Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand). This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.

- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.

- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. [http://researchspace.auckland.ac.nz/feedback](http://researchspace.auckland.ac.nz/feedback)

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form.
A Methodology for the Optimisation and Synthesis of Digital FPGA-Based Circuits

Richard Bruce Maunder

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Electrical and Electronic Engineering

Department of Electrical and Electronic Engineering University of Auckland New Zealand

February 2002
Torimakishi
yari no hayashi ni
iru toki wa
kotate wa onoga
kokoro tozo shire

When surrounded
by a forest
of spears,
know that you must use
your own mind as a shield.

Morihei Ueshiba
Abstract

This thesis examines the development and analysis of a methodology for designing and optimising digital FPGA-based circuits. The focus is on assisting the designer at a very high level—as either a pre-processor or as a specific solution design that can be reapplied. The study includes an investigation of related research, methods, and techniques.

Initially, a synthesiser is developed to produce hardware description language listings for stochastic neural networks. This highlights the advantages and disadvantages of special-purpose synthesis. In particular, the close hardware mapping of stochastic summation and sigmoidal activation functions is described.

The second synthesiser is developed for general purpose synthesis. It is based on synchronous sub-circuit modules and supports hierarchical designs. The modules are described in static libraries or in extensions to the synthesiser core that create the modules during synthesis. The modules can be of various data types including parallel, serial, and stochastic representations. The synthesiser supports hardware resource-sharing to trade-off cell count and cycle count. It analyses the circuits in terms of their pipelined data flow and operating latency. A separate module scheduler and controller circuit is synthesised as part of the synthesiser’s operation.

A genetic algorithm is tailored to the problem of digital circuit optimisation through the development of specific structures and procedures. In particular, a concise encoding of the circuit is developed that the genetic algorithm can manipulate. Specific crossover and mutation mechanisms are also developed to complement the functionality of the synthesiser. The searches are effected by altering module data type, hardware resource sharing, and module implementation version. A fitness function is derived that makes use of a number of optimisation parameters to objectively evaluate each particular circuit. The features of each circuit are calculated and estimated during the analysis phase. This includes an estimate of cell-count for a particular target technology, for which the concept of a hardware compression factor is introduced.

The developed software implementations are examined in terms of the software
development process. Various engineering issues are discussed along with details of the development strategies employed. The synthesis software is designed for robustness, low development time, and low debugging time. The cost is a less-than-optimal operating speed of the synthesis and optimisation code.

Two detailed case studies are examined to show the synthesis and optimisation performance. A formula is derived to calculate the quantitative extents of the design spaces being searched. The optimiser is shown to converge to ‘fit’ solutions—in terms of the desired constraints and the objective fitness function.

Clustering of solution points in the design spaces is observed. These are generally narrow in terms of cycle count and considerable in terms of cell count. Trends are apparent across the design space showing that module data type and resource sharing are correlated to cell-count and cycle-count. Different circuit optimisations produce different correlations such that an overall generalisation was not appropriate.
Acknowledgments

I am grateful to my supervisors, Dr George Coghill and Dr Zoran Salcic. I feel lucky to have had two such great academics and wonderful people watching over my research for its duration. They are both men that I have the utmost respect for.

Dr Coghill has provided a lot of technical assistance from his broad background in computational intelligence. He has also ensured that non-technical matters associated with my research were taken care of. Over the duration of my doctoral research he has become a very good friend and confidant. Lastly, I am very thankful for his being able to secure funding to allow me to attend international conferences and see first-hand the relevant research that is being carried out throughout the world.

Dr Salcic has similarly used his vast knowledge of the areas associated with field programmable hardware to provide me with a great deal of technical assistance. He has also been interested in making sure the non-technical aspects of the research were running smoothly and he has become a trusted friend and colleague.

I thank my fellow student, Matthew Savage, for his assistance with the development of a library of HDL modules compatible with the core HDL synthesiser. I would also like to acknowledge the considerable effort that Brian Mitchell put into proofreading the draft thesis manuscript.

I thank the University of Auckland for its financial support through a Doctoral Scholarship. I also thank the Connectionist Based Information Systems group at Otago University for its financial support.

Finally, I am grateful to my friends and family for supporting me throughout this research.

This thesis was prepared using the \LaTeX document preparation system.
Contents

List of Figures xi
List of Tables xv
Acronyms xvii

1 Introduction 1
  1.1 Research Objectives ........................................... 1
  1.2 Research Context .............................................. 2
    1.2.1 Current Synthesis Technology ............................. 3
  1.3 Major Research Achievements ................................ 5
  1.4 Background ...................................................... 6
  1.5 Thesis Organisation ............................................ 9

2 FPGA Technology and Design Methods 13
  2.1 Introduction .................................................... 13
  2.2 Field Programmable Logic ..................................... 13
  2.3 Altera FLEX 10K Device Family ............................... 15
    2.3.1 Altera Hardware Nomenclature ............................. 16
  2.4 FPGA Design Tools ............................................ 16
    2.4.1 Hardware Description Languages ........................... 17
    2.4.2 Graphical Design Tools ................................... 18
    2.4.3 Optimisation Tools ........................................ 19
    2.4.4 Simulation .................................................. 19
  2.5 Custom Configurable Computing ............................... 20
    2.5.1 Dynamic Circuit Reconfiguration .......................... 21
  2.6 High-Level Synthesis .......................................... 22
    2.6.1 Hardware Description Language Synthesis ................... 23
    2.6.2 Hardware Libraries ......................................... 23
2.6.3 Hardware/Software Co-Design ........................................... 24
2.7 Design Optimisation ......................................................... 25
  2.7.1 Variable-Width Data Paths ........................................... 26
  2.7.2 Hardware Evolution ..................................................... 27
2.8 Conclusions ........................................................................ 29

3 Stochastic Neural Network Synthesis ................................. 31
  3.1 Introduction ................................................................. 31
  3.2 Hardware Neural Network Implementations ......................... 32
    3.2.1 FPGA-Based Implementations ..................................... 33
  3.3 Stochastic Number Representation ..................................... 34
    3.3.1 Stochastic Number Generation ..................................... 35
    3.3.2 Random Number Generation ....................................... 35
  3.4 Stochastic Neural Networks ................................................ 40
  3.5 Synthesiser and Analysis Tool Overview .............................. 41
    3.5.1 Stochastic Neural Network Synthesiser ............................ 41
    3.5.2 Stochastic Neural Network Analysis Tool ....................... 42
  3.6 Special Purpose Synthesis Techniques ................................. 42
    3.6.1 Neuron Activation Function ......................................... 44
    3.6.2 Weight Generation .................................................... 47
    3.6.3 Weight Multiplication ................................................. 48
  3.7 Synthesised Neural Network Case Study ............................... 48
  3.8 Conclusions ..................................................................... 50

4 Synthesis Using Synchronous Modules ................................. 53
  4.1 Introduction ..................................................................... 53
    4.1.1 Design Entities ......................................................... 54
    4.1.2 Synthesiser Outline .................................................... 54
  4.2 Synchronous Modules ....................................................... 55
    4.2.1 Functional Modules ...................................................... 60
    4.2.2 Auxiliary Modules ....................................................... 60
  4.3 Synthesis Variables .......................................................... 61
    4.3.1 Module Implementation Version ..................................... 62
    4.3.2 Module Data Type ......................................................... 63
    4.3.3 Module Resource Sharing ............................................. 64
  4.4 Synthesis Algorithm .......................................................... 65
    4.4.1 Feedback Paths .......................................................... 67
4.4.2 Circuit Repair ............................................ 69
4.5 Dataflow Analysis ........................................... 71
4.5.1 Feedforward Analysis ................................. 74
4.5.2 Dataflow Ordering of Resource-Shared Modules .... 75
4.5.3 Analysis of Feedback Paths ......................... 76
4.5.4 Pipeline Analysis ........................................ 77
4.6 Module Generators ........................................ 81
4.6.1 Estimated Hardware Resource Requirements .... 82
4.7 Control Unit ................................................ 83
4.7.1 External Control Signals ........................... 87
4.8 Graphical Design Tool .................................... 88
4.9 Case Study .................................................. 90
4.10 Conclusions ............................................... 90

5 Circuit Optimisation ........................................... 95
5.1 Introduction ................................................ 95
5.2 Genetic Algorithm Overview ......................... 96
5.3 Redundancy Removal .................................... 96
5.4 High-Level Genetic Algorithm Optimisation ............. 99
  5.4.1 Problem Encoding ..................................... 99
  5.4.2 Genetic Algorithm Parameters ...................... 103
  5.4.3 Other Optimisation Techniques .................... 104
5.5 Fitness Function Derivation .............................. 104
  5.5.1 Fitness Scaling ........................................ 105
  5.5.2 Derivation of a Hardware Compression Factor (γ) .. 107
5.6 Selection and Reproduction .............................. 112
5.7 Specific Crossover Mechanism ......................... 113
5.8 Specific Mutation Mechanisms ......................... 114
  5.8.1 Module Group Mutation ............................. 116
  5.8.2 Group Data-Type Mutation ......................... 116
  5.8.3 Group Implementation-Version Mutation .......... 116
  5.8.4 Group Resource-Sharing Mutation .................. 117
5.9 Distributed Genetic Algorithm Processing ............... 117
5.10 Optimisation Algorithm .................................. 118
5.11 Conclusions ............................................... 118
6 Synthesiser Performance and Design-Space Analysis

6.1 Introduction .................................................. 121
6.2 Quantitative Design-Space Extent .......................... 122
  6.2.1 Module Grouping ...................................... 122
  6.2.2 Solutions for a Set of Functionally Equivalent Modules .... 123
  6.2.3 Solutions for Circuits with Functionally Different Modules .... 125
6.3 Case Study 1: Matrix Exponentiation Circuit ........ 126
  6.3.1 Circuit Topology and Operation .......................... 126
  6.3.2 Analysis Approach .................................... 129
  6.3.3 Optimisation Results .................................. 132
  6.3.4 Design Space ......................................... 138
6.4 Case Study 2: Artificial Neural Network Circuit .... 140
  6.4.1 Circuit Topology and Operation .......................... 141
  6.4.2 Analysis Approach .................................... 143
  6.4.3 Optimisation Results .................................. 146
  6.4.4 Design Space ......................................... 148
6.5 Optimisation Performance .................................. 149
  6.5.1 Determining Optimisation Parameters .................. 150
  6.5.2 Design-Space Fitness-Surface Transformation ........ 151
6.6 Design-Space Feature Analysis ............................ 152
  6.6.1 Macroscopic Features .................................. 152
  6.6.2 Microscopic Features .................................. 159
6.7 Conclusions ................................................. 162

7 Synthesiser Software Development and Implementation Discussion 163

7.1 Introduction .................................................. 163
7.2 Software Engineering Issues ............................... 164
  7.2.1 Software Engineering .................................. 164
  7.2.2 Software Design and Quality Management ............. 165
  7.2.3 Development-Team Issues ............................... 166
  7.2.4 Coding Issues ......................................... 166
7.3 Development Paradigms .................................... 167
  7.3.1 Portability Issues ...................................... 167
  7.3.2 Relational Database Engines ............................ 168
  7.3.3 Graphical versus Command Line Interfaces ............ 170
7.4 Development Tools .......................................... 171
7.5 The Stochastic Neural Network Synthesiser ............. 172
7.6 Evolution of the Synthesiser Source Code ................................. 173
  7.6.1 Synthesiser Debugging ............................................. 174
  7.6.2 Module Generators ................................................. 175
7.7 Distributed Genetic Algorithm Implementation .......................... 176
7.8 Conclusions ............................................................. 177

8 Conclusions ............................................................. 179
  8.1 Introduction .......................................................... 179
  8.2 Thesis Summary ....................................................... 179
  8.3 Discussion and Conclusions ......................................... 181
  8.4 Proposed Future Extensions ....................................... 184
    8.4.1 Suitable Types of Target Application ....................... 187
    8.4.2 Application to Dynamically Reconfigurable Circuits ...... 188

A Stochastic Neural Network Synthesiser Reference ...................... 191
  A.1 Synthesised AHDL Output .......................................... 191

B Synchronous-Module Based HDL Synthesiser Reference .................. 199
  B.1 Introduction .......................................................... 199
  B.2 Command Line Interface ............................................ 199
  B.3 Input Files .......................................................... 200
    B.3.1 Module Database and Interface Shell ......................... 200
    B.3.2 Circuit Netlist Description .................................. 200
    B.3.3 Genetic Algorithm Parameters ................................ 202
  B.4 Output Files ....................................................... 202
    B.4.1 Synthesised Modules ........................................... 202
    B.4.2 Main Circuit Design Listing .................................. 205
    B.4.3 Control Signal Generator Design Listing .................... 205
    B.4.4 Design Report File ............................................. 208
    B.4.5 Optimisation Results File .................................... 208
  B.5 DLL Synthesiser Extensions (Module Generators) ................... 211
  B.6 Distributed Genetic Algorithm Client Processes ................... 213
  B.7 Settings for the Altera Max+Plus II Environment ................... 214

C Circuit Module Grouping ................................................. 217
  C.1 Introduction .......................................................... 217
  C.2 Grouping of Modules into Two Groups ................................ 217
  C.3 Design Space Size Estimation Formula Derivation .................. 218
## List of Figures

1.1 Overall design process showing the context of this research.......................... 3

3.1 Uncorrelated pseudo-random bit streams from a single LFSR .................. 37
3.2 Hierarchical structure of synthesised stochastic neural networks ............ 43
3.3 Neural network topology for XOR example network ............................... 44
3.4 Continuous sigmoidal activation function ........................................... 45
3.5 Circuit for the generation of a weighted stochastic bit streams ............... 48
3.6 On-chip simulation results for the XOR neural network .......................... 50

4.1 Design entities and abstractions used in circuit synthesis ......................... 55
4.2 Global view of the synthesis flow ................................................... 55
4.3 Synchronous module connections .................................................... 57
4.4 Overview of the synchronous-module based synthesis ............................. 58
4.5 Different data type representations .................................................. 63
4.6 Synchronous module resource sharing .............................................. 65
4.7 The need for defining feedback paths ............................................... 69
4.8 Circuit repair through module insertions .......................................... 70
4.9 Reservation table representation of collision avoidance ......................... 73
4.10 Collision vectors, reservation tables, and forbidden latency sets .............. 74
4.11 Effect of feedback on dataflow ..................................................... 78
4.12 Modified state diagram example .................................................... 80
4.13 Logic-cell usage vs accuracy for generated multiplier modules ............... 84
4.14 The determination of module control signals ..................................... 86
4.15 Hierarchical external control-line descriptions .................................... 89
4.16 Graphical design-tool screenshots .................................................. 89
4.17 Pixel averaging filter mask and operation ........................................ 91
4.18 Pixel averaging filter circuit .......................................................... 91
4.19 Pixel averaging filter timing diagram .............................................. 92

5.1 Recursive circuits exhibiting removable redundancy .............................. 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Chromosome-encoding circuit example</td>
<td>101</td>
</tr>
<tr>
<td>5.3</td>
<td>Genetic algorithm encoded circuit representation</td>
<td>102</td>
</tr>
<tr>
<td>5.4</td>
<td>Effect of hard limits and penalties on the fitness function</td>
<td>106</td>
</tr>
<tr>
<td>5.5</td>
<td>Estimated versus actual logic cell usage</td>
<td>108</td>
</tr>
<tr>
<td>5.6</td>
<td>Logic cell estimation error versus compression factor</td>
<td>112</td>
</tr>
<tr>
<td>5.7</td>
<td>Visual circuit abstraction of an arbitrary circuit</td>
<td>114</td>
</tr>
<tr>
<td>5.8</td>
<td>Crossover demonstration</td>
<td>115</td>
</tr>
<tr>
<td>6.1</td>
<td>Matrix exponentiation circuit's functional structure</td>
<td>128</td>
</tr>
<tr>
<td>6.2</td>
<td>Initial design space extent and target solutions for matrix circuit</td>
<td>131</td>
</tr>
<tr>
<td>6.3</td>
<td>Fitness optimisation curves for matrix circuit goal 1, trial a</td>
<td>133</td>
</tr>
<tr>
<td>6.4</td>
<td>Fitness optimisation surface for matrix circuit goal 1, trial a</td>
<td>134</td>
</tr>
<tr>
<td>6.5</td>
<td>Cell-count optimisation surface for matrix circuit goal 1, trial a</td>
<td>134</td>
</tr>
<tr>
<td>6.6</td>
<td>Cycle-count optimisation surface for matrix circuit goal 1, trial a</td>
<td>134</td>
</tr>
<tr>
<td>6.7</td>
<td>Fitness optimisation surface for matrix circuit goal 1, trial b</td>
<td>135</td>
</tr>
<tr>
<td>6.8</td>
<td>Fitness optimisation surface for matrix circuit goal 2, trial a</td>
<td>135</td>
</tr>
<tr>
<td>6.9</td>
<td>Cell-count optimisation surface for matrix circuit goal 2, trial a</td>
<td>136</td>
</tr>
<tr>
<td>6.10</td>
<td>Cycle-count optimisation surface for matrix circuit goal 2, trial a</td>
<td>137</td>
</tr>
<tr>
<td>6.11</td>
<td>Fitness optimisation surface for matrix circuit goal 3, trial a</td>
<td>138</td>
</tr>
<tr>
<td>6.12</td>
<td>Cell-count optimisation surface for matrix circuit goal 3, trial b</td>
<td>139</td>
</tr>
<tr>
<td>6.13</td>
<td>Cell-count optimisation surface for matrix circuit goal 3, trial c</td>
<td>139</td>
</tr>
<tr>
<td>6.14</td>
<td>Exploded design space for matrix circuit goal 1, trial a</td>
<td>140</td>
</tr>
<tr>
<td>6.15</td>
<td>Artificial neural network topology as output from SNNS</td>
<td>142</td>
</tr>
<tr>
<td>6.16</td>
<td>Fitness optimisation surface for neural network circuit, trial a</td>
<td>147</td>
</tr>
<tr>
<td>6.17</td>
<td>Fitness optimisation surface for neural network circuit, trial b</td>
<td>147</td>
</tr>
<tr>
<td>6.18</td>
<td>Fitness optimisation surface for neural network circuit, trial c</td>
<td>148</td>
</tr>
<tr>
<td>6.19</td>
<td>Fitness optimisation surface for neural network circuit, trial d</td>
<td>149</td>
</tr>
<tr>
<td>6.20</td>
<td>Exploded design space for matrix circuit goal 1, trial b</td>
<td>153</td>
</tr>
<tr>
<td>6.21</td>
<td>Exploded design space for matrix circuit goal 1, trial c</td>
<td>153</td>
</tr>
<tr>
<td>6.22</td>
<td>Exploded design space for matrix circuit goal 2, trial a</td>
<td>154</td>
</tr>
<tr>
<td>6.23</td>
<td>Exploded design space for matrix circuit goal 3, trial a</td>
<td>154</td>
</tr>
<tr>
<td>6.24</td>
<td>Exploded design space for neural network circuit, trial a</td>
<td>155</td>
</tr>
<tr>
<td>6.25</td>
<td>Exploded design space for neural network circuit, trial b</td>
<td>155</td>
</tr>
<tr>
<td>6.26</td>
<td>Exploded design space for neural network circuit, trial c</td>
<td>156</td>
</tr>
<tr>
<td>6.27</td>
<td>Exploded design space for neural network circuit, trial d</td>
<td>156</td>
</tr>
<tr>
<td>6.28</td>
<td>Solution repetitions for matrix circuit goal 3, trial a</td>
<td>157</td>
</tr>
<tr>
<td>6.29</td>
<td>Spread of module data-type vs cycle-count across a design space</td>
<td>158</td>
</tr>
</tbody>
</table>
6.30 Spread of module-sharing vs cell-count across a design space . . . . . . 159
List of Tables

3.1 8-bit bipolar stochastic stream representations of real values ........................................... 34
3.2 LFSR feed-off taps for delayed versions of the PRBS .......................................................... 40
3.3 Process overview of SNN synthesis and verification ............................................................. 42
3.4 Input file describing the XOR neural network ....................................................................... 43
3.5 Truth table for the bipolar XOR function .............................................................................. 49

4.1 Synchronous synthesis algorithm outline .............................................................................. 56
4.2 Parameter section of the module interface shell file .............................................................. 59
4.3 Effects of modifying synthesis variables .............................................................................. 62
4.4 Resource sharing in a parallel circuit ..................................................................................... 66
4.5 Resource sharing in a serial circuit ......................................................................................... 66
4.6 Synchronous synthesis algorithm ........................................................................................ 68
4.7 Important dataflow and pipeline terms .................................................................................. 72
4.8 Algorithm for generating modified state diagrams ............................................................... 79
4.9 Algorithm for finding all greedy cycles ............................................................................... 81
4.10 Subset of modules defined in the module generators ............................................................ 83
4.11 Generated multiplier-module logic cell usage ...................................................................... 84
4.12 Derived formulae for generated multiplier modules ............................................................ 84
4.13 Results for various pixel-averaging circuit implementations .............................................. 92
4.14 Effect of circuit implementation with different divider circuitry ......................................... 93

5.1 Redundancy removal algorithm ............................................................................................. 98
5.2 Information contained in chromosome ‘head’ ....................................................................... 101
5.3 Information contained in chromosome ‘tail’ ......................................................................... 102
5.4 Resource usage for memory-cell based parallel lower-half multipliers ................................. 110
5.5 Resource usage for memory-cell-based parallel full-width multipliers ................................ 111
5.6 Complete optimisation algorithm ......................................................................................... 119

6.1 Different ways of grouping modules in a circuit ................................................................. 123
6.2 Number of estimated solutions for circuits of functionally equivalent modules ................................. 125
6.3 Salient results for matrix exponentiation circuit resource usage ....................................................... 130
6.4 Matrix exponentiation circuit results for optimisation goal 1 ....................................................... 132
6.5 Matrix exponentiation circuit results for optimisation goal 2 ....................................................... 136
6.6 Matrix exponentiation circuit results for optimisation goal 3 ....................................................... 138
6.7 The effect of different neuron activation function resolutions ..................................................... 144
6.8 Salient results for artificial neural network circuit resource usage .................................................. 145
6.9 Optimisation parameters and results for artificial neural network circuit ...................................... 146
6.10 Best-circuit features for matrix circuit optimisation goal 2, trial b .............................................. 160
6.11 Best-circuit features for matrix circuit optimisation goal 3, trial c .............................................. 160
6.12 Best-circuit features for neural network optimisation trial c ....................................................... 161
6.13 Best-circuit features for neural network optimisation trial d ....................................................... 161

A.1 Synthesised random number generator ..................................................................................... 192
A.2 Synthesised random bit-stream modulator ................................................................................ 193
A.3 Synthesised stochastic neural network ...................................................................................... 195
A.4 Synthesised hidden stochastic neuron ....................................................................................... 196
A.5 Synthesised stochastic synapse .................................................................................................... 197
A.6 Synthesised stochastic neuron bias ............................................................................................. 198

B.1 BNF syntax of the module interface shell .................................................................................. 201
B.2 BNF syntax of the circuit netlist file .......................................................................................... 202
B.3 Example circuit netlist file ........................................................................................................ 203
B.4 BNF syntax of the genetic algorithm parameter file ............................................................... 204
B.5 Example genetic algorithm parameter file ................................................................................ 204
B.6 Synthesis output-directory listing ............................................................................................ 205
B.7 Synthesised AHDL multiplier module ....................................................................................... 206
B.8 Synthesised main AHDL circuit ................................................................................................ 207
B.9 Synthesised controller circuit ...................................................................................................... 209
B.10 Generated synthesis report file ................................................................................................ 210
B.11 GA-optimisation results file ...................................................................................................... 212
B.12 Settings for the Altera Max+Plus II compiler ......................................................................... 215

C.1 Circuit module combinations $n^m$, $S_A(m, n)$, and $S_E(m, n)$, $2 \leq m \leq 5$ ............................. 220
C.2 Circuit module combinations $n^m$, $S_A(m, n)$, and $S_E(m, n)$, $6 \leq m \leq 9$ ............................. 221
Acronyms

AHDL  Altera Hardware Description Language
ANN  Artificial Neural Network
APEX  Advanced Programmable Embedded Matrix
API  Application Programming Interface
ASIC  Application Specific Integrated Circuit
BNF  Backus-Naur Form
CLI  Command Line Interface
CPU  Central Processing Unit
DFF  D-type Flip Flop
DLL  Dynamic Linked Library
EAB  Embedded Array Block
FLEX  Flexible Logic Element matriX
FLS  Forbidden Latency Set
FPGA  Field Programmable Gate Array
FPIC  Field Programmable InterConnect
FPLD  Field Programmable Logic Device
GA  Genetic Algorithm
GUI  Graphical User Interface
HDL  Hardware Description Language
HW/SW  Hardware/Software
KLOC  Kilo-Lines Of Code
LAB  Logic Array Block
LAN  Local Area Network
LC  Logic Cell
LE  Logic Element
LFSR  Linear-Feedback Shift Register
LUT  LookUp Table
MAL  Minimum Achievable Latency
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLP</td>
<td>Multi-Layer Perceptron</td>
</tr>
<tr>
<td>MPG</td>
<td>Mask Programmable Gate Array</td>
</tr>
<tr>
<td>MSI</td>
<td>Medium-Scale Integration</td>
</tr>
<tr>
<td>PRBS</td>
<td>Pseudo-Random Binary Sequence</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access (read/write) Memory</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SNN</td>
<td>Stochastic Neural Network</td>
</tr>
<tr>
<td>SNNS</td>
<td>Stuttgart Neural Network Simulator</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static RAM</td>
</tr>
<tr>
<td>SSI</td>
<td>Small-Scale Integration</td>
</tr>
<tr>
<td>TDF</td>
<td>Text Design File</td>
</tr>
<tr>
<td>VCL</td>
<td>Visual Component Library</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>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>XOR</td>
<td>eXclusive OR (⊕)</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Research Objectives

The paramount objective of this thesis is to present a methodology that assists digital circuit designers in the design and optimisation of their circuits. This methodology is based on the automatic synthesis of hardware description language (HDL) circuit descriptions. The resulting HDL listing is optimised for given constraints and design criteria.

