tr>
<td>HTPulse</td>
<td>hardware\Power_supply</td>
<td>6</td>
<td>Design (OrCAD) and PCB files (PCB)</td>
</tr>
<tr>
<td></td>
<td>hardware\Pulse_generator</td>
<td>6</td>
<td>Design (OrCAD) and PCB files (PCB)</td>
</tr>
<tr>
<td></td>
<td>hardware\HT_output</td>
<td>6</td>
<td>Design (OrCAD) and PCB files (PCB)</td>
</tr>
<tr>
<td></td>
<td>hardware\PCB_layout</td>
<td>14</td>
<td>PCB layouts in GIF format</td>
</tr>
<tr>
<td>HTPulse</td>
<td></td>
<td>10</td>
<td>Firmware for Atmel AT89C2051 (C51)</td>
</tr>
<tr>
<td>PCLink</td>
<td></td>
<td>8</td>
<td>PC communication program files (VB)</td>
</tr>
<tr>
<td>Espresso</td>
<td></td>
<td>4</td>
<td>Program to reduce boolean expressions</td>
</tr>
</tbody>
</table>

Table R.2: CD-ROM layout for tools used in this project ('tools' directory)

Of the three sets of sources files shown in Table R.3, only T16 and ST2 come as complete projects. Only the relevant source files for the other modules are located on the CD-ROM (in the 'source' directory) since the complete project would have included the ‘intellectual property’ of Trenz Electronik GmbH (TE-51) in the source format.

<table>
<thead>
<tr>
<th>Directory</th>
<th>No. of files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T16</td>
<td>12</td>
<td>T16 source files (C51)</td>
</tr>
<tr>
<td>ST2</td>
<td>43</td>
<td>ST2 source files (HDL/XST)</td>
</tr>
<tr>
<td>TE-51_peripherals</td>
<td>11</td>
<td>TE-51 peripheral source files</td>
</tr>
<tr>
<td>CBG_HWFT_SP</td>
<td>9</td>
<td>CBG, HWFT and SP source files</td>
</tr>
</tbody>
</table>

Table R.3: Source files for peripheral modules ('source' directory)
Techniques intended to reduce the impact of program-flow errors on embedded systems

<table>
<thead>
<tr>
<th>Directory</th>
<th>No. of files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>40</td>
<td>Source files for Alarm (C51)</td>
</tr>
<tr>
<td>Krider</td>
<td>7</td>
<td>Source files for Krider (C51)</td>
</tr>
<tr>
<td>Prog</td>
<td>7</td>
<td>Source files for Prog (C51)</td>
</tr>
</tbody>
</table>

Table R.4: Source files for test programs (‘test_programs’ directory)

The ‘base’ test programs (i.e. without any software- or peripheral-based error detection and/or correct technique) are located in the ‘test_programs’ directory. These programs are listed in their respective subdirectories as shown in Table R.4.

<table>
<thead>
<tr>
<th>Directory</th>
<th>No. of files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF_FT</td>
<td>89</td>
<td>NF and FT's evaluation – Chapter 4</td>
</tr>
<tr>
<td>MIT</td>
<td>25</td>
<td>Predicting program-flow error outcome – Chapter 7</td>
</tr>
<tr>
<td>ST</td>
<td>13</td>
<td>ST's simulation – Chapter 9</td>
</tr>
<tr>
<td>ST2</td>
<td>13</td>
<td>ST2's simulation – Chapter 10</td>
</tr>
<tr>
<td>BCT</td>
<td>16</td>
<td>BCT's simulation – Chapter 11</td>
</tr>
<tr>
<td>CBG</td>
<td>61</td>
<td>CBG's simulation – Chapter 15</td>
</tr>
<tr>
<td>HWFT</td>
<td>41</td>
<td>HWFT's simulation – Chapter 16</td>
</tr>
</tbody>
</table>

Table R.5: Script and results of simulation experiments (‘simulations’ directory)

The simulation scripts, test program source files (for some simulations) and results are shown in Table R.5. All simulation experiments, apart from that carried out in Chapter 4, were carried out with 8051Sim-NG. NF and FT's evaluation was carried out with dScope.

<table>
<thead>
<tr>
<th>Directory</th>
<th>No. of files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBG</td>
<td>73</td>
<td>CBG's benchmarking – Chapter 18</td>
</tr>
<tr>
<td>HWFT</td>
<td>49</td>
<td>HWFT's benchmarking – Chapter 18</td>
</tr>
<tr>
<td>SP</td>
<td>49</td>
<td>SP's benchmarking – Chapter 18</td>
</tr>
<tr>
<td>CBG_HWFT_CBG</td>
<td>49</td>
<td>CBG, HWFT and SP's benchmarking – Chapter 18</td>
</tr>
</tbody>
</table>

Table R.6: Benchmarking results (‘benchmark’ directory)

The benchmarking script and results are located in the ‘benchmark’ directory.