The first objective in developing an overall methodology is the development and presentation of a synthesis application. One that is capable of producing a complete HDL listing that fully specifies a target FPGA circuit and is able to be unambiguously compiled into a configuration bit stream.

Two different synthesisers are presented. The first is for application specific circuit synthesis that targets stochastic neural network implementations. This synthesiser incorporates techniques that are only applicable to such target circuits. The second synthesiser is for general-purpose synthesis. It targets the broad range of FPGA circuits that are data-path oriented.

Another important research objective is the development and presentation of the optimisation process—in particular a genetic algorithm (GA) that works in conjunction with the general-purpose circuit synthesiser. This involves deriving an encoding mechanism for the representation of a circuit in a form that may be manipulated easily by the GA. Other problem-specific GA mechanisms (crossover and mutation) are developed that enable the GA to efficiently process the circuit representations. The performance of the optimiser is analysed to ensure that it is behaving in the desired manner.

The final objective is the analysis of the resulting design space for each circuit. Each circuit's design space shows many different implementation versions of a functionally
equivalent circuit. The aim of this analysis is to determine if there are general features
that can be extracted to assist designers when they are performing manual design.
In particular, the use of different data representations (such as serial or parallel) in
different parts of a circuit is analysed. The analysis is performed in two distinct parts:
macroscopic and microscopic. Each highlights different levels of interaction between
sub-circuit component implementations.

The research is focussed on, but is not limited to, FPGA based circuits. The result-
ing design methodology is applicable to other design areas, such as VLSI. Application
specific integrated circuits can be produced from hardware description language list-
ings, allowing the employment of the general-purpose synthesiser's core. Moreover, the
GA optimiser will work in conjunction with the synthesiser regardless of the style of
target HDL that is implemented. Importantly, the model allows software modules and
components to be incorporated in conjunction with hardware modules.

This thesis is based in part on previously published results and findings. References
to the author's publications in relation to this thesis are given in the bibliography[54,
55, 56, 57, 76, 77].

1.2 Research Context

The synthesis and optimisation methodology presented in this thesis is performed at a
very high level of abstraction. It is not only a method whose output can be applied to
a given circuit design problem, but features that are produced by the method can be
reused by a designer. In fact, one of the main intentions of the algorithms is to provide
mentoring support to human designers.

The synthesis algorithm makes use of predefined HDL modules, and parameterised
module generators. The algorithm chooses implementations from among these modules.
In this way, advances in module design techniques and algorithms can be directly
applied to the methodology. Moreover, because the synthesis HDL output is still at
a high level, the methodology takes advantage of advances in lower-level synthesis
technologies—including those in commercial synthesis tools.

One of the methodology's main departures from conventional high-level synthesis
is the inclusion of diverse data types. In addition to varying degrees of parallelism in
the inter-module buses, the algorithms automatically provide for stochastic bit-streams
(including the associated random number generation-and-distribution hardware).

Figure 1.1 provides a graphical summary of the context of this research as part of
the whole synthesis-oriented design process.
1.2 Research Context

Figure 1.1: Different stages of the design process showing the context of this research.

1.2.1 Current Synthesis Technology

This section describes some of the current and recent research in the area, and how it relates to the methodology presented in this thesis.

Leading high level synthesis research is not only to be found in the realm of academia, but also in the commercial domain by companies such as Celoxica\footnote{http://www.celoxica.com}. Their design suite is built around Handel-C, a sequential programming language based on ISO/ANSI-C with a simple expression of parallelism. Software methodologies such as debugging and libraries are provided. This method differs from our methodology because it does not have an intermediate HDL step, which can build a designer's knowledge base. In addition, broad data-type trade-offs are not provided for.

Marinescu and Rinard\cite{marinescu}, and Stone and Manolacos\cite{stone} present synthesis methodologies similar to our approach, but without the data-type trade-offs and design-space search. Their approaches do, however, have aspects that are not contained in our approach.

McFarland\textit{ et al.}\cite{mcfarland} provide a comprehensive overview of high-level synthesis. Much of the background given is based around scheduling algorithms. They use the term\textit{ self-documentation} to describe how design decisions made by a synthesiser, and their effects, can be automatically documented. The extension of this is the idea that we use, that a designer can learn from analysing these design decisions, and apply the knowledge in future designs.

Santiago and Deharbe\cite{santiago} combine formal verification methods with high-level synthesis to produce an integrated design flow. The core of their research is a translator that takes a synchronous subset of SMV (a symbolic model checker) and produces VHDL-RTL\cite{vhdl-rtl} (a formally defined subset of VHDL used for register transfer level
A method to compute all optimal trade-off points in a design space is presented by Blythe and Walker[9]. These are used to completely characterise a design space. Careful pruning of sub-optimal solutions at each stage in the design cycle is the key to their method, which also uses both time and resource scheduling. The method includes clock-length determination, but ignores the effects of a controller, registers, and interconnects.

The aim of Bruni et al. [12] is also to characterise a circuit’s design space in an unbiased manner. This is done as a pre-synthesis analysis stage to determine the bounds of the design space using Monte-Carlo techniques. They show that the results can be used to help subsequent optimisations escape from local minima. Such a technique would have application in conjunction with our approach.

Fang and Wu[23] concentrate on exploiting design structural hierarchy for the automated partitioning of designs across multiple FPGA and FPLC devices. Their synthesis approach is integrated with the partitioning algorithm. The synthesiser input is a design specification in Verilog, and the outputs are multi-FPGA partitions in XNF (Xilinx Netlist Format) and Verilog. This type of approach is complementary to our methodology and could be applied to pre-processing or post-processing.

A genetic algorithm is used by Ascia et al. [5] to optimise the configuration of highly-parameterised designs. Their targeted design space is formed by considering different bus sizes and software algorithms, and noting the effects these have on hardware resource area, power consumption, and performance. The specific parameterised systems considered are hierarchical memories (including caches) where mean access time is of primary concern.

The approach taken by Potkonjak and Wolf[70] is to iteratively apply both operating-system and behavioural-synthesis techniques to derive and refine the final solution. This is based on their novel approach for heuristic optimisation at the task-sharing level—that is, the efficient sharing of application-specific hardware by several computational tasks. This is likened to the scheduling of periodic, independent tasks in a multitasking operating system. Their optimisation stage uses a branch-and-bound search that iterates from a coarse, fast estimation of the design parameters, to a slower, more accurate estimation. There is some overlap between this method and our approach, but the application to multiple periodic tasks (with hard timing requirements) means the method's relevance lies more with processor-oriented designs. Such designs—where computationally feasible—may have traditionally been implemented in software.
1.3 Major Research Achievements

The major achievements of this study are summarised below:

Achievements

- A novel methodology is presented for high-level design optimisation. This methodology is focussed upon assisting designers in high-level design decisions—before further synthesis and device-fitting implementation stages are performed. Importantly, advances in the supporting design algorithms and methods are exploitable, as this study complements existing design and synthesis tools.

- Design decisions such as the use of stochastic, bit-serial, or bit-parallel numbers, resource sharing, and hardware/software algorithm implementation, are those that the methodology assists the designer in making. The inherent non-linear effects that these design decisions have on the resulting circuit are accounted for in the algorithm.

- The resulting synthesis output is at a very high level, such that inventive parts of the optimised circuits can be reused and incorporated into a human designer’s repertoire.

- The developed method is realised in a scalable implementation. The derivations of the Genetic Algorithm details (fitness function and genetic operators) are presented. Existing technologies (such as pipeline analysis and control, and random number generation) are incorporated in the development method.

- A metric is derived to estimate the effect (for a specific FPGA family) of subsequent optimisations performed by Altera’s compiler and toolkit. This is important for the initial appraisal of a design’s fitness.

Experimental Results

- The method was verified through software implementation. The correctness of the synthesised HDL output was confirmed through subsequent synthesis and simulation. The optimisation algorithm’s convergence was checked via evolutionary trials and subsequent plotting of the observed design space.

- The design-space showed the expected trade-offs between hardware resource usage and circuit operation speed (compute time and average latency).
1.4 Background

This thesis presents research that is founded on well known techniques, but which are employed in unique combinations. Furthermore, it makes use of technologies that have not traditionally been available. The combination of techniques includes the application of sufficient computing power to the established problem of optimisation while making use of programmable logic, pipelining techniques, Genetic Algorithms, and stochastic systems. The optimisation task is extended to form a complete design methodology by considering the automatic synthesis and implementation of the resulting optimised circuits. The resulting optimisation results are analysed with the hope of gaining insight into the processes at work, even though the circuit optimisation and synthesis is designed to be an automatic procedure. This section presents a historical context for the research and provides some general references.

Programmable Logic

Although the topic is covered in more depth in chapter 2, it is important to note here a few of the features of programmable logic that enable this research to be performed. FPGA technology is dense enough that full microprocessor circuitry can be implemented in a single programmable device. Moreover, they can be operated at high enough frequencies (>100MHz) to be useful in actual applications, not just as a rapid prototyping tools.

FPGAs support a wide variety of development tools, but the ubiquitous nature of hardware description languages allows generic synthesis tools, such as those presented in this thesis, to be useful to designers. Synthesised sub-circuit listings can be reused and modified by designers. The HDLs that are most suitable for reuse are those with a wide base of adherents—such as VHDL, AHDL\textsuperscript{2}, and Verilog.

As it is with all engineering technologies, much effort goes into the optimisation of FPGA based circuits. It is evident from any FPGA academic journal or technical conference that there are many techniques for circuit optimisation

Pipelining Techniques

Pipelining has been used as a form of computational parallelism for as long as modern computers have been commonplace, with implementations such as Cotton’s 1965 high-speed pipeline circuit\cite{16}. Davidson extended pipeline theory in the early 1970s for efficiently utilising pipelines in the processing of data\cite{18, 68}. As with their application

\textsuperscript{2}Altera Hardware Description Language.
to VLSI and computer architecture, pipeline techniques are equally applicable to FPGA based circuits.

Each pipeline stage processes data in parallel with the other stages. A computational task can be initiated while prior tasks are still being processed within the pipeline. Processing multiple tasks in the pipeline reduces the average time taken for each computational task. The drawback is latency, and longer pipelines generally mean that the first result is not available as quickly as it would have been with a non-pipelined system. The number of stages in a pipeline cannot be arbitrarily large, but is limited by the number of integral sub-functions that a function can be split into[66].

Kogge gives a thorough mathematically-based coverage of pipelined computer architecture in his book[44]. Both Lewin[46] and Stone[90] give similar but less mathematical introductions to pipelining. These are given in the context of actual hardware implementation versus a more theoretical approach.

Genetic Algorithms

Genetic algorithms are familiar tools to most engineers and computer scientists. Goldberg's book[28] presents a broad yet deep coverage of the history and foundations of genetic algorithms and related optimisation techniques for those unfamiliar with the area. Similarly, Fogel's in-depth 1994 article[24] covers the prior 35 years of development of GAs and associated techniques such as evolution strategies and evolutionary programming. GAs are ideal for solving "NP-complete" problems and those where there are no less-expensive problem-specific methods[27]. GA optimisation is discussed in more detail in chapter 5.

Formal adaptive design methods have been developed for the application of genetic algorithms[52] because their use is so widespread. Although not directly related, the close ties with hardware are exemplified by the implementation of GA processors in hardware[31]. Parallel hardware is used to provide increased processing throughput for the inherently parallel algorithms.

Genetic algorithms have been successfully used to evolve hardware circuits entirely, and at a very low-level. A representation of the circuit is encoded in a form that a GA can manipulate (sometimes simply the FPGA configuration bit-stream[83]). As with other GA optimisation, the performance of each individual circuit is analysed before genetic operators are applied. There is no need for redundant constraints on the design, which can be implicit in designers' preconceptions of what circuit structures should be

---

3 Problems that cannot be solved in polynomial time.

4 More intriguing is Adleman's implementation of a GA in actual DNA[1].
like[93].

Thompson presents the genetic evolution of digital oscillator circuits purely guided by the accuracy of their oscillator outputs[92]. This was initially performed with a simulation of ideal, noise-free digital hardware. Thompson also presented the evolutionary results where the circuit was simulated on an actual device. This is known as *intrinsically* hardware evolution and makes use of real semiconductor physics. If a single device is used for the simulation, spurious aspects—such as stray line inductance and capacitance—of the device can be utilised by the GA. This makes the analysis and operational explanation by a human difficult.

In addition to the low-level evolution of a hardware design, GAs are also used to assist designers and are included internally in design tools. One example of assistance that can be provided is with interconnection routing optimisation[47]. It can be seen that there are many levels of design abstraction where GAs are useful aids to a hardware designer. Most of the individual circuits in the design-space being searched with intrinsic hardware evolution do not work. It is one of the main advantages of genetic optimisation at a higher level, that each individual circuit’s performance can be guaranteed.

Genetic evolution of computer programs is generally known as genetic programming. This technique manipulates a genetic representation of a computer program. Through crossover and mutation, and subsequent running of each program, an algorithm can be evolved that performs the desired function[45]. The application of genetic programming to hardware description languages moves it into the arena of hardware optimisation. Each individual HDL solution can be simulated on a computer and thus its performance can be determined. Research has shown that this method is a viable option for producing HDL code[83].

**Stochastic Systems**

The use of sub-circuitry that employs stochastic number representation allows a great trade-off of processing time and hardware real-estate. The use of stochastic number streams requires the implementation of random, or almost random, streams of binary data. This study relies heavily upon existing mathematical and electronics research into the efficient generation of pseudo-random binary sequences.

Historically, random numbers have been important for simulation of natural phenomena, sampling, numerical analysis, testing computer algorithms, unbiased decision making, telecommunications, and recreation. There is such a wide area of application that the generation of random number sequences has become a comprehensive sci-
ence. Initially, people that required random numbers used techniques such as throwing dice or drawing balls from a barrel. In 1927, a book of 40,000 random digits taken from census surveys was published. Subsequently, mechanical means were developed to produce random numbers. The first machine-produced table was of 100,000 digits in 1939[42]. Soon after computers were introduced, algorithms were developed that would efficiently produce the required random number sequences. It should be noted that there are many algorithms that appear to produce random numbers, but closer examination reveals inadequacies. These can fail by falling into a cycle, or converging to a single value (such as with ‘Algorithm K’)[42].

The generation of random number sequences in hardware has been generally focused on sequences of ones and zeros produced from correctly configured shift-registers. The resulting sequences can have an arbitrarily large repeat cycle, and their properties are well known[2, 99]. The application of these sequences ranges from neural networks[96] to telecommunications[86].

Summary

Pipelining techniques, Genetic Algorithms and stochastic systems are combined to form the basis of the presented FPGA design method. The GA uses a non-deterministic ‘fitness’ of each individual circuit that is computed via an in-depth analysis of the circuit’s pipelined data path. The issues of data representation within the circuit are also addressed during the fitness calculation. The allowable representations have been extended from the common parallel and serial formats to stochastic bit stream representations.

The remainder of this thesis describes the development of the design and optimisation method, and the results that were obtained.

1.5 Thesis Organisation

This thesis is presented in 8 chapters and supplementary appendices. Following the introductory chapter, chapter 2 covers existing field programmable logic technologies and design methods. The major contributions of this thesis appear in chapters 3 through 6, with chapter 7 providing a discussion of the software development aspects of the research. These chapters are followed by the thesis conclusions. The appendices provide technical information on the developed software.

Chapter 2 introduces FPGA technology and associated design methods. An introduction to field programmable logic is presented, along with an overview of con-
figurable computing. Aspects of FPGAs are described that make them suitable for specific tasks; such as their dynamic reconfigurability, their high degree of parallelism, and their reprogrammability. Of major importance is their role in rapid design prototyping. This chapter also discusses design tools for use with programmable logic. These include hardware description languages, graphical design tools, and tools associated with the optimisation of designs. In addition to standard hardware description languages, there are a number of special-purpose languages and compilers presented. Lastly, this chapter discusses the automatic synthesis of HDL listings from high level descriptions. These descriptions are often in higher-level, application-specific, description languages.

Chapter 3 presents a high level tool that assists in the creation of stochastic neural networks for implementation on FPGAs. The tool includes an automatic neural network synthesiser and utilities for assisting in the analysis of the resulting design. The neural network synthesis tool presented herein takes a weight matrix description of a network’s neural interconnections and produces a complete AHDL description that can be compiled for one or more Altera devices. The motivation for this research is to give an example of how the details of FPGAs can be hidden from a designer, allowing designs to be specified at a high level. It is important that the tool produces not only a solution, but that the method used in the solution both supports and exploits the features of programmable hardware. Special-purpose, problem-specific synthesis techniques were included in the neural network synthesiser—in particular the significant aspects of interconnection routing and activation function computation.

Chapter 4 presents an approach to high-level synthesis of digital circuits using synchronous modules. The result is a general purpose synthesiser for the development of a much broader range of circuits. The implemented synthesiser takes as its input a functional description of the circuit in the form of a netlist using predefined functional modules with desired parameters, and produces an AHDL description as an intermediate circuit representation. The functional modules can be designs entered and produced using different design entry tools and HDLs. The synthesis allows hardware resource sharing, variable data-path widths, variable bit resolutions, and various number representations (e.g. parallel, serial, stochastic) for different parts of a circuit. During synthesis, pipelined circuit analysis ensures coherent dataflow through the circuit is produced. At the end, the overall control unit that controls data flow through the circuit is automatically generated. The synthesiser is the first part of the implementation of a tool for the optimisation
of circuit designs with FPGAs as a target technology.

Chapter 5 presents a novel approach to optimising high level designs for circuits to be implemented on FPGAs. The aim is to search circuit design spaces using a Genetic Algorithm (GA) to find solutions that optimise circuit speed and circuit size under given constraints. The optimisation algorithm is designed to work with the synchronous-module based synthesiser presented in chapter 4. A genetic encoding for circuits is specified that can be manipulated by the GA. Specific genetic crossover and mutation mechanisms are presented that perform this manipulation by altering various aspects of parts of the circuit including number representation, the amount of hardware resource sharing, and the implementation versions or algorithms used. A formula for the calculation of the quality or fitness of a circuit is also derived, along with a mechanism for scaling the fitness to avoid the unwanted artefacts of evolution: population stagnation or becoming overwhelmed by super-individuals. The GA was implemented in software such that it could be run in a distributed manner, on multiple computers in parallel, over a local-area network.

Chapter 6 presents an analysis of the performance of the synchronous-module based synthesis and optimisation routines. As the GA searches a design space for an optimal solution, many designs are encountered and considered. The structure of this design space is analysed from a macroscopic perspective—looking at the groupings of designs within the multidimensional search space. Correlations are drawn between the features of the circuits and their groupings. The features considered are module data-type and resource sharing. The design space is also considered from a microscopic perspective. Here, the relative positions of designs within each group in the design space are considered. Again, these relative positions are correlated with the features of the circuits comprising the group. The performance of the optimiser is also investigated, and circuit case-studies are used in this analysis.

Chapter 7 is a discussion of software development focussing on the procedures and methods used in the writing of the synthesiser and optimisation code. The progressive development of various implementations of the synthesiser is detailed and discussed. Software development paradigms that were applicable to the design and implementation of the source code in this thesis are also commented upon. Code portability, relational database usage, choice of development tools, and the appropriateness of language extensions are discussed as they relate to the maturing discipline of software engineering.
Chapter 8 provides a summary of the thesis. The conclusions drawn in each chapter are presented and discussed. In addition, overall conclusions are drawn and presented. Future extensions to the work are also suggested.

Appendix A provides a guide for the stochastic neural network synthesiser. It describes the command line interface and the format for the input files that describe the neural network. Example output files are given that show the synthesised AHDL.

Appendix B provides a manual for the use and operation of the synchronous-module hardware description language synthesiser and optimiser software. It provides example input and output files, and describes the command line interface. The exported-function interface to the DLL synthesiser extension modules is given to allow further enhancement of the synthesiser.

Appendix C provides a comparison of methods for estimating the number of module combinations comprising a circuit. This is associated with the synthesiser of chapter 4 and compares formulae derived and discussed in chapter 6.
Chapter 2

FPGA Technology and Design Methods

2.1 Introduction

This chapter introduces Field Programmable Logic (FPL), and presents some of the technologies that are discussed and used in this thesis. In addition, background material is given to provide the foundation that the methodologies developed in this research build upon. The arena of programmable logic is immense, and so only relevant aspects will be focussed upon.\footnote{There are a number of very interesting FPL topics to view on the internet and in publications such as *IEEE Spectrum*, *IEEE Computer*; and *IEE Review*. In addition, take time to look at http://www.xilinx.com and http://www.altera.com.} The aspects outlined begin with an overview of field programmable logic—in particular the Altera FLEX 10K device\cite{3}. Note that the results are applicable to other Altera families with minor modification. Next, existing research and methodologies are discussed that cover FPGA design tools, custom-configurable computing, high-level design synthesis, and design optimisation.

2.2 Field Programmable Logic

The use of field programmable logic has altered the design processes for the creation of digital systems. FPGAs\footnote{The terms FPLD (Field Programmable Logic Device) and FPGA (Field Programmable Gate Array) are used interchangeably throughout this thesis to refer to high-end programmable logic devices.} enable the rapid prototyping, development, and testing of digital circuits.

FPGAs are VLSI circuits that are designed to be configured to perform desired tasks. They are made up of logic cells that can each be configured to operate as a lookup
table, latch, or arithmetic function. These logic cells exist in a grid of programmable interconnects of varying lengths. The logic cells can consist of transistor pairs, two-input logic gates, multiplexers, lookup tables, or wide fan-in AND-OR structures[78]. These different logic-cell granularities map efficiently to different sorts of circuit designs. In some FPGAs there also exist programmable RAM cells that can be used, for example, to implement a dual-port RAM or large lookup-table based function.

There are three main technologies utilised in the programming of FPGAs. The first is Static RAM based (SRAM) where a switch transistor is controlled by the logic level of an SRAM bit. These need to be re-programmed from external memory every time the device is powered up, but they generally contain the circuitry to initialise themselves automatically. The second programming technology is EPROM based (Erasable Programmable Read-Only Memory\(^3\)) that uses the logic values in the EPROM cells to control switching transistors. These devices retain their configuration when power is lost from the FPGA. Lastly, there are FPGAs configured through the use of antifuses. The configurations for these devices are non-volatile. Antifuses are permanent short-circuits caused by the applied programming voltage. Antifuses are more compact than either of the other programming technologies.

There are associated disadvantages with using FPGAs that mean they are not the best choice in all circumstances. Some of the general features of design using FPGA technology, compared to MPGA (Mask Programmable Gate Array) and VLSI ASIC (Application Specific Integrated Circuit) technologies, are given below.

**Speed:** The maximum operating frequencies of FPGAs lie between those of SSI (Small Scale Integration) devices and MPGA/ASIC circuitry. This is mainly due to the spurious resistance and capacitance introduced by the programmable interconnects. In addition, the generic layout of logic components and their interconnects produces non-ideal routing compared to the scope a designer has in VLSI design.

**Density:** ASIC and MPGA logic is more densely packed than FPGA logic. This is partly due to the on-chip programming circuitry present in FPGA devices filling space that cannot be utilised by the circuit designer.

**Development time:** The tools for developing FPGA designs are highly affordable even for small design houses. These tools include prototyping and simulation utilities that increase the speed at which design/test iterations can proceed. Some aspects of design for MPGA and ASIC circuitry are absent from the FPGA design process resulting in a more rapid development. Such aspects include test-pattern generation, mask making, and wafer fabrication.

\(^3\)Or EEPROM (Electrically Erasable Programmable Read-Only Memory).
Prototyping and simulation: The rapidity at which testing can be done reduces the overall design time. Design flaws can be found and rectified quickly, and the process can continue right up until manufacture of the finished circuit. Timing simulations indicate operating frequency constraints of FPGA, ASIC, or MPGA designs. However, the timing characteristics are generally known more accurately for FPGAs than for the other technologies, resulting in a quicker simulation for the same timing accuracy.

Manufacturing time: The lead time for ‘manufacturing’ an FPGA-based circuit is lower than for the other custom-configurable technologies because it simply consists of configuring the device. MPGA and ASIC circuit manufacture is not performed in-house, and can introduce significant delays in the production phase.

Extra design methods: FPGAs allow the use of some unique design methods. One is that on each power-up a device can be configured with a different hardware design. This can be useful in circuits that operate in different modes or in a single design that can be sold as two distinct products. In addition, the use of run-time (dynamic) reconfiguration can be utilised to reconfigure the FPGA while the circuit is operational. This has been exploited by designers and researchers to increase the effective density of the FPGA circuitry as discussed in section 2.5.1.

Future modifications: FPGAs provide a low cost method of changing a design late in the design cycle, or even after the product has been released. It is important to allow for design modifications when choosing a device so that there are enough logic and I/O resources available.

Inventory risk: As with SSI and MSI devices, FPGAs offer a low inventory risk because the same device can be utilised in, and perform different functionality in, different designs and products. MPGA and ASIC devices are constrained to perform their specific function, and future designs that incorporate them are bound by this.

Cost: All the above features impact on the cost of using each technology—it is not just the device costs that are important. The break-even point for device quantities depends on the particular application.

2.3 Altera FLEX 10K Device Family

The synthesis and optimisation algorithms use technology specific mapping for their operation so a suitable device family was chosen to meet the research requirements for this thesis. The Altera FLEX (Flexible Logic Element matrix) 10K family was chosen
as the device basis of this research. The reasons for this are threefold. Firstly, the family has a wide range of members (from to 10,000 to 250,000 typical gates; logic and RAM). Secondly, its Embedded Array Blocks can be used to implement RAM-based functions. These provide a different way of implementing some functions, thus providing a broader design-space to be searched. Lastly, it is a family of powerful devices that has a lot of support in terms of the Max+Plus II development environment (for timing analysis, compilation and simulation). The family is established and so there is a large base of designs available for modification and reuse.

In general, the FLEX 10K family is described as applicable to memory functions, complex logic functions (such as digital signal processing), and wide data path manipulation[3]. Additionally, the 10K family is appropriate target technology for the implementation of microcontrollers.

2.3.1 Altera Hardware Nomenclature

The algorithms presented in this thesis operate on, and calculate, values denoting the hardware resource requirements of a circuit for implementation in a target FPGA. There are many hierarchical levels of abstraction inside an FPGA and for this reason it is wise to clarify the nomenclature observed herein.

There are two different types of logic block that are dealt with:

**Logic Array Block (LAB):** The usage of LABs is measured in, and reported by Altera's software in, logic elements (LE) or logic cells (LC). The term *logic cells* was chosen as the standard for this thesis for quantitatively describing LAB usage.

**Embedded Array Block (EAB):** The building blocks of the EABs are RAM bits or memory bits. The term *memory cell* is used in this thesis to represent a memory bit when performing quantitative analysis of EAB usage. This is done to keep the term generic for different device structures. In some cases, reporting and analysis may done on memory cells that represent multiple memory bits.

2.4 FPGA Design Tools

This section discusses the wide variety of design tools available to the FPGA circuit designer. The research of this thesis is based around Altera FPGAs and the Altera Max+Plus II development environment. For this reason, the focus will be on the features of Max+Plus II with some discussion of alternative environments and tools.

The design process for programmable logic systems requires tools for the following aspects:
Design entry: These tools provide methods for describing the functionality of a circuit. This can be done through graphical design entry, a hardware description language, or some other appropriate means.

Physical synthesis: Tools that provide for the translation of the design input into a low-level format that can be used to program an FPGA device. There are many stages performed by a synthesis tool, but the final result is a configuration bit-stream able to be downloaded into an FPGA.

System verification: Tools that utilise accurate timing models of devices and their configurations. Both functional and timing simulations can be performed and the circuit operation and performance can be verified.

Reporting: These tools provide technical reports on device timing, resource usage, and aspects required for the use of the configured FPGA in a circuit (such as its pin I/O layout).

2.4.1 Hardware Description Languages

Arguably the most important FPL design tools are the Hardware Description Languages (HDL). These allow a design to be expressed in a hierarchical manner. They share similarities with software programming languages, such as Pascal and C, but are generally concurrent languages that do not contain the concept of program flow. There is a growing tendency towards crossover between programming languages and HDLs as compilers exist that can produce the hardware configuration and low-level software op-codes from a single source listing. (See sections 2.5 and 2.6.3.) HDLs support many software design paradigms and design methods. Two examples are source-code version control systems, and object-oriented design (incorporating features such as data abstraction and encapsulation). In addition, HDLs can be used in conjunction with the other FPGA design methods.

The most widely-used HDL in industry is VHDL (Very high speed integrated circuit HDL). It is supported by most, if not all, FPGA vendors. It is useful for describing the behaviour of digital systems at various levels of abstraction, and is portable between many design environments. It is a specification language that follows an object-oriented approach; stressing object-oriented specification and reusability concepts[78]. VHDL was developed for the US government in the early eighties, and is the official standard HDL for the United States Department of Defense[87]. Since 1987 it has been standardised by the Institution of Electrical and Electronic Engineers in IEEE standard

\[^4\]Verilog HDL is also widespread in industry, but not quite to the extent of VHDL.
Altera has a proprietary HDL, known as AHDL (Altera HDL). It is a high-level, modular, concurrent language that is tightly integrated into the Max+Plus II design environment. It is convenient for the design of digital circuits using boolean functions, equations, truth tables, and macrofunctions. The language includes native constructs for creating circuit components such as registers, state machines, lookup tables, and other sequential and combinational logic circuits.

AHDL was chosen over VHDL as the language for intermediate circuit representation in this thesis. The first reason for this is that legitimate AHDL descriptions are always synthesisable into configuration bit-streams. Secondly, AHDL can be considered to be a subset of VHDL. The similarities mean that the majority of the research using AHDL is also applicable to development using VHDL.

Complementing the widely used HDLs are many languages used mostly in academic research, and in the proof and testing of new concepts and methods. An example of such a language is Pebble[49] whose features include the support for design target parameters (such as design size and number of pipeline stages), implementation floor-plan placement attributes, and run-time reconfiguration.

2.4.2 Graphical Design Tools

In addition to hardware description languages, most FPGA development environments allow the graphical entry of designs. Altera’s Max+Plus II environment enables a design to be entered as a schematic diagram with interconnections of various hardware elements. The most common of these elements are:

**Primitive logic elements**: Fundamental digital-logic hardware elements such as boolean logic gates, buffers, and power connections.

**Macrofunctions**: Predefined functional sub-circuits such as arithmetic functions, shift registers, encoders and SSI functions from the ‘74xx’ integrated circuit family.

**Megafunctions**: Parameterised circuit modules that perform functions such as FIFO RAMs, multipliers, or multiplexers. There are also complex functions available such as a Digital Phase Detector, an NTSC Video Control Signal Generator, and a Phase-Locked Loop.

**Hierarchical designs**: Circuit designs made separately with any of Altera’s design tools such as with hardware description languages or graphical design entry.
2.4.3 Optimisation Tools

Designs need to be optimised to make the most efficient use of the limited available hardware resources. In addition to human-based optimisation, where knowledge of the circuit is used in choosing the circuit structure, there are tools to assist the FPGA designer in optimising hardware resource usage. FPGAs contain more complex architecture than corresponding ASIC devices and therefore circuit optimisation for them is a more difficult problem for an automated system.

The overall goal of an optimisation tool is to remove redundancy and ensure that the target circuit is implemented as efficiently as possible given various constraints—such as maximum circuit size, minimum operating frequency, or maximum time spent performing the optimisation. Two of the common design environments that have inherent optimisation are discussed below.

The widely used Synopsys\(^5\) system employs architecture-specific optimisation to make efficient use of the available hardware resource while optimising device performance. Specific optimisations performed by Synopsys include

- Automatic, complex I/O mapping.
- Automatic resource sharing.
- Automatic module sharing.
- Constraint-driven synthesis and analysis.

The Altera Max+Plus II environment contains a configurable optimisation goal. This can guide the optimiser to minimise area or to maximise speed. To eliminate result bias, the optimisation settings for Max+Plus II were set identically throughout the gathering of the results in this thesis. The settings used are given in table B.12 in appendix B.

2.4.4 Simulation

Simulation of FPGA based designs is an important step in the verification of a design's integrity and performance.

The conventional approach is to apply test vectors to the inputs of a design model and check that the outputs have the correct values. This can be performed on a software model of the design or on an actual configured programmable device. Another common approach is formal verification, which works by proving the analytical correctness of a design (unfortunately many real designs are analytically intractable). These simulation

\(^5\)http://www.synopsys.com
techniques can be blended together by applying symbols instead of logic bits to a design[26]. These symbols propagate through a design as logic would, but the technique improves functional coverage and provides much faster verification of a design.

Simulation methodologies that work on the propagation of test vectors (binary or symbolic) use states to represent the design at a particular point in time. Events occur that cause the system to change states. These transition events can be monitored in continuous-time or discrete-time.

The algorithms for performing the simulations can be time-driven or event-driven. The time-driven approach uses fixed-duration intervals (ticks) between clock increments. All scheduled events are processed after each clock increment and the system state is determined. The event-driven approach determines the system state for each event and then advances the system clock until the next event is scheduled to occur[87].

2.5 Custom Configurable Computing

Field programmable logic has enabled the development of a class of circuit design techniques known as custom configuration. This paradigm comprises a configurable microprocessor core implemented in FPL, which has its application specific operations implemented in conceptually separate functional units—able to be created by the system designer, or automatically synthesised. The basic functionality of the core usually includes fundamental operations such as register manipulation and comparison, program flow control, and memory access. The functional units can be reconfigured on-demand during circuit run-time, or pre-configured at the design stage. The former of these options gives an effective hardware density increase.

Custom configurable systems merge hardware and software and the implementation and optimisation of such systems becomes an exercise in hardware-software (HW/SW) co-design. A custom configurable design developed as part of the research presented in this thesis is CCSSimP[76, 77] (Custom Configurable Simple Processor). The features of the application-configurable CCSSimP, which is completely specified using an HDL, include:

- A 16-bit external data bus and a 12-bit address bus enabling direct access to 4096 16-bit memory locations.
- Two programmer-visible 16-bit working registers. These store operands and operation results.
- Memory mapped input and output.
• Load/Store microprocessor functionality. The simple instruction cycle consists of four machine cycles per instruction. All data transformations occur in the working registers.

• Direct addressing and basic stack addressing modes.

• The ability to easily define custom instructions by extending the core or by using external functional-block circuitry.

• Single-level interrupts.

Another example of a configurable processor is PRISM[6] (Processor Reconfiguration through Instruction-Set Metamorphosis). The approach is to take a C program description of an algorithm and extract information during compilation as to which parts would map efficiently into functional hardware blocks. The new hardware operations are automatically synthesised during compilation. They augment the core processor functionality to provide greater application-specific performance than a general-purpose processor. Published performance gains from using PRISM over a purely software implementation indicate speed improvements ranging from 2 to 54 times[6].

Custom configurable circuits approach the performance of ASICs while retaining much of the generality of conventional computing systems[105]. Development time can be reduced, compared to conventional FPGA hardware design, by the use of software compilers and assemblers—along with associated software design techniques.

### 2.5.1 Dynamic Circuit Reconfiguration

The alternative to the compile-time reconfiguration discussed above is dynamic reconfiguration; also known as run-time reconfiguration. It can involve the entire target device or be partial reconfiguration. There are a number of systems that have been developed that exploit dynamic reconfiguration to reap the associated rewards.

The Dynamic Instruction-Set Computer (DISC) is based upon a core that has performance-enhancing application-specific instructions. The special feature of DISC is that it employs partial reconfiguration. The functional operations are paged in and out of the FPGA on demand, as they are required by the running software code. Indicative published results show speed gains of more than 20 times using DISC techniques[104].

The technique of dynamic reconfiguration is particularly suitable to Artificial Neural Network (ANN) circuits because of their highly regular structure. The Run-Time Reconfiguration ANN (RRAANN)[20, 21] partitions the backpropagation algorithm of a multi-layer feed-forward network into three stages. The three backpropagation stages are feed-forward, backpropagation, and update. Stages are instantiated one at a time
to share the hardware resource through time. RRANN's dynamically reconfigurable architecture has shown a 500% increase in the number of neurons implemented on a single Xilinx XC3000 device. The drawback is that performance is adversely affected due to the reconfiguration overhead.

Hadley et al. present a design methodology for partially reconfigured systems whose overall goal is to maximise static circuitry while minimising dynamic circuitry[33]. This reduces the time spent on reconfiguration, resulting in a more efficient implementation. Differences between functional blocks are extracted from the given design, and these are exploited in the design process to minimise dynamic circuitry in the implementation. There are four aspects of the functional blocks that are considered by this methodology. Blocks are said to differ by:

**Precision:** The blocks perform the same function, but the number of bits manipulated is different.

**Constant value:** A different static value is used by each block while all perform the same function.

**Function:** The blocks are structurally similar, but perform different functions.

**Subset:** Where one block is functionally and structurally contained within the other.

Dynamic reconfiguration is providing circuit compute-speed improvements and effective density increases. Currently there are few commercial design environments that support dynamic reconfiguration, and few methodologies for developing such circuitry.

### 2.6 High-Level Synthesis

A designer needs automated assistance in the development of the complex digital hardware systems that are prevalent today. Software engineers have language compilers that take high-level concepts and produce low-level code. Analogously, there are many systems that support digital hardware designers by taking descriptions at varying levels of abstraction and producing lower-level designs. The inputs of such synthesisers vary from graphical designs to hardware description languages. The outputs vary from hardware description languages to configuration bit-streams. Often, the synthesised output is further synthesisable to produce a lower-level, less abstract, design representation. Differing levels of optimisation are usually performed during synthesis, although this is not mandatory.

High level synthesisers analyse the given functional description and through various algorithms determine the paths through which data will flow in the circuit. The main
2.6 High-Level Synthesis

synthesis tasks include scheduling and data-path synthesis[80]. Scheduling determines the time steps at which different parts of the circuit will be active. Data-path synthesis involves mapping the conceptual functional building blocks of the circuit to actual arithmetic functional units.

The synthesis methodology presented in this thesis is based on taking designs as high-level functional netlists and synthesising hardware description language listings.

2.6.1 Hardware Description Language Synthesis

The synthesis of an HDL listing from a high-level functional description produces a representation of the design that is still at an abstract level. This intermediate circuit description has two main advantages over going from high-level descriptions directly to a configuration bit stream.

The first advantage is that the HDL is, at least ideally, interpretable by a human designer. This allows analysis with the possibility of manipulating the HDL—possibly integrating it with further designs. Some syntheseses preserve the design hierarchy in the HDL representation[49]. In addition they allow node names to be defined in the initial description and carried over into the HDL during synthesis.

Secondly, the HDL is independent of the target programmable device. This means that different target devices can be chosen. Moreover, technology-specific mapping can be performed during compilation of the HDL. Other optimisations can be performed that optimise the design for a particular FPGA device or family. Apart from automatic optimisation, user-directed optimisations (such as floor-plan editing) are possible. The software to assist with the efficient implementation of HDL into FPGAs can remain separate from the HDL synthesiser—leaving each software package to its specialty.

2.6.2 Hardware Libraries

The trend over the last thirty years of software development has been to build up software libraries of common functions that can be reused at will by a programmer. Example libraries are the input/output libraries defined in <stdio.h> and <iostream.h> for C and C++ respectively. Higher level functionality has been included in libraries such as Borland's Object Windows Library (OWL) and, more recently, their Visual Component Library (VCL).

Similarly, the trend has been to provide libraries of reusable hardware objects to aid the designer in the description of a circuit's functionality. These are akin to discrete integrated circuits. Programmable logic hardware library objects exist in different levels of abstraction; some are concrete designs, while others accept parameters to de-
fine their exact functionality. Common parameters are bus width, number of pipeline
stages, and whether operations are signed or unsigned\cite{4}. The primitive logic elements,
'macrofunctions,' and 'megafuntions' outlined in section 2.4.2 are examples of hard-
ware libraries.

Higher-level libraries are available that perform a range of more complex functions
or utilise different methods of instantiating the same functionality. Often, such libraries
are created specifically for a class of target designs. One library of parameterised hard-
ware objects, presented by Lirk et al.\cite{48}, allows varying levels of pipelining and internal
data flow serialisation. It is a library that focuses on designs involving arithmetic, dig-
ital signal processing, and video operations.

2.6.3 Hardware/Software Co-Design

Traditionally, complex digital systems have been implemented in hardware, or in soft-
ware running on a hardware platform such as a microprocessor. Configurable logic has
removed the need for a demarcation between the two approaches. The result is Hard-
ware/Software (HW/SW) co-design in which a design is partitioned into both hardware
and software components. The basis is usually a configurable processor core that runs
software instructions. Parts of the design are implemented as application-specific hard-
ware modules that each implement a specific software command. Other portions of the
design are run as traditional software algorithms.

The approach allows an efficient use of hardware resource as some algorithms are
more appropriately implemented in software, while others map well to hardware. For
example, an algorithm to perform 'Gaussian Elimination' may require excessive hard-
ware resources to be implemented without exploiting the slower, iterative, software
solution (that hopefully falls within the desired timing constraints).

HW/SW co-design encompasses the approaches and algorithms employed in de-
signing a hardware/software system. One of the key problems is that of partitioning
designs into their resulting hardware and software components. There are many al-
gorithms for HW/SW partitioning, with two of the more common being based upon
software-oriented and hardware-oriented approaches\cite{64}. The former of these starts
with an infeasible solution consisting entirely of software components. These software
components are iteratively implemented in hardware until timing constraints are met.
Conversely, the hardware-oriented method starts with an entirely hardware solution
and moves components to software implementations if timing constraints can still be
satisfied. The effectiveness of a partitioning algorithm directly affects how optimally
the corresponding design performs. Once the partitioning algorithm has completed,
the software is compiled and the hardware synthesised.

A design that incorporates software and hardware components is generally defined using two separate methods: a programming language and a hardware description language respectively. There is no ubiquitous unifying design language. Two languages, proposed to fill this gap, have been presented for consideration by the industry[26]. The first is a new language that is a blend of Verilog and C, along with minor aspects of VHDL and Java. It is known as Superlog. The second language, SystemC, is an extension of C/C++ into the hardware domain. Class libraries are defined in C++ that incorporate the hardware functionality descriptions.

2.7 Design Optimisation

Design optimisation of FPGA circuits is necessary because there is always a limited hardware resource available, and usually other constraints such as the time in which circuit operation must be completed. There are often further constraints on the design such as design time and hardware cost. Design optimisation is the task of ensuring that all criteria are met; some of these are firm, while others are negotiable. An example of a firm criterion is the number of logic cells, if a target device has already been chosen. A negotiable criterion could be operating frequency (weighed up against circuit power consumption). The term optimisation does not imply that all circuit features are to be minimised. The levels of minimisation during optimisation are dependent upon the given optimisation criteria.

There are different levels of design optimisation performed during the design process, often performed iteratively. The highest, most abstract, levels occur during the initial design phase. Generally, the higher the level of optimisation, the more effect it will have on the resulting circuit. Consider the following four chronologically ordered optimisation levels ranging from high-level to low-level:

1. Functionality and features are decided upon. The accuracy of the circuit operation and results is chosen. Unnecessary features or excessive desired accuracy will affect the circuit greatly.
2. Circuit components (sub-circuitry) are chosen and their interconnection topology is designed.
3. Redundant logic is automatically removed.
4. The floorplan is laid out and the configuration bit-stream is synthesised. This is also an automated step.
Before optimisation can occur, the criteria must be communicated to the person or algorithm performing the task. This is trivial where there are firm requirements, but not necessarily easy when trade-offs are permissible. Consider a requirement where there is an ideal hardware resource usage ($C$ logic cells), and an ideal circuit compute time ($T$ seconds). For this example it is desired that speed be optimised at a cost of logic cells "to a reasonable extent". Such vagueness is often adequate for a human designer but an automated optimisation algorithm needs to be given a function relating $C$ and $T$ to form a measurable goal.

An analytic interpretation is needed for analysing the results of the optimisation process to see whether conformance to criteria is obtained. It is important when comparing design possibilities that they are being compared fairly. For example, reducing the number of cycles for a circuit to produce a given result may not be advantageous if the cycle period is increased beyond a certain threshold. Similarly, comparing such broad metrics as the number of FLOPS\textsuperscript{7} can give misleading results; a sign-inversion is a lot less computationally intensive than a multiply operation, for example.

The following two subsections discuss design techniques that are particularly relevant to this research. Chapter 5 presents the application of these design techniques to the synthesis methodology introduced in this thesis.

### 2.7.1 Variable-Width Data Paths

The width of a data path is defined as the number of bits of data being processed simultaneously. Conventionally, these are bit-serial or bit-parallel. There is a further possibility of using digit-serial, which is a hybrid of parallel and serial\textsuperscript{39}. For example, a 32 bit number could be processed over 8 cycles with 4 bits processed in each cycle.

The advantage of using bit-parallel data paths is high throughput, at the cost of greater hardware resource requirements. Conversely, bit-serial data paths require less hardware resource but have lower throughput. Digit-serial techniques allow a range of solutions that lie between these extremes, offering extra flexibility to the designer.

Components of a circuit, such as adders and multipliers, can operate on data of different precisions. Shared buses transferring data of different precision can be optimised by utilising digit-serial data communication. By moving away from the conventional serial and parallel data paths, the effect is similar to pipelining the data communication stage to differing degrees.

Variable-width data paths are employed in the synthesiser of this thesis to expand

\textsuperscript{6}The time taken for a circuit to process its inputs and present the result at its outputs.

\textsuperscript{7}Floating-point operations per second.
the design space search with the aim of finding optimal solutions.

2.7.2 Hardware Evolution

One method of optimisation that has a wide range of applications, including hardware circuitry, is artificial evolution. Evolutionary techniques are generally known as Genetic Algorithms (GA) or Evolutionary Algorithms (EA).

There are two modes in which hardware evolution occurs: off-line and on-line. In off-line evolution, the GA is performed on a master computer and the fitness of each design is measured by implementing it on an actual device or via a simulation. On-line evolution occurs when there is no external controlling entity and the evolutionary process is embedded in the target device. The term evolvable hardware is reserved for techniques where an electronic circuit, as opposed to a simulation, is used to determine a design's fitness.

Thompson et al. present a summary of design space exploration through artificial evolution[94]. In their paper they make hypotheses about design spaces, characterise conventional hardware design, and develop evolutionary case studies.

Evolutionary techniques are useful in three main aspects of hardware design as discussed below.

Self-Replication, Self-Repair, and Self-Improvement

These evolutionary techniques are all, by definition, on-line techniques. The example systems and definitions below are discussed by Sipper and Ronald[85].

Self-Replication: Such a system is capable of producing a new faultless copy of itself. In this way they are able to recover from major faults.

Self-Repair: These systems are able to recover from minor faults. Techniques such as interconnection rerouting, and FPGA hardware sub-circuit instantiation, are employed. An example of a self-repairing system is The BioWatch developed by Mange et al., which is capable of detecting and fixing any flaws in its circuitry by reconfiguring its hardware to compensate for the faults.

Self-Improvement: A system capable of self-improvement utilises a dynamic configuration that is modified during run-time to improve its performance. One such example is The Firefly (Sipper et al.), which evolves itself to synchronise its initially random on-board oscillators. Another application of self-improvement is a prosthetic hand (Higuchi et al.) that modifies itself to adapt to the nerve signals
in the disabled recipient's forearm. Normally, a patient must learn to adapt their nerve signals to meet the requirements of an artificial hand.

**Low-Level Design Ingenuity**

Evolutionary design techniques can be used to create designs at a low level, such as by manipulating the configuration bit-stream for an FPGA. Alternatively, fundamental circuit building blocks, such as transistors or logic gates, can be combined by a genetic algorithm to produce a circuit with the desired functionality. Low-level designs are tested on the target device, and the output from the physical circuit is used to guide the genetic algorithm.

Perhaps the most well known evolved low-level design is the oscillator design presented by Thompson[92], which was evolved in an actual FPGA using the deviation of the circuit output frequency from that desired as the guiding evolutionary fitness. The resulting design was analysed and it was determined that, in the absence of artificial constraints introduced by a human designer, the design utilised inherent and spurious features of the FPGA. Effectively, the digital discrete-time components are treated by the genetic algorithm as continuous-time dynamic analogue systems exhibiting a broad range of dynamic behaviour[93]. The advantage of this approach is that 'free' resources, such as stray capacitances, can be used in a design. The disadvantages occur because the 'free' resources can alter with device, time, and temperature, and the circuit may cease to function. Moreover, the design must be thoroughly reverse-engineered (and extraneous effects considered) for a formal analysis of the design's robustness to be performed.

Simple, well-known circuits have had their design spaces partially searched using GAs, and designs have been produced that were previously patented. The search for patentable, previously unpatented (financially lucrative) designs continues[85].

Designers can learn from the results of low-level circuit evolution. When a novel design is created by a genetic algorithm, and it is interpreted, it can be used in conjunction with conventional design approaches.

---

8Nichols[63] discusses the limitations, and application, of patents to circuits and software developed using evolutionary methods. The conjecture is that design ownership is unclear—especially when the operation of the evolved solution is not understood. The issue is discussed further in 'Letters to the Editor,' *IEEE Computer*, June and September 1999.
2.8 Conclusions

High-Level Design Ingenuity

This aspect of evolutionary design, also known as *adaptive design*, is the approach used in the research presented in this thesis. It is an off-line technique that manipulates a high-level representation of a design using an evolutionary approach—such as a GA. The effect is that the design space is searched in an effort to find an optimal design given various design criteria and constraints. It is known also as *functional level evolution* and it is accepted that these higher-level methods enable the evolution of more complex applications[95] than do the other evolutionary techniques.

The approach differs from a low-level approach by the abstraction at which the circuit is represented. Instead of evolving configuration bit-streams, higher-level circuit modules are exploited. These modules are abstractions of circuit features, such as interconnects or functional sub-circuits designed *a priori*, that have known functions and effects. Moreover, this off-line approach utilises models of the modules. High-level design requirements can be applied, such as the use of synchronous clocking for all modules. Stray capacitances and other spurious effects are not utilised by the optimising evolutionary algorithm. Successful applications of this approach include performance-driven VLSI routing[47], and evolution of a structured-grammar hardware description language for automated hardware design[61].

The task of reverse engineering and analysing an evolved solution is much easier for the high-level approach than for low-level evolution.

2.8 Conclusions

The ever-maturing realm of field programmable logic is allowing an ever-diversifying path for system design and optimisation. The line between hardware and software has become very blurred, and design techniques and principles from each arena are being transferred to the other. New circuitry is possible, in which the hardware changes functionality between operations of the circuit or during its operation. This reconfigurability also allows optimisation techniques that exploit the ability to test a design on the actual target device. Programmable hardware has not only enhanced rapid prototyping and development, but has revolutionised digital hardware design.
Chapter 3

Stochastic Neural Network Synthesis

3.1 Introduction

Special purpose synthesis is performed by tools that have a specific knowledge of the particular circuit being implemented. This means that they can perform optimisations that would otherwise be invalid. The main advantage is that a more compact and better designed circuit may be created; the associated disadvantage being that the synthesiser has to be specifically written for a certain type of circuit. The synthesiser discussed here was specifically written to create stochastic neural networks and has been published[55].

Artificial Neural Networks (ANN) benefit from the parallelism of hardware, and the ability to specify ANNs at a high level is advantageous. Many neural network architectures have been designed for, and implemented in, programmable hardware. The main advantage of using programmable hardware is that a new topology may be created for each neural network instance. This means inter-neuron communication bandwidth can be ‘created’ where necessary.

Our aim was to develop a system that could take a high level description of an artificial neural network and synthesise HDL code to configure an FPLD to perform as the prescribed network. In addition, specialised techniques were used to make sure the FPLD solution complied with certain criteria; the main one being low hardware resource usage. A tool to abstract details of the testing process was also developed, such that it took data to be presented to the ANN and converted it into a format that could be used in the Altera simulation environment. A similar tool was necessary to convert the simulation outputs into a form that was interpretable and readily visualised. As
well as providing a tool that can assist hardware ANN designers, similar methodologies can be applied to other applications where the designer does not need to know the low-level details of FPLD configuration and testing, while specialised processes ensure that appropriate design methods are used.

Generally, neural networks are simulated purely in software, on microprocessor based hardware systems, or on specialised hardware, but each of these methods has its drawbacks. Software simulation is relatively slow with its inherent sequential processing. Microprocessor based systems are not designed to optimise interconnections and neural processing. Specialised hardware is often difficult to reconfigure for different ANN architectures.

FPLDs are appropriate as a medium for implementing neural networks because they provide the parallelism that is required for high-speed operation as well as the ability to be custom configured and routed for each particular neural network. The price for this flexibility is that FPLDs still have much less capacity than their VLSI counterparts and so fewer neurons can be implemented per device.

A decision had to be made at the outset as to which hardware method for implementing neural networks would be chosen. The architecture that provided a solution appropriate to demonstrate special-purpose synthesis was based on serial stochastic number representation for data within the neural network. The serial nature of these streams reduced the routing requirements greatly compared with a parallel solution. The implementations of the necessary multipliers and activation functions also have very efficient resource requirements when stochastic numbers are used.

3.2 Hardware Neural Network Implementations

The inherent parallelism of artificial neural networks means that they can be efficiently implemented in hardware. There have been many ANN models designed and implemented on hardware. A few of these that were targeted towards VLSI and non-FPGA hardware are described in this section.

Stochastic neural networks have been designed and implemented in hardware. These implementations make full use of the available hardware resources, both in terms of routing and in their ability to cluster appropriate groups of hardware together[2]. Similarly, the necessary task of implementing random number generators in VLSI hardware also makes use of the fine control that the designer has over placing and routing various structures[74].

There are a number of neural network models based on RAM (Random Access Memory). These can be efficiently implemented in standard hardware; using discrete
RAM devices or standard VLSI RAM designs. One such neural network, which possesses the natural capacity for generalisation, is the ‘i-pRAM’ presented by Gorse and Taylor[29].

VLSI also allows the use of analogue design techniques to implement circuits that share properties with biological neural networks—in particular, compact neural processing. The drawbacks of on-chip analogue circuits are their sensitivity to noise, lack of resolution, and the difficulty of communication across device boundaries. The advantageous characteristics are demonstrated in the Hippocampal Model implementation by Sheu et al., which uses analogue circuitry for compact parallel computing and digital circuitry for interconnection[84].

3.2.1 FPGA-Based Implementations

There have been many implementations of ANNs since FPGAs became popular for research in the early 1990s. A brief overview is given here to provide some insight into the possibilities that field programmable logic allows the ANN designer. The three following implementation methods are the most prevalent in the field.

Stochastic Bit Stream Implementations

ANN implementations based on stochastic bit streams take advantage of the correspondence between digital hardware structures and stochastic neural network topological structures. These similarities are evident in the low routing requirements and the relative ease of implementation of ANN specific arithmetic operations and numerical mappings. This means that such networks can be efficiently implemented in digital hardware. Stochastic Neural Networks are presented in greater depth in section 3.4.

Bit-Serial Implementations

These implementations exploit the hardware routing efficiency of serial data streams. In addition, they allow the necessary arithmetic bit stream operations to be performed on serial data, resulting in a lower hardware resource cost than operations on parallel data streams[50]. One example of a bit serial ANN implementation is that presented by Salapura, Gschwind and Maischberger[75]. Their associated ANN tools include the automatic synthesis of neural networks, but differ from the tools presented in this thesis because they deal with bit-serial data and not stochastic bit streams. Both number representations, however, result in dense network architectures.
<table>
<thead>
<tr>
<th>Stochastic bit stream</th>
<th>Real value represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111</td>
<td>+1.00</td>
</tr>
<tr>
<td>00000000</td>
<td>-1.00</td>
</tr>
<tr>
<td>11110000</td>
<td>0.00</td>
</tr>
<tr>
<td>00101011</td>
<td>0.00</td>
</tr>
<tr>
<td>00001000</td>
<td>-0.75</td>
</tr>
<tr>
<td>01110101</td>
<td>+0.25</td>
</tr>
</tbody>
</table>

Table 3.1: Example 8-bit bipolar stochastic stream representations of real values.

Dynamic (Run-Time) Reconfiguration Implementations

ANN implementations can increase the effective neuron density by using dynamically-reconfigurable programmable hardware. Each hardware neuron layer is subject to time division multiplexing (resource sharing), which increases the spatial efficiency of the network. This comes at the expense of losing the parallelism that enables all layers to be processed concurrently, but retains the parallelism within each layer. There are a number of ANN implementations that utilise dynamic reconfiguration[20, 21, 50], and partial reconfiguration[33].

3.3 Stochastic Number Representation

Stochastic bit streams are long, probabilistic, binary streams where the proportion of the total bits that are '1's represents a real numerical value. Each bit in a stochastic bit stream can be thought of as a random variable produced by a Bernoulli trial, whose probability depends on the numerical value of the entire stream. For example, a bipolar system may represent a positive '1' with a stream of all ones and a negative '1' with a stream of all zeros. In this case, a stream with half ones and half zeros would represent a zero value. Some examples of (impractically short) stochastic streams are shown in table 3.1. It can be seen from this example that there is a loss of information in representing numbers this way. There can be only nine distinct numbers represented by eight bits; these range from no ones to eight ones. In addition, stochastic bit streams must be long to reduce random errors. It can be shown that for a stochastic bit stream to achieve n bits of accuracy, it must consist of $2^n-2$ bits. This is necessary to ensure the elimination of inaccuracies due to random variances[97].

There are two common mappings of a numerical value, $x$, to a Bernoulli random variable: unipolar and bipolar. The unipolar representation has $0 \leq x \leq v$, while the bipolar representation has $-v \leq x \leq v$. 
3.3 Stochastic Number Representation

The major advantage of stochastic number representation in hardware is that the multiplication of two stochastic bit streams requires only a single two-input logic gate. In the case of two unipolar streams this is an ‘AND’ gate. In the case of two bipolar streams this is an ‘XNOR’ (eXclusive NOR) gate. The result of the multiplication is scaled by a factor of $1/n$, which keeps the range of the numbers normalised within a circuit—which is advantageous for a neural network circuit. Although this multiplication is simple, there must be as little correlation as possible between the two streams for accurate multiplication to be achieved.

3.3.1 Stochastic Number Generation

A stochastic bit stream is a weighted random bit stream. Interestingly, the generation of a stochastic bit stream representing any value within its range can be achieved using weighted random bit streams with fixed values. Each of these fixed-value streams is called an address line stream. With $x$ address line streams, the resulting stochastic bit stream can represent $2^{(2^x)}$ different values.

Each static weighted bit stream is produced by starting with a random bit stream where there is an equal probability of each bit being a ‘1’ or a ‘0’. This stream is applied to modulating hardware as described in section 3.6.2. The output of the modulator is a bit stream with the desired fixed weighting.

Once the address line streams are available, they can be applied to the inputs of a lookup table (LUT). These input lines are also known as address lines. The output of the LUT is a stochastic bit stream whose value depends on the binary value stored in it[97]. A LUT with three address lines will hold an 8 bit binary value.

If multiple, uncorrelated, stochastic bit streams are required in a circuit, the address line streams must be uncorrelated. The next section discusses the generation of random bit streams, focusing on the generation of multiple, uncorrelated streams from compact circuitry.

3.3.2 Random Number Generation

The random numbers that are generally created in hardware are actually pseudo-random binary sequences (PRBS) that repeat after a certain (large) number of clock cycles, often in the order of $2^{32}$ to $2^{64}$. There are methods for generating truly random numbers in hardware such as measuring the noise created by thermally agitated electrons, as in certain Intel chip-sets[60], but PRBSs are generally adequate.

Linear Feedback Shift Registers (LFSR) can be used to efficiently generate pseudo random binary sequences. (See Saarinen et al.[74] for a VLSI implementation and sub-
sequent analysis of the resulting statistical PRBS properties.) Implementing a LFSR for each desired random bit stream would require a massive hardware resource requirement.

To provide uncorrelated bit streams with a low hardware resource usage, the chosen method is based on delayed sequences. In this method, uniquely delayed versions of the PRBS provide the uncorrelated address line streams. This is effected by passing each PRBS through a D-type Flip-Flop between feed-offs. Although the resulting address line streams are highly overlapping (a shift of one bit), this method provides almost perfectly uncorrelated stochastic streams[7]. Using simple delays has the added advantage of being an operation that can be done in hardware spread throughout a device, thus reducing routing requirements compared with routing a unique PRBS from where it is centrally produced to where it is required.

The correlation of address line streams generated by a single LFSR can be reduced by using a number of non-overlapping portions of a PRBS. Each of the non-overlapping sequences can be used as one of the address line streams and then be shared amongst the circuitry using the previously described method of simple delays.

**Producing Delayed PRBSs from a Maximal LFSR**

It is possible to determine feed-off taps that, when XOR'd together, produce a delayed version of the PRBS for every possible delay $t$ where $0 \leq t < (2^n - 1)$ for a maximal LFSR with $n$ stages[2]. A **maximal** LFSR is one that produces an $n$-maximal PRBS of maximal length, which is $2^n - 1$ for an $n$ bit shift register. In addition to having maximal length, an $n$-maximal PRBS includes every possible combination of $n$ consecutive bits except the all-zero combination. To produce non-overlapping $n$-maximal sequences, the delay $t$ is chosen such that it is greater than the number of cycles that the circuit is to be operated for. For example, a circuit may be run for 1000 cycles and require four PRBSs. A 12 bit shift register would produce a maximal length PRBS with length 4095 bits. The four PRBSs could be taken from the maximal length PRBS with delays of 0, 1000, 2000, and 3000 cycles. As long as the circuit was not run for more than 1000 cycles, the delayed versions would be effectively uncorrelated[2].

Taking feed-off taps from the shift register stages creates additional load for the stages. This is a problem in VLSI circuitry with its limited fan-out and can be reduced by roughly even distribution of the feed-off taps across all shift register stages. However, it is not such a problem for FPLD based circuits.

Consider an $n$ bit shift register with XOR (⊕) feedback taps taken from the shift register stages given by a table of primitive polynomials[7]. The number $i$ in a primitive
Figure 3.1: Linear-feedback shift register (LFSR) showing taps for uncorrelated pseudo-random bit streams. (a) Feedback taken from stages $z_4$ and $z_1$ to produce a maximal length PRBS. The initial shift register contents are $00111$. (b) Feed-off taps from stages $z_3$, $z_2$, and $z_0$ produce a PRBS delayed by seven cycles.

A polynomial (ignoring $i = 0$) indicates that a tap should be taken from the output of the $(i-1)$th stage. The taps are XOR’d together and fed into the input of stage 0. The primitive polynomial $(5, 2, 0)$ indicates feedback taps are taken from the outputs of stages 1 and 4.

Each stage can be assigned a binary value $c_i$ indicating whether a tap is taken. The feedback taps can then be represented by a vector $c$ with $n$ binary elements $c_i$ where $n$ is the number of stages in the shift register. The general case is

$$ c = \begin{bmatrix} c_{n-1} & c_{n-2} & \cdots & c_2 & c_1 & c_0 \end{bmatrix} $$

(3.1)

In the example primitive polynomial $(5, 2, 0)$ shown in figure 3.1(a):

$$ c = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix} $$

(3.2)

The shift register is represented by the vector $z$ that is initialised with an arbitrary
binary seed. The general case of $z$ for an $n$-stage shift register on cycle $k$ is:

$$z(k) = \begin{bmatrix} z(k)_{n-1} \\ z(k)_{n-2} \\ \vdots \\ z(k)_2 \\ z(k)_1 \\ z(k)_0 \end{bmatrix}$$

(3.3)

The specific case for the initial shift register contents of 00111 on cycle zero (as shown in figure 3.1(a)) is represented as:

$$z(0) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

(3.4)

The five-stage shift register's feedback can be described in terms of the stage contents as $z_0 = z_1 \oplus z_4$. The entire shift register contents between two successive cycles $k$ and $k + 1$ are governed by the equation:

$$z(k + 1) = M \cdot z(k)$$

(3.5)

which is equivalent to:

$$\begin{bmatrix} z(k + 1)_{n-1} \\ z(k + 1)_{n-2} \\ \vdots \\ z(k + 1)_2 \\ z(k + 1)_1 \\ z(k + 1)_0 \end{bmatrix} = M \begin{bmatrix} z(k)_{n-1} \\ z(k)_{n-2} \\ \vdots \\ z(k)_2 \\ z(k)_1 \\ z(k)_0 \end{bmatrix}$$

(3.6)

Matrix $M$ is known as the state transition matrix[2] and is defined as:

$$M = \begin{bmatrix} 0_{n-1} & I_{n-1} \\ c_n & \end{bmatrix}$$

(3.7)

where $0_{n-1}$ is the $n - 1$ component all-zero column vector, $I_{n-1}$ is the $(n - 1) \times (n - 1)$ identity matrix, and $c_n$ contains the $n$ feedback tap coefficients given in equation 3.1.

For the example five-stage shift register with the taps given in equation 3.2, the state
transition matrix is:

\[
M = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0
\end{bmatrix}
\]  \hspace{1cm} (3.8)

On cycle \( t \), the shift register contents \( z(t) \) can be calculated using the state transition matrix and the initial shift register contents:

\[
z(t) = M^t \cdot z(0)
\]  \hspace{1cm} (3.9)

The delay, \( t \), is usually very large, to produce long-term non-overlapping pseudo-random binary sequences. The feed-off taps can be obtained from \( M^t \), which can be calculated relatively quickly (in \( \log(t) \) time) by first calculating a table of the binary powers of \( M \). That is, \( M^0, M^1, M^2, M^3, M^4, \ldots, M^{2^{n-1}} \).[99]

The fundamental PRBS can be taken as the output of shift register stage \( z_0 \). To determine the feed-off taps to obtain a delayed version of the PRBS, the example matrix \( M \) from equation 3.8 is raised to the power of \( t \), where \( t \) is the required delay—in this example a delay of seven cycles:

\[
M^t = M^7 = M^{4+2+1} = \begin{bmatrix}
1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0
\end{bmatrix}
\]  \hspace{1cm} (3.10)

The top row of the resulting matrix gives the XOR tap coefficients[2] in the form:

\[
\begin{bmatrix}
\alpha_0 & \alpha_1 & \cdots & \alpha_{n-2} & \alpha_{n-1}
\end{bmatrix}
\]  \hspace{1cm} (3.11)

(which is in reverse order with respect to vector \( c \) given in equation 3.1). Thus, in the matrix given in equation 3.10, the top row describes a tap coefficient vector indicating that the outputs of shift register stages \( z_0, z_2 \) and \( z_3 \) should be XOR'd together as shown in figure 3.1(b) to provide a PRBS delayed by seven cycles with respect to the fundamental sequence. Table 3.2 gives the tap expressions to produce sequences delayed by various numbers of clock cycles.

Conversely, it is possible to determine the PRBS delay associated with a given tap pattern by implementing an efficient algorithm for discrete logarithm calculation in a Galois field of polynomial degree \( n \): GF\((2^n)\) [2]. Coppersmith has shown for these fields
Table 3.2: Feed-off taps for delayed versions of the pseudo-random binary sequence from a 5 stage linear-feedback shift register.

(of characteristic two) a method for determining discrete logarithms\(^1\) whose asymptotic running time is \(O(\exp(O(n^{1/3} \log^{2/3} n)))\) \([15]\).

3.4 Stochastic Neural Networks

Artificial neural networks are parallel networks that are based upon a greatly simplified model of the mammalian brain. During the training phase, an ANN is presented with example data (and evaluations of that data for supervised learning). From this they learn to generalise—that is, they can classify data not previously presented. ANNs are used widely for tasks that are difficult to perform using conventional hardware and software systems. Examples include the control of complex systems and the recognition of hand-written characters.

For readers not versed in ANN theory, there are a few background sources worth reading: Freeman and Skapura[25] present an introduction to the neurobiological aspects of ANNs and a look at the implementation of ANNs from a programmer’s perspective. Hassoun[34] gives a highly mathematical introduction to the fundamentals of neural networks. Hush and Horne[36] provide a review of progress in the area of supervised neural networks, while Kit[41] provides a broad analysis of neural network models based on unsupervised learning.

\(^1\)Galois fields GF\(2^n\) are therefore not the best choice for cryptographic schemes that are based on the difficulty of calculating discrete logarithms in them[15].
A typical neural network consists of a number of interconnected neurons that implement a non-linear transformation of an input vector to an output vector. One of the most common network structures is the multi-layer perceptron (MLP) model. This is an arrangement of layers of perceptrons, a type of neuron. An MLP with two hidden layers (neuron layers between the input and output layers) has been shown to be capable of forming an arbitrarily close approximation to any continuous non-linear mapping[36].

The stochastic neural network (SNN) architecture described in this chapter is based upon those presented by Bade and Hutchings[7] and van Daalen, Jeavons and Shawe-Taylor[96]. It was chosen because it has been shown to be compact, and contains features that can be used to improve the density of implemented neurons in a programmable device. It is based on stochastic numbers in serial streams, has synapse weights stored on-chip, and reduces its activation function hardware into compact forms. The disadvantages of the architecture are its non-deterministic accuracy, and the long bit streams that have to be processed. The synthesised neural network is trained before synthesis and, once on the FPLD device, its synaptic weights are not modifiable without reconfiguration. Zhao, Shawe-Taylor and van Daalen[106] discuss various aspects of learning in stochastic neural networks.

3.5 Synthesiser and Analysis Tool Overview

Two complementary tools are profiled in this section: the stochastic neural network synthesiser and the corresponding analysis tool. The synthesiser produces the AHDL, which is subsequently compiled to produce the configuration bit-stream for an FPLD. The analysis tool is used in the verification of the functionality of the resulting neural network design. The SNN synthesis and verification process flow is given in table 3.3.

3.5.1 Stochastic Neural Network Synthesiser

The neural network synthesiser takes a text file description of the neural network to be synthesised and produces a number of text design files (TDF) that completely specify it. The TDFs contain synthesised AHDL ready to be compiled into a configuration bit-stream for the appropriate target device. Each contains the hardware description for a distinct part of the network; in this way a hierarchical project is created that can easily be interpreted and modified by a human designer. Moreover, the TDFs contain documentation that indicates what they represent and their respective parameters. This is important if a user wants to modify parts of the synthesised code, and for
Step 1  *Create weight file.* Obtain weight file from a trained ANN, using simulation software such as SNNS (Stuttgart Neural Network Simulator).

Step 2  *Synthesise SNN.* Synthesiser produces all AHDL design files for further compilation.

Step 3  *Compile synthesised AHDL.* Use the Altera design environment to compile the resulting text design files.

Step 4  *Create test input vector.* List real-valued inputs to be applied to the SNN circuit.

Step 5  *Convert test vector to stochastic vector.* Use analysis tool to create stochastic representation of the test vector.

Step 6  *Apply input to FPLD or simulator.* Use either the timing simulator or actual FPLD.

Step 7  *Convert results into real values.* Derive real-valued result vector from stochastic results using the analysis tool.

Step 8  *Graph output.* Convert result vector to a Matlab file for two-dimensional or three-dimensional graphing.

---

Table 3.3: *Overview of the process of synthesis and verification of stochastic neural networks using the SNN synthesiser and the SNN analysis tool.*

Locating possible bugs in the synthesiser. The hierarchy of the main SNN component files created is given in figure 3.2. An example of synthesised AHDL output from the synthesiser is given in appendix A.

### 3.5.2 Stochastic Neural Network Analysis Tool

The neural network analysis tool was developed to allow simple and efficient testing of a stochastic design by providing a level of abstraction from bit level testing. Inputs to a system can be specified as real numbers, and the test outputs are automatically converted to real numbers and stored in Matlab files for graphical display and subsequent analysis. The analysis tool uses a file of desired inputs and testing parameters to create a vector test file for use with the Altera design environment. Outputs from the design-under-test can then be obtained, either from a simulator (based on the compiled AHDL design) or an actual FPLD implementation. The analysis tool uses the original test-input file and the simulator output file to convert the binary information back into an array of real values. Options may be set to format this output into a Matlab (.m) file that produces a two-dimensional or three-dimensional plot when called from within Matlab.

### 3.6 Special Purpose Synthesis Techniques

The neural network is described in a text file, of which an example is given in table 3.4. Figure 3.3 gives the structure of the ANN that the example describes. The first four
Figure 3.2: Hierarchical structure of a synthesised stochastic neural network.

Table 3.4: Input file describing the XOR neural network. Activation type ‘0’ means that the activation function is to be implemented using a formula. A correlation of ‘0’ indicates that there is to be no address line sharing.

Lines of the file describe general features of the ANN. The first line gives the number of neurons in the network. The second line describes the type of activation function implementation to use: single LUT, multiple LUT, or formula based. Next, control over the amount of address line sharing is given. The more synapses that share the same address lines when generating stochastic numbers, the more correlation there will be in the network, and the less accurate its results. Lastly, the stochastic number mapping is given: unipolar or bipolar.

The remainder of the description file describes the neuron interconnections and weights. Each row describes a neuron and its connections. Starting from the left-hand column, the entries dictate a neuron’s identification number and whether it is an input, output, or hidden neuron. The next column gives the neuron’s bias and the
remaining columns (numbered 0 to \( n - 1 \)) give the input weight from the other neurons whose identification number is equal to the column number. Input neurons are slightly different in that a 1.00 appears in a column to indicate which physical network input they relate to.

The special purpose synthesis techniques are only applicable to a small range of circuit implementations such as stochastic neural networks. They provide a means of exploiting high level knowledge of the problem to allow more optimal circuits to be synthesised—in terms of hardware resource usage and time for the circuit to compute its results. The three major techniques used in the synthesiser are presented in the following subsections.

### 3.6.1 Neuron Activation Function

There are various methods for generating a sigmoidal activation function in hardware. The methods dealt with in this chapter arise from the properties of stochastic bit streams and are based on a simple threshold function that performs the summation of the synapse outputs as well as the sigmoidal activation function[7]. Assume there are \( m \) synapse outputs, \( X_0, X_1, \ldots X_{m-1} \). The sum of these outputs on clock cycle \( t \) is \( s(t) \). The output of the neuron is \( Y \), which is obtained by applying \( s(t) \) to the activation
3.6 Special Purpose Synthesis Techniques

Figure 3.4: Continuous sigmoidal activation function implemented using only a discrete Boolean algebraic expression. All inputs are assumed to share the same probability.

function $A$:  

$$Y(t) = A(s(t)) = A \left( \sum_{i=0}^{m-1} X_i(t) \right)$$  \hspace{1cm} (3.12)

where

$$A(s(t)) = \begin{cases} 1 & \text{if } s(t) > \frac{m}{2} \\ 0 & \text{otherwise} \end{cases}$$  \hspace{1cm} (3.13)

This threshold function outputs a one if the number of active inputs is greater than half the total number of inputs. This simple logic function can be easily implemented in hardware, and provides a continuous activation function based on the statistical distribution of the input bits. Figure 3.4 shows a sigmoidal activation function generated by this method. Representing the random vector of synapse outputs $[X_0, X_1, \ldots X_{m-1}]$ as $S$, the number of ones in $S$ is $s(t)$ on the given clock cycle $t$. Assuming all inputs $X_i$ to the threshold function have the same generating probability $p$ then $S$ will have a binomial distribution. The probability that $S$ will have $k$ bits set is:

$$P(s(t) = k) = \binom{m}{k} p^k (1 - p)^{m-k}$$  \hspace{1cm} (3.14)

where

$$\binom{m}{k} = \frac{m!}{k!(m-k)!}$$  \hspace{1cm} (3.15)
The probability value of the output $Y$ is:

$$P(Y(t) = 1) = A(s(t)) = \sum_{k=\frac{m}{2}}^{m} \binom{m}{k} p^k(1-p)^{m-k}$$  \hspace{1cm} (3.16)

The output of the threshold function clearly depends on the length, $m$, of the vector $S$, which is equal to the number of neuron inputs. The more inputs to the threshold function, the more non-linear it becomes, with a steeper transitional phase.

All inputs to the neuron do not have the same probability, so equation 3.16 is only an approximation assuming that they all share the same probability. They are, in fact, random variables with separate probabilities. The actual function is given in equation 3.17 and it has been shown that the actual function is slightly more non-linear than its approximation[7].

$$P(Y(t) = 1) = A(s(t)) = \sum_{j=0}^{2^m-1} A(j) \prod_{i=0}^{m-1} P(X_i = \Phi_{ji})$$  \hspace{1cm} (3.17)

where $j$ is treated as a binary number and

$$\Phi_{ji} = \begin{cases} 
1 & \text{if bit } i \text{ in } j \text{ is set} \\
0 & \text{otherwise} 
\end{cases}$$  \hspace{1cm} (3.18)

and

$$A(j) = \begin{cases} 
1 & \text{if } \frac{m}{2} \text{ bits in } j \text{ are set} \\
0 & \text{otherwise} 
\end{cases}$$  \hspace{1cm} (3.19)

The threshold function may be implemented and approximated in different ways to alter the resources used. The three methods utilised by the synthesiser in this chapter are:

**Formula:** This version of the activation function is the stochastic threshold calculation given above. It relies upon the AHDL compiler to implement it so that it makes efficient use of hardware. With a large number of inputs, the formula demands excessive on-chip resources.

**Single Lookup-Table:** This is functionally equivalent to the formula and should theoretically give exactly the same implementation results as the formula version. It consists of a lookup table with the same number of inputs as neuron weights. If the number of neuron weights is greater than the number of inputs to a logic element then the lookup table must be partitioned into multiple logic cells by the AHDL compiler. As with the exact formula, problems occur when there are many inputs. In this case, the large lookup tables (with perhaps more than 50 inputs) cannot realistically be implemented.
Multi-level Lookup-Table: This is a compact approximation to the activation function. It was originally presented for use in LUT based FPLDs[7], although as shown in this chapter, it is also useful in non-LUT-based systems. The idea behind this method is to have multiple layers of LUTs whose number of inputs matches the number of inputs in the target device's logic cells. For example a 9-input function can be approximated with four 3-input LUTs in two levels; three in the first, one in the second. This uses 32 bits (4 · 2^3) to define the function instead of 512 bits (2^9). The drawback of this method is that it is only an approximation, and large deviations from a binomial activation function can occur (depending on the distribution of its inputs).

3.6.2 Weight Generation

The synaptic weights are used for multiplication with a stochastic stream that is one of the inputs to a neuron. The weights must be stochastic numbers to enable stochastic multiplication. Storage of weights in their stochastic form is inefficient, and it is preferable to generate statistically different streams each time they are used. It is also preferable to store the weights locally to allow scaling to larger neural networks without heavy routing requirements.

One compact method is to store the stochastic value in an x-input LUT as a binary fraction with 2^x bits after the binary point. If the inputs (address lines) to the LUT are random with certain probabilities of each bit being a ‘1’, then the output of the LUT will be a stochastic stream having a numerical value equal to the binary weight. In the case of a real value stored in a 3-input LUT (8 bits), the address lines are (A_2, A_1, A_0). To generate the appropriate stochastic stream, A_2 must have a probability of each bit being a ‘1’ of 2/3. Similarly, A_1 and A_0 must have probabilities of 4/5 and 16/17 respectively.

The address line streams for such a purpose can be generated via the compact circuit, a bit-stream modulator element, shown in figure 3.5(a)[97]. A series of bit-stream modulator elements are cascaded together and provided with a binary value representing the weighting to give the stochastic stream. Also, uncorrelated PRBSs are provided as carrier inputs as shown in figure 3.5(b). This non-correlation is effected by feeding the PRBS to the modulator elements in the opposite direction to their flow[97] (leftwards in the diagram with a one-cycle delay between each modulator element carrier input). The synthesised VHDL for implementing the stochastic bit stream weighting is given in appendix A.

Uncorrelated address line streams to be provided to multiple LUTs can be efficiently
Figure 3.5: Circuit for the generation of a weighted stochastic bit streams. (a) Bit-stream modulator element. (b) Series of modulator elements with stream weight as input. Each modulator element expects a PRBS carrier input not correlated with any of the others. The binary weights are the fractional binary portion of a number, which is stored in a LUT.

generated using address-line sharing via simple delays as described on page 36. This address-line sharing is another of the specialised synthesis techniques only applicable to stochastic circuits.

3.6.3 Weight Multiplication

The multiplication of a neuron input stream by a weight is trivial once the weight has been expanded into a stochastic bit stream. The multiplication simply requires a two input logic function as discussed in section 3.3; an XNOR for bipolar streams and an AND for unipolar streams. This inherent property of stochastic numbers produces a great reduction in hardware resource usage for multiplication-intensive circuitry.

3.7 Synthesised Neural Network Case Study

Both the device simulation and operation of the an actual FPLD produce output files that are often larger than 50 Megabytes. This is an artefact of using long stochastic bit-streams. Also, the device simulation uses transmission delays that are very close to those found in the target FPLD. This in turn means that salient features in the output
<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output $Q = A \oplus B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 3.5: Truth table for the bipolar XOR (eXclusive-OR) function: $Q = A \oplus B$

files have extra entries at times that do not coincide with the desired clock transitions. These help to increase the output file size even further, and have to be accounted for during the parsing of it.

This section follows through an example of the development of a bipolar eXclusive OR function (XOR). A -1 represents a logic low signal and a +1 represents a logic high signal. The truth table for this function is shown in table 3.5. Note that the stochastic numbers within the circuit represent a range of fractional numbers between -1.00 and +1.00.

The first stage of the design was the derivation of the example input weight matrix file for the synthesiser. The weight file for the XOR example is given in table 3.4. This input file describes the network shown in figure 3.3. The neural network synthesiser used this input file to synthesise all of the VHDL necessary to be compiled to configure an Altera FPLD to perform the desired functionality.

The neural network analyser tool was then used to prepare the vector test file for operational testing of an actual configured FPLD. Real-valued inputs were given to the analyser in a text file. This was converted into a test vector file to be used in the Altera testing and simulation environment. The operation of an actual FPLD connected to a PC resulted in an output file being created by the Altera environment. When the output file was parsed by the analysis tool it produced a Matlab file that created the mesh shown in figure 3.6, confirming the correct circuit operation.

The synthesiser produced code that performed the desired neural network function when compiled and implemented on an FPLD. The time to move from design to implementation was much less than if code had been produced manually, even from standard libraries. Furthermore, there is less chance of a typographical error being introduced when much of the processing is being done by the synthesiser. When errors were made during the initial network description, it could be fixed and a new network could be readily re-synthesised. The analysis tools also minimised the manual work needed when testing the designs.

A wide sample space was sought so as to quantitatively compare the effects on com-
pile time and implementation resource usage that address line sharing and activation function type have. Unfortunately, the time taken for the Altera compiler to generate the FPLD configuration from the synthesised AHDL becomes a problem as more neurons are added. One particular example is of a 10-neuron (110-synapse) network with formula-based activation function taking 18 minutes to compile. The resources required were 895 to 1154 logic cells, depending on the target FPLD family. A similar 3-neuron, 12-synapse network requiring 157 to 169 logic cells took 2 minutes to compile. Other activation functions took much longer—in the order of tens of hours. It was, however, determined that multiple 3-input LUT activation functions required the fewest logic elements in the actual FPLD implementation.

The data processing throughput of the circuit is mainly dependent upon the chosen lengths of the stochastic bit streams. This, in turn, is determined by the desired calculation accuracy of the neural network.

### 3.8 Conclusions

This chapter has shown how a high level design tool can assist a designer in the implementation of hardware neural networks. The internal workings of FPLDs are abstracted from the designer, thus broadening the appeal of programmable logic. Moreover, specialised knowledge can be included during the synthesis process to produce a design that meets desired criteria. Described in this chapter, the use of stochastic numbers was chosen to reduce routing and hardware resource requirements. The resulting configuration, produced from synthesised AHDL, correctly performed the required neural
network function.

The inclusion of options to optimise circuit density gave rise to a powerful neural network design tool. The analysis tool provided an efficient method for assisting in the confirmation of an SNN's performance. It allowed designs to be tested at an operational level (with real numbers) rather than at an abstract bit level. Application specific synthesis worked better than general high level tools because solutions could be tailored to match the problem and particular FPLD technology. For example, an approach such as the conversion of C algorithms directly into HDL might not take advantage of stochastic numbers and their activation function implementation features.

The main drawback of using stochastic numbers is the time to process such long serial bit streams. In the case where time rather than space is the greatest concern, options should be given to control the numerical representation used within the neural network. Another hindrance was the length of time required to compile large neural networks after AHDL synthesis. Methods that utilise high level knowledge of the structure and layout of neural networks to assist the compiler should also be looked at, so as to reduce the post-synthesis compile time.

It is envisaged that a greater proportion of hardware design in the future will be done at the higher levels, as has been the trend with software development. Therefore, it is suggested that different methods be analysed to allow designers to give their descriptions of circuit functionality at various levels and with varying control over the synthesis process. One possibility is the use of synthesised application-specific execution units on a general purpose FPLD based microprocessor core[76]. These execution units would perform functions of varying complexities, and in manners that exploit the use of configurable logic and the fundamental parallelism of hardware. The use of a configurable core allows the efficient conversion of sequential algorithms and specific hardware blocks into a solution that combines both hardware and software. In the case of a neural network, each execution unit could be a neuron, which passes its result back to a sequential instruction loop. Furthermore, there can be many of these neurons operating in parallel, thus speeding up the network. Another area that could provide interesting results is the utilisation of a formal language for the description of artificial neural networks, such as that presented by Smith[88]. This would allow the further assimilation of software and hardware development tools.
Chapter 4

Synthesis Using Synchronous Modules

4.1 Introduction

The synthesiser described in chapter 3 was developed as a special purpose unit for the synthesis of stochastic neural networks. This chapter presents a general purpose synthesiser for the development of a much broader range of circuits[56, 54, 57]. The aim of the synthesiser is to take a high-level functional description of a circuit and produce an implementation of the circuit in AHDL. This intermediate language output may be further processed to physically instantiate the circuit on an FPGA device. Synthesis parameters may be chosen to guide the resulting circuit towards the planned hardware resource usage and circuit processing time constraints. The synthesiser has been designed to be used in conjunction with an optimisation algorithm that adjusts the synthesis parameters. This is done so that different design alternatives may be analysed and optimised for these desired criteria.

The types of circuits targeted by the synthesiser are those that are largely data-path oriented, with clearly defined functional blocks. Ideal applications would be the implementation of independent units to be controlled by a separate micro-controller, or circuits such as digital filters that benefit from a high degree of parallelism.

This chapter firstly defines the design entities used by the synthesiser, then specifies a global view of the synthesis process and describes its parts in more detail. An example synthesised circuit is then presented. Finally, the context for further improvements is discussed. This includes the use of the synthesiser as a part of an optimiser that searches throughout circuit design spaces to find circuit implementations that satisfy design criteria.
4.1.1 Design Entities

The circuit to be synthesised is described by its *functional description*, which is in the form of a netlist. The netlist can be created with the graphical design entry tool developed as part of this research. The netlist describes the circuit as a number of interconnected synchronous modules. The modules are synchronous as they are all clocked together and have their data flow controlled by a common *control unit*, which is also automatically synthesised. Each *synchronous module*, or simply *module*, is an abstraction of a circuit that performs a certain function, and can be implemented in various ways depending on its *module parameters* that are chosen by the synthesiser (or optimiser). In addition to describing the function of each module, the netlist also indicates the *accuracy*, or number of bits of resolution required. The part of the circuit that data propagates through is the circuit’s *data path*.

There are two different types of modules used by synthesiser: functional and auxiliary. *Functional modules* perform certain, predefined functions, such as ‘multiply’ or ‘filter’, and are used in the circuit’s functional description. *Auxiliary modules* are used and generated by the synthesiser in its task of ensuring the circuit performs correctly. Examples of auxiliary modules are accuracy converters, delay modules and modules for converting between *data types* (such as parallel, serial or stochastic).

Each *module implementation*, which can be described by various and multiple methods (such as HDLs and graphical design entry), has an interface to the synthesiser in AHDL known as the *module interface shell*. The module interface shell represents an unambiguous way of describing the module implementation. Different module implementations reflect the different ways that the module may be implemented, using different algorithms or design paradigms. A *module instance* is an unambiguously defined module; one that has been supplied with all its module parameters. These *module parameters* include the data type of the module, and the data path widths. Lastly there is the *physical module instance*, which is the actual on-device implementation. This differs from the module instance because, for example, a number of identical module instances can utilise *resource sharing*, and actually use the same hardware, or physical *module instance*, to perform their function. Figure 4.1 presents a hierarchical view of the design entities used in the design of complex data paths.

4.1.2 Synthesiser Outline

The input to the synthesiser is a netlist of interconnected functional modules, a library of *predefined module implementations*, and a group of *module generators*. These module generators have the ability to synthesise the HDL for functional modules (*synthesised*
modules) for a range of different parameters.

The output of the synthesiser is a design project consisting, at its highest level, of a synthesised data path and control unit, which are described in AHDL. The module instances that make up the resulting circuit are predefined AHDL files, AHDL and supporting files that are created by the module generators, or predefined design files created with any of the supported tools. A global view of the synthesiser is given in figure 4.2. Table 4.1 shows the major stages performed during synthesis.

4.2 Synchronous Modules

Synchronous modules are the fundamental building blocks of the synthesiser. Each module is an abstraction of a functional circuit, for which the appropriate module
Step 1  Load input data. A new design is described using graphical design entry with functional modules and their interconnections.

Step 2  Verify circuit description. The circuit is analysed to ensure all modules are available and are able to be synthesised.

Step 3  Repair circuit description. A repair stage in the synthesis algorithm ensures that data flows through the circuit without incompatibility between modules—such as differing data types or data-path widths.

Step 4  Analyse dataflow. The flow of data through the circuit and down feedback paths is determined, enabling further repair to be performed to ensure data flows correctly through the circuit.

Step 5  Perform pipeline analysis. Pipeline analysis of the resulting circuit is done to maximise the rate at which the circuit can be initiated.

Step 6  Calculate circuit parameters. These parameters describe the hardware resource usage and processing time of the circuit as well as information enabling the circuit to be used as a module in a hierarchical design.

Step 7  Determine control signals for control unit. Calculate the required functionality of the control unit.

Step 8  Synthesise output files. Once produced, these files are processed by an FPGA synthesiser to produce a resulting FPGA configuration file.

Table 4.1: Major stages performed by the synchronous synthesis algorithm.

instance is determined when all the module parameters have been defined. Each module instance consists of a synchronous FPLD circuit design, that may be designed using any design entry tool that the Altera Max+Plus II environment supports. At its highest level, the module has a standard interface shell that is written in AHDL in a Text Design File (TDF). The main assumption with each module is that it will be operated by a global clock at a clock frequency low enough not to have to consider propagation delays over and above the whole number of clock cycles for the module's operation. The format of signed numbers is two's complement.

The standard module AHDL interface shell gives the synthesiser the ability to interconnect modules that are designed in a number of formats. All inputs and outputs of a module are assumed to be of the same accuracy and data type and, therefore, bus width. A module such as a multiplier, which operates on two input buses and results in a single output bus twice as wide, is modified to fit this criterion. The interface shell may either define the multiplier's inputs as the same width as the output (where the top input bits are unused), or its output as two separate outputs each as wide as each input. These two different multiplier circuits will be distinctly named functions.

A graphical representation of the module inputs and outputs is given in figure 4.3. In addition to the standard inputs and outputs, there are optional global clock and clear signal inputs. There are control lines to provide control signals to subcircuitry
within the module. The are also external inputs that allow data, such as constant values, to be provided to subcircuitry within the module.

A module can be made up of any number of design files in a hierarchical structure. The files supported (with their filename extensions given in the form .xxx) include AHDL files (.tdf), VHDL files (.vhd), Verilog HDL files (.v), state machine design files (.smf), graphical design files (.gdf), waveform design files (.wdf), Xilinx netlist format files (.xmf), OrCad schematic files (.sch), and EDIF (Electronic Design Interchange Format) files (.edf). The AHDL files can utilise the latest features of the Altera compiler\(^1\) including Macros, Functions, Megafunctions and its Library of Parameterised Modules (LPM)\(^4\). Lastly, memory initialisation files (.mif) and Intel-format hexadecimal files (.hex) are supported for the definition of memory contents to be instantiated in an FPLD. When multiple files are required for a module, they are listed in the parameter section of its interface shell so that they are automatically copied to the output directory during synthesis. The ability to support all of these design formats makes it possible to convert existing FPLD designs into modules compatible with the synthesiser. Figure 4.4 gives an overview of the components produced and used by the synthesiser.

The parameters given in the module interface shell, which is an AHDL text design file, completely describe each module to the synthesiser. They are placed in an AHDL comment so that they are ignored by the compiler. Table 4.2 provides an example

\( ^1 \)Currently MAX+plus II Version 9.11
of the module interface shell’s parameter section. A more detailed description of the interface shell parameters can be found in section B.3.1 of appendix B. The parameters are calculated for a specific target FPGA family; in this case the Altera 10K family of devices. These devices have logic elements and memory cells, which are treated as separate resources by the synthesiser. The logic elements are contained within Logic Array Blocks (LAB), while the memory cells are contained within Embedded Array Blocks (EAB).

The control signals required for each module are described in, but not implemented within, each module because they are generated in one place—in a separate control circuit. This increases the demand on routing resources, but allows a more efficient implementation of the control unit by eliminating the generation of identical control signals in multiple places in the circuit. Moreover, if a binary encoded state machine is used for the control unit, it is usually the case that combining the control units into one state machine will result in a more compact solution. The synthesised circuit contains a separate controller that can be instantiated on-chip with the synthesised data path, or ignored if the synthesised circuit is going to be used in a higher-level design. In the latter case, the synthesised circuit describes what control signals are required, and when, for implementation in an overall circuit controller. Each module’s control signals are described by binary vectors that indicate whether each control line should be low or
<table>
<thead>
<tr>
<th>Parameter line in module interface shell</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Begin AHDL comment</td>
</tr>
<tr>
<td>[function]=multiply;</td>
<td>Function (how it is referenced by a designer)</td>
</tr>
<tr>
<td>[description]=Standard multiplier;</td>
<td>Textual module description</td>
</tr>
<tr>
<td>[10k_cells]=100;</td>
<td>Number of logic cells used</td>
</tr>
<tr>
<td>[10k_mem_cells]=220;</td>
<td>Number of memory cells used</td>
</tr>
<tr>
<td>[cycles]=7;</td>
<td>Compute time (in cycles)</td>
</tr>
<tr>
<td>[inputs]=2;</td>
<td>Number of input buses</td>
</tr>
<tr>
<td>[outputs]=1;</td>
<td>Number of output buses</td>
</tr>
<tr>
<td>[clocked]=y;</td>
<td>Global clock signal required</td>
</tr>
<tr>
<td>[clrn]=y;</td>
<td>Global clear signal required</td>
</tr>
<tr>
<td>[parameters]=0;</td>
<td>Number of design parameters (required in netlist)</td>
</tr>
<tr>
<td>[collision_vector]=xx..x.xx;</td>
<td>Collision vector (also describes latency)</td>
</tr>
<tr>
<td>[associated_files]=mult.hex,m.tdf;</td>
<td>Comma-separated list of other files required</td>
</tr>
<tr>
<td>[can_share]=y;</td>
<td>Whether the module can share its resources</td>
</tr>
<tr>
<td>[indatatype]=ser;</td>
<td>Data type of input buses (par=parallel, ser=serial etc.)</td>
</tr>
<tr>
<td>[inaccuracy]=8;</td>
<td>Bits of resolution of the input buses</td>
</tr>
<tr>
<td>[inbuswidth]=1;</td>
<td>Bus width of the input buses</td>
</tr>
<tr>
<td>[outdatatype]=ser;</td>
<td>Data type of the output buses</td>
</tr>
<tr>
<td>[outaccuracy]=8;</td>
<td>Bits of resolution of the output buses</td>
</tr>
<tr>
<td>[outbuswidth]=1;</td>
<td>Bus width of the output buses</td>
</tr>
<tr>
<td>[converter]=n;</td>
<td>Are the input and output data types different</td>
</tr>
<tr>
<td>[external_lines]=10;</td>
<td>Number of external control lines (from outside circuit)</td>
</tr>
<tr>
<td>[external_pulses]=2;</td>
<td>Unique external control pulses (multi-line, single-cycle)</td>
</tr>
<tr>
<td>[ext_control0]=0-3.0;</td>
<td>External control pulse #0 (LineFrom-LineTo,Cycle)</td>
</tr>
<tr>
<td>[ext_desc0]=Control line A;</td>
<td>Textual description of external control line #0</td>
</tr>
<tr>
<td>[ext_control1]=4-9.0;</td>
<td>External control pulse #1 (LineFrom-LineTo,Cycle)</td>
</tr>
<tr>
<td>[ext_desc1]=Control line B;</td>
<td>Textual description of external control line #1</td>
</tr>
<tr>
<td>[control_lines]=1;</td>
<td>Number of control lines from control unit</td>
</tr>
<tr>
<td>[control_pulses]=1;</td>
<td>Unique control pulses (single-line, multi-cycle)</td>
</tr>
<tr>
<td>[control0]=0,1-2;</td>
<td>Control pulse #0 (Line,CycleFrom-CycleTo)</td>
</tr>
<tr>
<td>%</td>
<td>End AHDL comment</td>
</tr>
</tbody>
</table>

Table 4.2: Parameter section of the module interface shell file, with example values given.
high for each cycle after the module is initiated. If a module requires external control
signals or inputs, these are similarly described. These external controls are assumed to
be implemented outside the entire synthesised circuit project and applied at the correct
times to the external control line inputs.

4.2.1 Functional Modules

The functional modules are interconnected in the input netlist to describe the circuit's
functionality. They each operate on their input data and provide the resulting output
data a certain time later. When a module (or circuit) is supplied with data to process,
it is known as an initiation. A module may have internal data storage between initiations
and there are no restrictions on its internal operation. The module may even be
a complete circuit, possibly including a processor core. At the most abstract level, the
module is described by its function and accuracy alone, a '12-bit signed multiplier' for
example. During synthesis, a module implementation is chosen for each module, such
as '12-bit signed multiplier using a lookup table'. Subsequently, the module implementa-
tion is narrowed to its actual instance, such as 'Serial 12-bit signed multiplier using
a lookup table'. The instance must either exist as a predefined design with a module
interface shell in AHDL, or be able to be synthesised by a module generator during the
synthesis process.

4.2.2 Auxiliary Modules

The auxiliary modules are used in the repair stages of the synthesis to ensure correct
data flow and compatible data types throughout the circuit. They may be used by
a designer if the designer has a understanding of, and need to control, the low-level
operation of the circuit. The auxiliary modules are synthesised during the synthesis
process, and can be produced for accuracies from 2 to 999 bits, for serial and parallel
(the more common) data types. The modules are:

Data type converters: These convert data types from serial to parallel and vice
versa.

Accuracy converters: The accuracy converters convert data from a lower to a higher
accuracy or vice versa. When converting to a lower accuracy, the high order
bits are lost. When converting to a higher accuracy, the high order bits are
generated—so that the represented number is the same in both accuracies. In
both cases, it is ensured that signed and unsigned numbers retain their integrity.

Delay modules: The delay modules are made from D-type Flip-Flops (DFF) with
4.3 Synthesis Variables

`enable` inputs. If the circuit is only required to operate for a single iteration then non-pipelined delays are used. In the parallel case, a single row of DFFs latch the data and store it until it is required. In the serial case, if the delay ($d$) is less than the accuracy ($a$) of the data then there are $d$ serially connected DFFs, otherwise there are $a$ serially connected DFFs that latch when the input data has been fully clocked in. If the circuit is to be used for multiple iterations, pipelined delays are added to allow a lower average latency. In this case, there are enough delay elements to pipe-in a new input every clock cycle: $d \cdot a$ elements for the parallel case and $d$ for the serial.

**Multiplexers:** A multiplexer is used on each input of a shared module instance. The inputs that originally went to the corresponding inputs on the shared modules are fed as inputs to this multiplexer. The inputs are applied to the module instance at times determined by the control unit.

**Bus splitters:** The bus splitter module provides the designer with a method of controlling bus interconnections within the circuit at bit-level. These splitters can join any number of varying width buses and placeholder bits into one bus and then break it again into any number of different, varying width, buses and placeholder bits. This capability is required for operations such as combining two bytes into a 16 bit word, or when required to convert the accuracy of a 16 bit number into an 8 bit number, but using the most significant bits instead of the least significant. The bus splitters support parallel and serial data types.

**Inputs and outputs:** These are purely virtual modules. Their parameters are used in timing and resource usage calculations, but they are not synthesised as they are inherent parts of the circuit.

4.3 Synthesis Variables

There are a number of variables associated with each module of the circuit. They are chosen by the designer, or by the optimisation algorithm controlling the synthesiser. They affect the overall hardware resource usage and the time taken for the circuit to process its inputs (`compute time`). Also affected is the `average latency` of the circuit, which describes how many cycles (on average) must pass before new inputs can be processed by the circuit. Often this is given as the number of completions (or initiations) per cycle, known as the `initiation rate` which is the reciprocal of the average latency. The combination of hardware resource usage and timing gives a solution point for the circuit in the possible design space of functionally equivalent circuits. Note that
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Module Version</th>
<th>Size (Logic Cells)</th>
<th>Size (Memory Cells)</th>
<th>Compute Time (Cycles)</th>
<th>Average Latency (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>A</td>
<td>155</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>parallel</td>
<td>B</td>
<td>135</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>parallel</td>
<td>C</td>
<td>137</td>
<td>0</td>
<td>2</td>
<td>1.00</td>
</tr>
<tr>
<td>parallel</td>
<td>D</td>
<td>163</td>
<td>0</td>
<td>3</td>
<td>1.00</td>
</tr>
<tr>
<td>parallel</td>
<td>E</td>
<td>176</td>
<td>0</td>
<td>4</td>
<td>1.00</td>
</tr>
<tr>
<td>parallel</td>
<td>F</td>
<td>64</td>
<td>7168</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>serial</td>
<td>A</td>
<td>65</td>
<td>0</td>
<td>22</td>
<td>33.00</td>
</tr>
</tbody>
</table>

Table 4.3: The effect of different synthesis variables on an 11 bit unsigned multiplier module. Note that the module version is in relation to its data type and so is not necessarily unique.

The actual duration of a cycle is determined by the worst-case combinational circuit propagation delay.

Altering the module variables allows different solutions to be analysed. Generally, there will be a tradeoff between hardware resource usage and circuit compute time. A more subtle tradeoff can be obtained between circuit compute time and average latency. An optimisation algorithm can use these tradeoffs to search through the design space for a solution that conforms to specified criteria.

The variables that can be modified for each module are its implementation version (the algorithm or structural model it is based on), the data type it operates on, and whether the implementation should share hardware resources with other modules that perform the same function. The minimum accuracy of a module is a design decision and cannot be reduced. Increasing it does not usually have an advantageous effect, so it is not considered to be one of the variables. Table 4.3 shows the resulting module parameters for an 11 bit unsigned multiplier for different synthesis variables.

### 4.3.1 Module Implementation Version

The module implementation version separates functionally identical modules that are implemented in different ways. For example, a multiplier can be implemented using a shift-and-add regime or it can use a lookup table. For each case, different kinds of hardware resources are required (logic cells and memory cells). Additionally, the compute time is different. The different versions are used in the circuit optimisation stage. They can be described in separate TDF files for input to the synthesiser or within the module generators that are described in section 4.6.
Figure 4.5: Different data type representations for 6 bit numbers. L and H indicate the low order bit and high order bit respectively for each data type.

Each module implementation version is assigned a unique version number at start-up of the synthesiser. These version numbers have no meaning between initiations of the synthesiser application. Therefore, it means nothing to store this information. For example, version one of a parallel multiplier might be the shift-and-add implementation during one run and not exist during a subsequent run. The dynamic allocation of version numbers means that the module library can have modules added or removed without the synthesiser making assumptions of what versions are available.

4.3.2 Module Data Type

In general, it is the data type of a module that has the biggest effect on its compute time, hardware resource usage and average latency. The current module library consists mainly of parallel and serial data types, with a few stochastic data typed modules. The synthesiser has been written to allow extra data types to be added. Having more data types offers the opportunity to develop circuits that comply more closely to desired circuit design constraints. Data types that are combinations of serial and parallel representations offer finer tradeoffs between time and hardware resource usage. Example data types could be 16 bits wide for 2 cycles, representing a 32 bit number, or 4 bits wide for 8 cycles representing the same number. An example of various data types being used to represent a 6 bit number is shown in figure 4.5.

Data type tradeoffs usually allow a larger module to perform its function in less time than a smaller module, although this is not always the case. Determining the actual resource usage for a given module depends on the topology of the circuit that contains the module. Data type converters must be factored into the calculation, as must delay modules and multiplexers. The resource usage for these items is greatly influenced by
bus width. The effect on the average latency of the circuit and its compute time must be determined for the circuit as a whole. The features of the circuit are affected in a complex way when a module's data type is altered. The choice of a data type that has a smaller, faster module, can end up producing a larger, slower overall circuit.

Special data types, such as stochastic representations, have additional requirements as discussed in chapter 3. Stochastic streams require the implementation of pseudo-random number generation within the circuit. In addition, the correlation of the generated stochastic streams must be kept as low as possible. The advantage gained in using them is that signed multiplication of two numbers can be performed in a 2-input lookup table with a compute time of 0 cycles. The related disadvantage is that the bit streams required are very long, such that the average latency of the very small multiplier can be in the order of thousands, if not tens of thousands, of cycles.

4.3.3 Module Resource Sharing

Resource sharing is performed by replacing two or more similar modules in a circuit with one module that has its inputs multiplexed to represent the original inputs of the modules. This trades circuit size against compute time because the shared modules can not be used in parallel. An example is given in figure 4.6. It can often be the case, especially for wide parallel data paths, that the resource usage of the multiplexers that perform the resource sharing, combined with the additional interconnect requirements, is greater than the savings made.

The functions performed by the modules being shared must be identical, as must their data type. The accuracy of the physical module instance must match the greatest accuracy of the shared modules. This can introduce the need for extra accuracy conversion modules. The dataflow analysis stage of the synthesiser handles the resource-shared modules slightly differently to other modules (described further in section 4.5). In addition, the controller circuit has to produce the extra control signals required to drive the multiplexer.

Some modules cannot share their resources. This is inherent in the module design and is set by the module designer. Any module that records information about its state in a prior circuit iteration will operate incorrectly if it is repeatedly used in a given circuit iteration. An example situation occurs with an accumulator module that adds the current input to a sum of all prior inputs. If two or more of these accumulator modules are implemented in a shared resource then the accumulated sum will be incorrect—as there is only one physical set of memory elements being shared.

Table 4.4 shows the effects of module resource sharing in two circuits using parallel
multpliers. Table 4.5 shows the effects of resource sharing when serial multipliers are used for the same circuits. The two circuits are:

1. Two independent 16 bit multipliers, each taking two 8 bit inputs to produce a 16 bit result.

2. Three multipliers. Two of them take 4 bit inputs to produce 8 bit results. The third takes the two 8 bit results as inputs to produce a 16 bit product.

Note that the reported logic cell usage takes into account the inclusion of accuracy converters, multiplexers, pipelined delays, and the controller circuit. The given sizes are obtained from compilation of the HDL and fitting the design into an Altera 10K FPLD. It is assumed that the inputs to each of the circuits are of the same data type as the multipliers to give a fair comparison of resource sharing. From the tables, it is apparent that the tradeoff between cycles and cells is not always present; often both parameters increase. This makes design decisions about using resource sharing difficult.

### 4.4 Synthesis Algorithm

The synthesis algorithm produces all the HDL files required to compile the design for implementation on an actual FPLD device. The algorithm is given in table 4.6. The
<table>
<thead>
<tr>
<th>Physical Multiplier Instances</th>
<th>Compute Time (Cycles)</th>
<th>Size (Logic Cells)</th>
<th>Average Latency (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two-multiplier parallel circuit:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>707</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>421</td>
<td>2.00</td>
</tr>
<tr>
<td>Three-multiplier parallel circuit:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>352</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>298</td>
<td>2.00</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>459</td>
<td>1.60</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>464</td>
<td>3.20</td>
</tr>
</tbody>
</table>

Table 4.4: Circuit parameters resulting from resource sharing modules in two-multiplier and three-multiplier parallel circuits. No memory cells (for use as look-up tables) are used by the parallel multipliers. The average latency is for five circuit iterations.

<table>
<thead>
<tr>
<th>Physical Multiplier Instances</th>
<th>Compute Time (Cycles)</th>
<th>Size (Logic Cells)</th>
<th>Average Latency (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two-multiplier serial circuit:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>209</td>
<td>48.00</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td>262</td>
<td>96.00</td>
</tr>
<tr>
<td>Three-multiplier serial circuit:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>210</td>
<td>48.00</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>252</td>
<td>48.00</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>218</td>
<td>96.00</td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td>316</td>
<td>144.00</td>
</tr>
</tbody>
</table>

Table 4.5: Circuit parameters resulting from resource sharing modules in two-multiplier and three-multiplier serial circuits. No memory cells (for use as look-up tables) are used by the serial multipliers. The average latency is for five circuit iterations.
input netlist describes the functionality of the circuit, suggests data types of modules to use if available, and recommends which modules should share their resources. The netlist also indicates the overall data type and accuracy of the circuit, and whether it should operate on signed numbers. For a module that has a signed and unsigned version, either (or both) can be chosen; it does not depend on the whether the overall circuit is signed or not. Other module-dependent parameters are also provided in the netlist. Functionally equivalent modules that have different parameters (such as constants) are synthesised in distinct TDFs, because their exact functionality is unique. In addition, the accuracy of each component module is indicated in the netlist. These cannot be reduced by the synthesiser, even to match a lower overall circuit accuracy, because the internal circuit operation may require the higher accuracy. Therefore, the lower bound on each module’s accuracy is a design decision, and should be kept as low as possible. Module interconnections are also provided in the netlist.

4.4.1 Feedback Paths

The compute times of the abstract modules are not known at the circuit design stage, and so a delay element in the original design has no meaning. Delay elements cannot be used to latch data for use in a subsequent iteration of the circuit, because the synthesiser ensures all input data to a module arrives coherently. A feedback connection is defined for the synthesiser as a connection where the data should arrive at its destination on the correct input cycle for the destination module, but in the following circuit iteration. Connections defined as feedback paths can therefore be used to delay data between iterations, instead of delay elements.

Connections must be designated by the designer as feedback or not; it is not possible for the synthesiser to determine the type of connection. Consider the circuit of figure 4.7(a), which contains two adder modules, X and Y, which have a compute times of x and y cycles respectively, where \( x < y \). It has two paths that could be chosen to be feedback connections and shows how subtly different circuits are produced by different choices of the feedback paths. If the path from Y to X is chosen as the feedback path, then the repaired circuit is shown in figure 4.7(b). The compute time of this circuit is \( x \) cycles and its average latency is \( x + y \) cycles. The function it performs (with respect to its iteration, \( n \)) is:

\[
Q_n = \begin{cases} 
  A_0 & n = 0 \\
  A_n + B_{n-1} + Q_{n-1} & n \geq 1 
\end{cases}
\]  \hspace{1cm} (4.1)

With the feedback path taken from X to Y, the repaired circuit is that of figure 4.7(c). In this case the compute time is \( x + y \) cycles and the average latency is \( x + y \) cycles.
Step 1 Load input data
   1.1 Load library of predefined modules
   1.2 Load library of module generators
   1.3 Load circuit description as a netlist file
Step 2 Verify circuit description
   2.1 Check all modules are known and are able to be synthesised
Step 3 Repair circuit description
   3.1 Choose closest modules to those defined in netlist
   3.2 Ensure data type and accuracy compatibility throughout circuit
Step 4 Analyse dataflow
   4.1 Determine dataflow order of resource shared modules
   4.2 Determine start cycle values for all modules
   4.3 Insert delays in forward path to ensure dataflow coherency
   4.4 Ensure data type and accuracy compatibility in feedback paths
   4.5 Insert delays in feedback paths to ensure dataflow coherency
Step 5 Pipeline analysis of circuit
   5.1 Calculate forbidden latency set
   5.2 Calculate collision vector for circuit
   5.3 Prepare modified state diagram
   5.4 Derive control strategy using greedy cycles
Step 6 Calculate circuit parameters
   6.1 Estimate circuit size
   6.2 Calculate compute time
   6.3 Calculate average latency
Step 7 Determine control signals for control unit
   7.1 Calculate the control signals required for each module on each cycle
Step 8 Synthesise output files
   8.1 Copy required predefined modules and associated files to output directory
   8.2 Synthesise necessary modules and their associated files
   8.3 Synthesise module interconnections
   8.4 Synthesise controller circuit
   8.5 Write report on circuit
   8.6 Write parameters into synthesised circuit to produce a reusable module

Table 4.6: Synchronous synthesis algorithm.
Figure 4.7: The need to define feedback connections. (a) Original designed circuit where modules X and Y are two-input adders. (b) Repaired circuit having chosen Y to X as the feedback path. (c) Repaired circuit having chosen X to Y as the feedback path. (d) Repaired circuit having chosen both paths as feedback paths. Delay modules ‘D’ are added delays for data flow coherency.

The function performed is:

\[ Q_n = \begin{cases} 
  A_0 + B_0 & n = 0 \\
  A_n + B_n + Q_{n-1} & n \geq 1 
\end{cases} \tag{4.2} \]

Using both possible paths as feedback paths, the resulting circuit is shown in figure 4.7(d). The compute time is \( x \) cycles and the average latency is \( y \) cycles. The function it performs is:

\[ Q_n = \begin{cases} 
  A_0 & n = 0 \\
  A_1 + B_0 & n = 1 \\
  A_n + B_{n-1} + Q_{n-2} & n \geq 2 
\end{cases} \tag{4.3} \]

### 4.4.2 Circuit Repair

The first task of the synthesiser is to repair and modify the circuit to compensate for the high level at which the circuit was described. The minimum information provided by a designer is the functionality and accuracy of each module in the circuit and the module interconnections. The modules can be of any accuracy and data type, and no responsibility is placed upon the designer to ensure data flows through the circuit coherently. Figure 4.8 shows the repair of a small example circuit.
Figure 4.8: Circuit repair through module insertions. (a) The original circuit. (b) The circuit after repair by the addition of accuracy converters, data type converters and a delay element.

The most similar available module is chosen for each module in the circuit—the exact requested module may not be available. The chosen module’s functionality and module specific parameters (such as constants or whether signed) must match those of the requested module exactly. Initially, modules that match the prescribed data type but have the same or greater accuracy are searched for. If none are found then other data types are considered and the module with the lowest available matching accuracy is chosen.

When stochastic numbers are used, the random number generator LFSR is required, as are delays to provide uncorrelated address lines for stochastic number production (as described in section 3.3). These do not affect the data flow of the circuit and so are not added to the netlist until just prior to synthesis.

For resource sharing, multiplexers are required and all but one of the modules that have their resources shared must be removed. Connections need to be remade to the multiplexer and shared module. Again, these changes are not made until just prior to synthesis. The modules that are part of each resource sharing group are remembered however, and are treated specially in the dataflow analysis.

Data type converters are added to produce data type compatibility. This ensures that the data at each module input is of the correct data type for the module to operate on. Converters are also added to ensure the data type of each overall circuit output
matches that defined in the netlist. When a number of different data types are used in a design, there may not be a converter available to perform the correct conversion. Moreover, if a converter is available, it might not do the conversion in the most efficient manner—in terms of hardware resource usage. For this reason, a separate algorithm was written that takes all the available converter modules and iteratively determines combinations of converters to provide all possible conversions. It also replaces converters that may have unnecessarily large hardware resource requirements with smaller ones. An example might be a stochastic to serial converter that benefits from being converted to an intermediate parallel representation, requiring two converter modules. These new converters can be implemented and then used by the synthesiser. Warnings are given if there is no means of performing conversions between two data types.

Accuracy converters are inserted to match up module accuracies. These are required for the HDL to eliminate errors from disconnected inputs to modules. From a semantic point of view, they are required to ensure that the sign bit of signed numbers retains its integrity.

Delays to ensure coherent data arrival at each module are subject to the way data flows through the circuit, and are therefore inserted during the dataflow analysis stage.

4.5  Dataflow Analysis

Analysis of the flow of data through the circuit is performed to ensure correct operation of the circuit. From this analysis, control signals for each module are determined and the circuit structure is modified to ensure the data travels through the circuit in a coherent fashion—with each module having all its inputs available at the same time. Lastly, the dataflow analysis provides information regarding the compute time of the circuit and shows how often pipelined initiations of the circuit may be made. The dataflow analysis is the most time consuming algorithm within the synthesiser, taking approximately 45% of the synthesiser’s processing time.

A summary of some of the important dataflow and pipeline terms, based on those given by Kogge[44] is given in table 4.7. More detail is given below:

Collision Vector

In a collision vector, a ‘0’ indicates a permitted latency, and a ‘1’ indicates a collision. There are three different conventions for representing collision vectors:

1. starting with cycle 1 as the leftmost character. (The notation used in this thesis.)

---

Thus, care should be taken when looking up tables by collision vector (such as tables of MALs).
Table 4.7: Summary of the important dataflow and pipeline terms

2. starting with cycle 1 as the rightmost character.

3. starting with cycle 0 as the leftmost character. (There will always be a collision character as the leftmost character.)

For example, a circuit that will produce collisions if reinitiated after latencies of 1, 2, or 6 cycles will have a collision vector of 110001. In this case, initiation is allowed after latencies of 3, 4 or 5 cycles. Note that the compute time may be shorter than the length of the collision vector. For example, the output data could be ready on the fifth cycle even though the circuit cannot be reinitiated for six cycles.

Average Latency

The average latency ($t_{avg}$) is calculated by dividing the duration of the latency cycle by the number of initiations made in the cycle. The true average latency ($t_{avg(n)}$) calculated by the synthesiser is more accurate, in that it is given for a set number of iterations ($n$), and is therefore more appropriate in the analysis of a circuit’s timing. Consider the latency sequence (3, 8, 3, 8, 3, …), which gives a latency cycle of (3, 8) and an average latency of $t_{avg} = (3 + 8)/2 = 11/2 = 5.5$. The true average latency for three initiations would be $t_{avg(3)} = (3 + 8 + 3)/3 = 14/3 \approx 4.67$. The average latency represents the true average latency as the number of iterations approaches
Figure 4.9: Reservation table representation of collision avoidance. The letters X and Y represent hardware usage by a module in the given clock cycle for the first and second initiations of the circuit respectively. A collision occurs when two or more initiations require the use of the same piece of hardware. The circuit that this reservation table describes can be initiated without collisions with a latency cycle of \((2, 3, 5, 3)\).

infinity. The true average latency produced by the synthesiser, because it is not the limit to infinity, can be less than the minimum achievable latency (MAL). References to “average latency” imply \(t_{\text{avg}}\) unless otherwise stated.

**Reservation Table**

A reservation table is a convenient way to describe hardware requirements within a circuit. There is a row in the table for each module, or arbitrary piece of hardware. Each column represents a cycle of circuit operation. An entry appears in the table if the module, or piece of hardware, is being used on the given cycle. Note that although in many cases a module’s row in the reservation table is the same as its collision vector, this is not necessarily the case. Figure 4.9 shows how the reservation table highlights collisions in various parts of the circuit for different initiation latencies.

**Forbidden Latency Set (FLS)**

This is an encoded representation of the reservation table, which describes when hardware usage collisions will occur. It can be derived from a reservation table. Any given reservation table will produce exactly one FLS, whereas an FLS could represent many different reservation tables. This implies that there is a loss of information in encoding a
Figure 4.10: The relationship between collision vectors, forbidden latency sets, and reservation tables. (a) Reservation table showing initiation latencies that cause collisions. (b) The collision cycles from (a) shown as an FLS. (c) The collision vector representing the FLS of (b) by the position of its ones.

resonation table into an FLS. Figure 4.10 shows the relationship between a reservation table, the FLS it describes, and the resulting collision vector.

4.5.1 Feedforward Analysis

Each module has a start cycle, on which all input data is provided to the module and it begins performing its function. This can be thought of as the cycle on which the module is initiated. There is also a cycle on which the output data of a module holds its correct value and is ready for further use in the circuit. This cycle is known as the data-ready cycle.

The dataflow analysis begins with the forward path. All feedback connections are ignored for the calculation of all modules’ start cycles, but are treated separately as discussed in section 4.5.3. If feedback is found that has not been explicitly defined, the process is halted. The start cycles (on which subsequent module initiations will occur) must be calculated for each module to allow a control signal generator to provide module controls at the correct times. Delays are also inserted to bring about data coherency for data travelling through the circuit. These delay modules are added after all start cycles have been calculated, and can be pipelined delays or regular latched delays, depending on whether or not the circuit is to be repeatedly initiated.
4.5 Dataflow Analysis

The feedforward analysis to calculate module start cycles is a recursive procedure that begins with the output module of the circuit. There is always exactly one output module, but this module may contain multiple output buses. All input connections to the module currently being analysed are traced to their source modules. Their compute time is added to their start cycle to give the data-ready cycle for each module. The start cycle for the module being analysed is the maximum of the data-ready cycles of its input modules. Modules that have no inputs, such as the virtual input modules, constants, or modules with only external inputs, are assigned a start cycle of zero. The procedure is repeated for each of the modules providing data to the module under scrutiny.

The analysis algorithm ensures that data flows correctly when module resource sharing is used, by limiting initiations of the shared module instance. This is detailed in section 4.5.2.

Due to the fact that feedback connections are ignored for the above analysis, and that a module's inputs and outputs may all be feedback connections, there may be modules that do not appear to be part of the circuit's data flow paths. These modules will not be given a start cycle by the described feedforward analysis. Therefore, every module that has not already had its start cycle calculated is considered to be an output module and the feedforward analysis is performed to find its start cycle and the start cycles of modules that it depends upon. After this is done, the start cycles for all modules are known.

4.5.2 Dataflow Ordering of Resource-Shared Modules

A module that is sharing its resources cannot, by definition, have its hardware used by more than one module at a time. The switching is effected by the addition of a multiplexer, which introduces no delay as it is implemented as a purely combinational module. Since the inputs for only one of the shared modules are enabled at any time, it is possible that extra delays are introduced by the elimination of overlapping inputs.

Implementing resource sharing generally lengthens the compute time of a circuit. It can do so to an unnecessary extent if the modules—replaced by a single module—are switched into the circuit in the wrong order (with respect to the natural data flow of the circuit). Therefore the order of the resource-shared modules must be chosen carefully so as to provide the most efficient module usage. The order is obtained by firstly ignoring resource sharing and performing the feedforward dataflow analysis, to obtain the start cycles for each module. The shared module that will be used first is the one with the lowest start cycle. The remaining order is obtained by arranging modules
by increasing start cycle value. This works because no module can depend on another module (while ignoring feedback connections) and yet have a lower start cycle. If two resource-shared modules are given the same start cycle, the dataflow order is chosen arbitrarily.

A minimum delay of one cycle is required between the arrival of data at two modules that are sharing their hardware resources. This is to eliminate the race condition that will occur if a purely combinational module has its outputs fed back to its inputs. Even if this delay is added between users of the shared module, outputs from the module are able to bypass it to feed other modules. This can produce a case where the collision vector is longer than the compute time. For example, a purely combinational shared module is connected directly to the circuit inputs (via a multiplexer) and the circuit outputs. In this case, reinitialization of the circuit is delayed by one cycle to allow the module to hold the output data valid. An example of this case is shown in table 6.3 (on page 130).

4.5.3 Analysis of Feedback Paths

Once the feedforward dataflow analysis has been performed, the start cycles for each module in the circuit are known. Feedback paths are not included in the feedforward dataflow analysis. Instead, fixed length delay elements are added in the feedback paths to ensure that data will be ready on the correct module start cycle on the subsequent circuit iteration. If variable length delays were used, along with their associated control circuitry, the feedback paths could be included in the feedforward dataflow analysis, and resulting non-constant latency sequences could be derived as shown by Stone[90]. Using static delays in the feedback path provides a latency of a constant duration, but usually results in a higher average latency.

It is also possible that converters will be required in the feedback paths. These are added earlier than the delay modules, because they can introduce a delay component of their own, as part of their conversion. If the combined delay they introduce is greater than the required delay for the feedback path, this has the effect of increasing the average latency. In this situation, a new constant-duration latency cycle must be found.

The determination of a constant-duration latency cycle is simpler than a full pipelined analysis because an exhaustive search is possible and it is evident when the lowest value solution is found. The latency cycle is found from the forbidden latency set, and includes the implicit forbidden latency of zero.

To determine whether a certain latency cycle will cause a collision, the FLS is
compared with a copy of the FLS that has all its values increased by the proposed latency. If any values appear in both FLSs then a collision will occur. This comparison is repeated for multiples of the proposed latency up to the length of the circuit’s collision vector; this ensures that the constant delay does not cause any collisions. As soon as a collision is determined, the next proposed latency can be tried. When there are no collisions for a certain proposed latency, and assuming that the search started at one and increased, the minimum constant-duration latency cycle has been found and the search can cease. The worst case will be a latency cycle of one greater than the highest value in the FLS.

The process is analogous to comparing the reservation table with a shifted copy of itself and looking for collisions. As an example, consider a circuit that has an FLS of \{0, 3\}. To see if initiations can begin after one cycle, this is compared with versions shifted by one: \{1, 4\}, \{2, 5\}, and \{3, 6\}. The third trial shows a collision because they both include the value 3. Next, versions shifted by two are tried: \{2, 5\} and \{4, 7\}. This produces no collisions, and therefore the latency cycle of (2) is chosen.

Consider the circuit shown in figure 4.11(a). Ignoring the feedback paths, the compute time is 5 cycles. Assume extra information indicates that the average latency is 4 cycles with a latency cycle of (3, 5). The smallest number of cycles, between when feedback is applied to a module, and data is available to be fed back again, is 5 cycles (over the entire circuit in this case). Therefore the minimum latency is 5 cycles when feedback is considered. Unfortunately, this might not be possible because of pipeline collisions. In this example, the smallest constant-duration latency cycle is (8). Delay elements are added as shown in figure 4.11(b) to ensure every module has its data every 8 cycles.

### 4.5.4 Pipeline Analysis

The aim of pipeline analysis is to determine a control strategy that allows initiations of the circuit to occur before the prior circuit iteration has finished, without producing collisions with hardware usage. This reduces the average latency of the circuit. The control strategy discussed in this section does not allow dynamic changes to the times that data is presented to the circuit, but it does attempt to implement maximal throughput. This means if an initiation time sequence is [0, 2, 4, 6, ...], but the input data to the circuit will not be ready after the first initiation (on cycle 0) until cycle 3, then the input must wait and be applied on cycle 4. Stone[90] presents a very elegant and fast controller for dynamically controlling pipelined circuits. This was based on a prior design by Davidson[18] that uses a shift-register but does not attempt to max-
Figure 4.11: The effect of feedback on dataflow. (a) Circuit with feedback connections. (b) Circuit with delay elements in the feedback paths to ensure data coherency with a constant duration latency of eight cycles.

imise throughput. The pipeline analysis techniques used in this section are based on the algorithms described by Kogge[44], Lewin[46], Battersby[8], Stone[90], and one of the founders of pipeline research, Davidson[18, 68].

The algorithm for deriving a control strategy uses the collision vectors of all the modules comprising the circuit to build a forbidden latency set. The FLS can be represented by a collision vector for the overall circuit, and this can be used to derive the initiation latency sequence and latency cycle. The circuit collision vector is output from the synthesiser and used as a module parameter in hierarchical designs.

A reservation table is created with one row for each latency in the FLS. The MAL is always greater than or equal to the number of marks in any single row of the reservation table, but in such an arbitrary representation, this is correct, but not useful (as there are always two marks). It is more useful to note that the MAL is less than or equal to the number of '1's in the circuit's collision vector[44].

To ensure that resource-shared modules are correctly represented in the FLS and reservation table, modifications to their entries are made. The reservation table rows for each module in a resource-sharing group can be combined, to represent the one module actually implemented. The reservation table rows are merged by performing a logical OR of the rows representing modules that share resources. The additional FLS entries are derived by observing the forbidden latencies described by the new
4.5 Dataflow Analysis

Step 1 Start with the initial collision vector as the initial state.
Step 2 For each unprocessed state and for each \( k \) such that the \( k \)th bit of the corresponding collision vector is zero:

- 2.1 Shift the collision vector \( k \) bits left.
- 2.2 Delete the first \( k \) bits.
- 2.3 Append \( k \) zeros to the right.
- 2.4 Logically OR with a copy of the initial collision vector.
- 2.5 This is a new state. Connect to the current state by an arc of value \( k \).

Step 3 Include an arc with value ‘\( \geq d \)’ from each state back to the initial state where \( d \) is the compute time of the reservation table.

---

Table 4.8: Algorithm for generating the modified state diagram.

merged rows (as shown in figure 4.10). The minimum one-cycle delay (discussed on page 76) is preserved, since every module has at least one entry in the reservation table corresponding to a forbidden latency of 0 cycles.

The first step in calculating the latency cycle is to create a modified state diagram\([18]\) (known also as a reduced state diagram). Table 4.8 shows the algorithm presented by Kogge for creating a modified state diagram from a circuit’s collision vector—known as the initial collision vector. The algorithm states represent the contents of a shift-register controller, under all circumstances where a new circuit operation has just been initiated. The modified state diagram for the initial collision vector \( 01010011 \) is given in figure 4.12.

A greedy strategy is used to calculate the latency cycle, and thus the control strategy. It is known as a greedy strategy because it attempts to initiate a circuit at the earliest possible time, which may not give optimal performance. The average latency of any resulting greedy cycle is greater than or equal to the MAL. The upper latency bound on greedy cycles is equal to the number of ones in the initial collision vector plus one. (Davidson has described another branch-and-bound type search of the modified state diagram that will generate all the MAL cycles.)

Kogge’s presentation of the algorithm for finding all greedy cycles is given in table 4.9. The procedure takes no more than \( N \) iterations of steps 2 to 4, where \( N \) is the number of states in the diagram. The greedy cycle of \( (1, 5, 5) \) can be seen in figure 4.12 by following the lowest value arcs from the initial collision vector. This gives an average latency of approximately 3.67, which can be compared to the visually more apparent (and lower) average latency of 3 obtained by following the arcs of value 3 from the initial collision vector. An initiation time sequence can be readily formed from the best
Figure 4.12: Modified state diagram for the initial collision vector 01010011. The numbers on the arcs represent the number of cycles from one state to the next. The greedy cycle of latencies (1, 5, 5) is shown in bold lines starting from the initial collision vector.
4.6 Module Generators

Step 1: From the modified state diagram pick any state.
Step 2: Follow the sequence of minimum latency arcs until either the state from step 1 is encountered or there are no more arcs.
Step 3: In the former case the traced sequence is a greedy cycle. Record it and remove all states in it from the diagram.
Step 4: Pick another state and go back to step 2.

Table 4.9: Algorithm for finding all greedy cycles.

Greedy cycles. This sequence is used to create a control signal generator to control all of the modules as described in section 4.7 and to calculate when external signals are required as shown in section 4.7.1.

The performance of a pipelined circuit can be increased by the addition of delays as developed by Kogge[44], and Davidson and Patel[68]. They have shown that, by the addition of delay elements, it is always possible to attain maximum performance by removing collisions. The initiation rate is increased at the cost of an increased compute time.

4.6 Module Generators

The synthesiser creates circuits that are built from interconnected modules: either auxiliary or functional. The auxiliary modules are created by the synthesiser during the synthesis process. The functional modules are either predefined with AHDL module interface shells, or they are synthesised by module generators. Each module generator is a separate program (see section 7.6.2) that is initialised and controlled by the synthesiser. A module generator is able to synthesise all the files required for a module implementation. These files include the AHDL module interface shell, the AHDL functional text design file, and associated files such as memory initialisation files. The module generator can synthesise module implementations for various parameters of accuracy and data type. Moreover, a single module generator can synthesise functionally different modules—such as multiplication and division.

As part of the module generation process, the module implementation’s parameters (such as logic cell usage, memory cell usage, compute time, and collision vector) are determined and entered into the synthesiser’s database. The average latency is not a parameter because it is meaningless without the context of the circuit for which the module is intended. It is, however, inherently described by the collision vector. The modules’ values are determined, often empirically, by the module designer as part of the
module design process. This is described in more detail in section 4.6.1. The module
generators eliminate the need to have a large number of separate predefined module
implementation files; they are effectively encapsulated within the module generators.

The library of module generators provides the synthesiser with a wide range of
module implementations, such that the most appropriate implementation for a module
closely matches the desired accuracy and data type. Furthermore, the larger the variety
of module implementations for a desired module, the larger the search space is for the
optimisation algorithm to find the most suitable circuit for given criteria.

There is currently a library of 18 supporting module generators that synthesise 61
operations for various accuracies and data types. Table 4.10 gives a subset of these.
Different versions of a synthesised module may not support all the same accuracies
and data types as another implementation version. There is no limit to the number of
different module implementations that can be described in any single module generator,
and for each module function, as many data types and accuracies can be supported as
required.

A function cannot have different versions implemented in a single module generator.
A module version is associated with a module generator filename during initialisation of
the synthesiser. A particular generated version is chosen by the synthesiser by choosing
a module generator. Two versions of the same module function within the same module
generator would not be distinguishable by the synthesiser.

4.6.1 Estimated Hardware Resource Requirements

The designers of the module generators may use their knowledge of the design to cal-
culate the memory cell usage, compute time, and collision vector for the synthesised
modules. These are often functions of the accuracy being synthesised. Other param-
eters, such as whether or not the module is signed, also affect these features.

Logic cell usage is more difficult to estimate because there are no simple models for
the AHDL compiler and fitter. Therefore, this is obtained empirically by generating
and compiling examples, then formulating a relationship between the input parameters
and logic cell usage. An example is outlined below to show the process.

Five different parallel-multiplier module generators were developed that can syn-
thesis modules from an accuracy \( A \) of 2 bits to 999 bits. The multipliers are such that
they output the correctly-signed lower \( A \) bits of the output where each input is \( A \) bits
accurate. The five module generators differ in the compute time \( T \) of the multipliers;
they range from purely combinational, to four-stage pipelined implementations with a
compute time of four cycles.
Table 4.10: A subset of the modules defined in the library of module generators.

A useful operating range of $2 \leq A \leq 32$ was chosen for the modeling. A sample of implementations in this range were generated and compiled. The numbers of logic cells required ($n$) for the sample implementations are shown in table 4.11.

Next, an appropriate model was chosen. This was done by noting the linearity of the graph of log($n$) versus log($A$) as shown in figure 4.13. The graph indicates that the model should be of the form given in equation 4.4 where $k_T$ and $R_T$ are constants for each different generator (with different compute times, $T$).

$$n_{(T,A)} = k_T A^{R_T} \quad (4.4)$$

The derived model constants are mathematically determined from the slope and offset of the graphed lines, and are presented in table 4.12.

### 4.7 Control Unit

The control unit generates all of the control signals for each module at the correct time so that they will correctly process their data. For pipelined circuits that are operated for multiple iterations, the control signals are generated to be correct for each iteration. The controller circuit hardware description is based on a counter and a lookup table. The actual hardware implementation on an FPGA is up to the compiler's discretion.
<table>
<thead>
<tr>
<th>Accuracy (bits)</th>
<th>Compute Time (cycles)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>12</td>
<td>19</td>
<td>21</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>72</td>
<td>64</td>
<td>78</td>
<td>87</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>155</td>
<td>135</td>
<td>137</td>
<td>163</td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>186</td>
<td>164</td>
<td>165</td>
<td>195</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>816</td>
<td>728</td>
<td>719</td>
<td>740</td>
<td>833</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1482</td>
<td>1337</td>
<td>1329</td>
<td>1333</td>
<td>1488</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.11: The logic cell requirements for generated multiplier modules.

![Figure 4.13: Logic cell usage versus accuracy of generated multiplier modules—shown on a logarithmic scale. Each curve represents a different module generator with its own compute time, T.](image)

<table>
<thead>
<tr>
<th>Compute Time (T)</th>
<th>$k_T$</th>
<th>$R_T$</th>
<th>Logic-Cell Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.435</td>
<td>2.383</td>
<td>$n_{(0,A)} = 0.435 \cdot A^{2.383}$</td>
</tr>
<tr>
<td>1</td>
<td>0.446</td>
<td>2.346</td>
<td>$n_{(1,A)} = 0.446 \cdot A^{2.346}$</td>
</tr>
<tr>
<td>2</td>
<td>0.967</td>
<td>2.094</td>
<td>$n_{(2,A)} = 0.967 \cdot A^{2.094}$</td>
</tr>
<tr>
<td>3</td>
<td>1.518</td>
<td>1.949</td>
<td>$n_{(3,A)} = 1.518 \cdot A^{1.949}$</td>
</tr>
<tr>
<td>4</td>
<td>1.986</td>
<td>1.885</td>
<td>$n_{(4,A)} = 1.986 \cdot A^{1.885}$</td>
</tr>
</tbody>
</table>

Table 4.12: Derived formulae for generated multiplier modules. The ‘Compute Time’ is given in cycles.
The controller is synthesised in AHDL and implemented in a separate TDF to the main circuit. The control unit is ignored for module size calculations, because when the circuit is used as a module in a higher-level design, the controller is not retained as part of the module—its control signals are produced by the control unit of the circuit described at the highest level. This means that for any given hierarchical circuit design, only the circuit described at the highest level has a control unit, which controls all the modules (including modules within modules).

After circuit repair and analysis the synthesiser has all the control signal requirements for each module in its database. This information is obtained from the TDF input files, the module generators, and from the synthesiser itself (for the fundamental modules). After the start cycles for each module are known, these control signals can be offset for each module and a control signal table built up. Once the latency sequence has been calculated, and the initiation time sequence derived, the control signals for further iterations are added to the control signal table. A done signal is also added to the control signal table that is high when the output of the circuit is valid. On the first iteration of the circuit, the done signal is high after compute time cycles. The subsequent done signal highs are spaced by the number of cycles obtained from the initiation latency sequence.

When a module shares its resources with other modules, it must be supplied with the control signals of all the modules it represents. This is done by performing a logical OR between each corresponding control line for all the modules in the group of resource shared modules. The resulting lines become the control lines for the physical module instance that is sharing its resources. A module’s control signals for subsequent iterations never interfere with each other. This is because information about its unavailability due to control signal processing is inherently part of its collision vector (which provides the dataflow analysis algorithm with enough information to avoid any destructive signal interaction).

If a multiple-iteration circuit is required, the controller can repeat its control signals over a period dictated by the latency cycle. The first repeat cycle can sometimes be on the first clock cycle of circuit operation. It can always occur on, or before, the latter of:

1. The last control signal for the module on the first circuit iteration.

2. The last control signal for the module on the circuit iteration prior to the start of the latency repeat cycle.

An example of the control line derivation of a three-module circuit is given in figure 4.14. In this example, the control signal repeat cycle could have begun on the first clock cycle.
Figure 4.14: The determination of control signals for a three-module circuit with latency sequence (2,5,2,5,2,...) and start cycles of 0, 1 and 0 for modules A, B and C respectively. The digits in the table indicate control signals for the corresponding iteration. (a) The control signals for each module offset by the module start cycles. (b) Control line combination for resource sharing of modules A and B. (c) Resulting control signal table for multiple iterations. (d) Possible repeat cycle of the control signals for ongoing circuit operation where cycles 2–8 are repeated (marked with an x).
4.7.1 External Control Signals

A module may require external control inputs. These could be switching signals, address buses, or data buses\(^3\). They can originate from a separate circuit, such as a microcontroller. The external input lines do not necessarily require a data type and are not limited to the same bus width as the standard module inputs. If they are associated with a type, this is given in the textual description of the input. The data must be provided by any external hardware in the correct format. No automatic type conversions are performed because the external inputs may just be arbitrary control lines.

When a module is loaded into the synthesiser from a TDF file or synthesised by a module generator, the required external controls (defined in the module interface shell, or by the module generator) are automatically entered into the database. These database entries describe which external control lines require a signal on a certain clock cycle. Control lines that are conceptually related, such as the bits in an external parallel binary constant, are grouped together for simpler interpretation. The same group can require different input signals on different clock cycles.

To determine the correct clock signal for the external controls in the overall circuit, the clock cycle requirements for each module are offset by the start cycle of the module. When used in a design that has multiple circuit initiations, each external control line group will be used at least once per iteration, and the reinitialisation sequence is used to determine the module start cycle for each initiation.

A hierarchical description of each external control line group is built up (using `::` characters to separate circuits from modules), so that even in a high-level design the module related to an external input can be determined. This method of referencing uses the unique module reference that is assigned to each module in a circuit, thus differentiating between two or more similar modules in a circuit. When resource sharing is used for two or more modules, any external control lines that these modules require will appear as the same group of inputs. Each module’s input requirements, in this situation, are detailed in the resulting external control line descriptions.

An example of the hierarchical buildup of external control line descriptions is given in figure 4.15, showing the relevant section of the synthesiser report file for each circuit. The cycles shown are dependent on the topology of the given circuit, and \( n \) is the circuit iteration, which starts at 0. Figure 4.15(a) describes the external input requirements for an adder module that has one normal input (the first addend), whose data type and accuracy are unimportant for this discussion. The external input expected on cycle 2

---

\(^3\)External numerical constants, for example.
is from an external source—possibly a microcontroller. This external input is expected as a parallel 6 bit input that defines the memory address of the second addend. The module implementation will provide the functionality to retrieve the data from the memory source. The 6 bit address bus will be physically connected to the resulting circuit by the lines described in its module interface shell as ExternalControl[5..0].

Figure 4.15(b) shows the external input requirements for a circuit that contains two of the adders from figure 4.15(a). In this circuit, the two adders' descriptions are 'Main adder' and 'Secondary adder'. The physical connections for the external inputs to the 'Main adder' are ExternalControl[5..0], which are required on cycles 4 and 4n + 6, and the ones to the 'Secondary adder' are ExternalControl[11..6], which are required on cycles 3 and 4n + 5.

Figure 4.15(c) shows the effect of resource sharing the modules in figure 4.15(b). The descriptions of the external inputs remain identical, as do the cycles on which the external inputs are required. The difference is that the modules are sharing the same external control lines, namely ExternalControl[5..0].

Figure 4.15(d) presents the external input requirements for a circuit that has the circuits of figure 4.15(a) and figure 4.15(b) as modules. There are a total of 18 external control lines arranged in 3 groups of 6.

4.8 Graphical Design Tool

A graphical design tool (GDT) was developed to allow the entry of complex circuits. Modules can be chosen from TDF files or module generators. Their data types and accuracies can be set, and parameters for each module, such as the constant for a 'constant multiplier', can be given. Modules that are to share resources can also be indicated. Figure 4.16 shows the main screen and parameter entry screen of the graphical design tool.

The output of the GDT is a netlist file that becomes the input to the synthesiser. Overall circuit parameters can be entered to indicate the input and output data type and accuracy. The function name must also be given, which will become the reference for the circuit to be used as a module in a higher-level design.

The GDT allows hierarchical design entry. The drawback of using hierarchical designs is that the synthesiser, and the optimisation within it, works at the module level. Consider a circuit with 2 neuron modules, each containing 16 multipliers. The best resource sharing that can be performed at the module level is obtained by implementing one neuron and multiplexing it in time. If the circuit was entered with all 32 multipliers, there would be myriad combinations of ways to share the resources of the multipliers.
Figure 4.15: Hierarchical external control line descriptions. (a) The external controls for a simple adder that expects an external 6 bit input as the memory address of one of the addends. (b) The external controls for a circuit that includes two external adders. (c) The effect of sharing the resources of the two external adders of (b). (d) The required external control lines of a top-level circuit that includes modules of (a) and (b). (n is the current circuit iteration, where n = 0, 1, 2, ...)

Figure 4.16: (a) Main design screen of the graphical design tool showing an example circuit. (b) Entry window for the principal module parameters.
Because this allows such powerful optimisation possibilities, an option is included in the GDT that allows the export of a netlist that includes all of the lowest level modules of the circuit's modules and its modules' modules and so on. These are recursively extracted and given unique module numbers and references.

4.9 Case Study

A pipelined circuit was developed to provide an example of the synthesiser's operation. The circuit will perform the averaging of a group of pixel values within a $3 \times 3$ mask. The pixel values are fed into the circuit in a pipelined fashion, three adjacent pixels at a time, $(A_n, B_n, C_n)$, on circuit iteration $n$. All pixel values are available as 8 bit parallel numbers. The operation of the averaging filter is described by equation 4.5.

$$Q_n = \frac{A_n + B_n + C_n + A_{n-1} + B_{n-1} + C_{n-1} + A_{n-2} + B_{n-2} + C_{n-2}}{9} \quad (4.5)$$

An approximation to equation 4.5 can be obtained by ignoring the value of the centre pixel, represented by the $B_{n-1}$ term. This becomes equation 4.6, which is much simpler in terms of its hardware implementation.

$$Q_n = \frac{A_n + B_n + C_n + A_{n-1} + C_{n-1} + A_{n-2} + B_{n-2} + C_{n-2}}{8} \quad (4.6)$$

The full mask, the approximated mask, and the movement of the mask over a pixel-based image are shown in figure 4.17. The circuit topology fed into the synthesiser is shown in figure 4.18. Results obtained from the synthesiser for various combinations of data types and resource sharing are given in table 4.13. Note that with resource sharing, if modules with different accuracies are shared, the highest accuracy is chosen for the implemented module, which in addition to the added multiplexer and delays can make resource sharing an unattractive option. Figure 4.19 presents a timing diagram for one of the implemented circuit variations. The figure is the result of a full timing simulation of the design, obtained after circuit compilation. It shows the pipelined operation of the circuit, and its propagation delays.

The circuit was modified to average all nine pixels (equation 4.5). This contrasts the difference between circuit solutions resulting from a full integer divide and a binary shift operation. Table 4.14 shows the different resulting parameters for circuits composed of only parallel modules—for both the 8 pixel and 9 pixel averaging-filter circuits.

4.10 Conclusions

This chapter presented an HDL synthesiser that processed high-level circuit descriptions. It was developed for analysing and synthesising hierarchically designed circuits.
Figure 4.17: Pixel averaging filter. (a) The $3 \times 3$ spatial averaging mask. (b) Hardware-implementation-friendly approximation to the $3 \times 3$ spatial averaging mask. (c) The mask applied to an image with the arrows showing the direction of mask movement.

Figure 4.18: Pixel averaging filter circuit using a $3 \times 3$ spatial mask approximation given by equation 4.6. The pixels A, B and C are fed into the circuit in a pipelined fashion.
<table>
<thead>
<tr>
<th>Parallel Module #s</th>
<th>Shared Modules</th>
<th>Actual Logic Cells</th>
<th>Estimated Logic Cells</th>
<th>Compute Time (Cycles)</th>
<th>Average Latency (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,3,4,5,6</td>
<td>-</td>
<td>97</td>
<td>109</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>100</td>
<td>88</td>
<td>12</td>
<td>12.00</td>
</tr>
<tr>
<td>1,2,6</td>
<td>-</td>
<td>120</td>
<td>119</td>
<td>12</td>
<td>12.00</td>
</tr>
<tr>
<td>2,4,5,6</td>
<td>-</td>
<td>527</td>
<td>596</td>
<td>20</td>
<td>11.00</td>
</tr>
<tr>
<td>1,2,3,4,6</td>
<td>-</td>
<td>331</td>
<td>355</td>
<td>12</td>
<td>12.00</td>
</tr>
<tr>
<td>1,2,3,4,5,6</td>
<td>(1,2,3,5)</td>
<td>284</td>
<td>352</td>
<td>3</td>
<td>4.00</td>
</tr>
<tr>
<td>6</td>
<td>(1,2,3,5)</td>
<td>308</td>
<td>290</td>
<td>48</td>
<td>48.00</td>
</tr>
<tr>
<td>4,6</td>
<td>(1,2,3,5)</td>
<td>566</td>
<td>534</td>
<td>48</td>
<td>48.00</td>
</tr>
<tr>
<td>1,2,3,4,5,6</td>
<td>(3,5)</td>
<td>153</td>
<td>196</td>
<td>1</td>
<td>2.00</td>
</tr>
<tr>
<td>6</td>
<td>(1,2,5)</td>
<td>240</td>
<td>219</td>
<td>36</td>
<td>36.00</td>
</tr>
<tr>
<td>6</td>
<td>(1,2)(3,5)</td>
<td>157</td>
<td>142</td>
<td>22</td>
<td>20.00</td>
</tr>
</tbody>
</table>

Table 4.13: Results for various pixel-averaging circuit implementations using logic-cell based modules only. The sizes shown are actual implementation hardware usage after HDL compilation (including controller circuit), and estimates from the synthesiser. The ‘Parallel Module #s’ column lists the module numbers of the modules implemented as parallel modules; the remaining modules are implemented as serial.

Figure 4.19: Pixel averaging filter timing diagram for implementation with compute time of 3 cycles and constant latency of 4.00 cycles. The inputs are presented starting with the Start signal and then every fourth clock cycle. The outputs hold their correct values when the Done signal is high. Note the settling time caused by propagation delays before the output lines stabilise.
Table 4.14: Comparison between using divide-by-8 and divide-by-9 division modules in the pixel-averaging filter circuit with all parallel modules and no resource sharing. The sizes are those calculated by the synthesiser.

The original contribution presented in this chapter is the development of an entire synthesis algorithm that is a unique combination of techniques and methods to create diverse and novel circuit solutions. This includes a number of supporting subsystems and concepts, such as the module interface shell and module generators.

The synthesiser is able to utilise modules designed with a wide range of tools, and combine them to produce the required circuit functionality. Variable width data paths are supported, as are different data types, and resource sharing. It was shown that varying the presented module parameters can produce designs with different size and time requirements—giving different possible solutions in a circuit’s design space. The synthesiser provides a foundation for circuit optimisation, and is used in conjunction with the optimisation algorithm described in chapter 5 to provide the results and examples given in the remainder of the thesis.

Proposed extensions to the synthesiser are:

- Tighter integration with hardware-software co-design through the inclusion of software modules and associated algorithms for their processing.
- Estimation of the propagation delay of each synthesised module could be made. Analysis including this parameter would allow the circuit’s clock frequency to become part of the optimisation goal. The circuit repair would have to sum sequential propagation delays and ensure module outputs stabilise before the latching edge of the clock signal, otherwise delay elements would be needed. The average latency of different circuits could be compared by normalising them into seconds by dividing by the clock frequency (in Hertz) of each circuit.
- A larger library of synchronous modules that utilise a wider range of data types. This would provide more possible solutions, some of which may more closely
match an ideal solution.
Chapter 5

Circuit Optimisation

5.1 Introduction

The aim of this chapter is to present the optimisation algorithm used in conjunction with the synchronous-module synthesiser. The task of the optimisation algorithm is to search the design-space of possible circuit solutions, with the goal being to find an optimal design for given certain search criteria. A genetic algorithm (GA) was chosen as the search algorithm. Problem specific encoding methods and the associated genetic manipulation mechanisms of mutation, crossover, selection and reproduction are developed and presented. The encoding method is important because it determines what features of the circuit are able to be manipulated. It also influences the complexity of the developed crossover and mutation mechanisms. These mechanisms affect the direction and convergence of the search.

Essentially, the intent of the optimisation is to provide a circuit solution from a high-level description. Given design criteria of allowable hardware resource usage (logic cells and memory cells) and time based criteria (circuit compute time and average latency), the GA searches the design space for an appropriate solution. This is achieved by modifying the data types and implementation versions of various modules in the circuit and allowing the synthesiser to subsequently ‘repair’ the circuit and calculate its conformance to the design criteria. Once a solution is found that is acceptable, the synthesiser completes the circuit synthesis process and produces the complete HDL description of the circuit for subsequent compilation and on-device implementation.

The results obtained from using the optimisation techniques and algorithm derived in this chapter are given in chapter 6. The research presented in this chapter has, in part, been published[54, 56].
5.2 Genetic Algorithm Overview

Genetic algorithms are robust, general-purpose search techniques that are very popular in the disciplines of Computer Science and Engineering. They are often easily outperformed by conventional search techniques, but are useful where there are no known techniques that are appropriate for a particular problem, or the specialised techniques are too computationally intensive or are infeasible to implement. GAs are applicable to problems where a good solution is adequate; where it is not necessary to find the optimal (best) or only solution (such as in cryptographic searches where there is only one valid solution). There are a wide variety of problems for which a ‘near-optimal’ solution is acceptable[38]. GAs mimic biological evolution by breeding a population of possible solutions together and using the fitness of each individual to evolve an optimised solution using Darwinian “survival of the fittest.”1 GAs differ from traditional search and optimisation methods in four fundamental ways[28]:

- GAs operate on a coding of the parameters, not the parameters themselves.
- GAs search from a population of solution points, not a single point.
- GAs utilise a blind search that uses payoff information in the form of an objective fitness function instead of derivatives or other auxiliary knowledge.
- GAs use probabilistic transition rules, not deterministic ones.

5.3 Redundancy Removal

Within a circuit there may be groups of modules that perform identical functions on identical data, and therefore introduce redundancy. These groups are often not self-evident, but can result in larger than necessary circuits. It is necessary that redundancy is accounted for to allow an accurate estimation of hardware resource usage to be made. The process of looking for identical sub-circuit structures can be done in a preprocessor before the rest of the synthesis and optimisation begins.

The process of determining redundant data paths is performed by symbolically analysing a circuit’s nodes, which are formed from the module interconnections or module outputs. An extra parameter is required for each module stating whether its inputs are order-sensitive or not. (It would be possible to extend this to determine whether certain subsets of inputs are order-sensitive for a module.) As an example, input order is not important for an addition module, whereas it is for a subtraction

---

1Cowley[17] presents a very interesting article on the role that physical beauty plays in the “fitness function” of human biological evolution.
module. The inputs are assigned to the symbolic module function in alphabetical order when they are not order sensitive. This ensures that two identical modules with identical inputs have identical symbolic node values.

When identical symbolic node values are determined, one of the circuit's data paths producing the identical nodes can be trimmed back until values are required by other modules in the circuit. That is, if multiple modules produce the same node value, all but one of them is not needed. Modules that feed these redundant modules are only needed if they also feed other non-redundant modules—and so on.

In general, a higher level analysis (with knowledge of algebraic rules) cannot be performed without information about module accuracies and how numerical overflows are handled. That is, even if the nodal analysis algorithm knew about the commutative and distributive properties of algebraic modules, the result of the function \( Q = 64(A - B) \) could be different in fixed-point arithmetic depending on whether \( A - B \) was calculated first, or \( 64A \) and \( 64B \) were calculated and the subtraction subsequently performed.

Figure 5.1(a) shows a circuit in which \( N1 \) and \( N2 \) can be visually determined to be identical. The analysis process, however, indicates two different values for these nodes:

\[
\begin{align*}
N1(k) &= \text{add}(A(k), N1(k-1)) \\
N2(k) &= \text{add}(A(k), N2(k-1))
\end{align*}
\]

In the case where a node depends on the value of itself from a previous iteration, it should be replaced with a symbol—in this case an 'X'. In this example, both nodes \( N1 \) and \( N2 \) can be rewritten as:

\[
\begin{align*}
X(k) &= \text{add}(A(k), X(k-1))
\end{align*}
\]

indicating that they are identical.

Finally, when there are multiple nodes that depend on the values of each other from previous circuit iterations, as shown in figure 5.1(b), each node must first be expanded recursively to be written in terms of itself. The first step gives the nodal values:

\[
\begin{align*}
N1(k) &= \text{add}(A(k), N2(k-1)) \\
N2(k) &= \text{add}(A(k), N1(k-1))
\end{align*}
\]

Expanding these gives:

\[
\begin{align*}
N1(k) &= \text{add}(A(k), \text{add}(A(k-1), N1(k-2))) \\
N2(k) &= \text{add}(A(k), \text{add}(A(k-1), N2(k-2)))
\end{align*}
\]

Now, as for the circuit in figure 5.1(a), when the value of a node depends on a previous value of itself it is replaced with an 'X'. This shows that \( N1 \) and \( N2 \) are indeed identical.
Step 1 For each node:

1.1 Expand its functional description to a function of the circuit inputs and prior values of itself
1.2 If node is self dependent:
   1.2.1 Replace self-dependent terms with an ‘X’
1.3 If current node functional description is identical to one already processed then mark as ‘redundant’

Step 2 Eliminate modules that lead only to redundant nodes
Step 3 Reconnect the orphaned inputs, that were connected to the eliminated module outputs, to the remaining non-redundant module

Table 5.1: Redundancy removal algorithm used in circuit optimisation.

\[ X(k) = \text{adder}(A(k), \text{adder}(A(k-1), X(k-2))) \];

The removal of redundancies before the optimisation process is an important step in finding a good design solution. Ideally it is done by the circuit designer, but in the cases where it is not obvious, symbolic analysis ensures that the optimisation algorithm has a better chance of finding a suitable solution. Table 5.1 gives the algorithm for removing redundant modules for the general case.
5.4 High-Level Genetic Algorithm Optimisation

An initially random population of possible solutions is modified through each individual undergoing the genetic operations of mutation, crossover, selection and reproduction. An individual (or chromosome or string), which represents a possible solution, is an encoding of the circuit's design, with various features (or detectors) that can take on different values and represent aspects of the circuit. The genetic operations that are performed on the individually modifiable attributes of the circuit result in different compute times and different hardware resource usages, while guaranteeing the same functionality. Parameters are given to the GA that define how important size and time are in the optimisation process, and whether there are any hard constraints that must be met.

5.4.1 Problem Encoding

The genetic operations of a GA are performed on an encoding of the problem being optimised. Therefore, each individual solution in the GA population must be represented by a collection of features, often in the form of a list of features to facilitate simple crossover mechanisms. Each feature is given a fixed position within the list, which relates to its physical interpretation. The encoding regime is chosen to reduce the probability that crossover and mutation operations will produce chromosomes that cannot be mapped to real solutions. The concept of a schema (plural, schemata), presented by Holland[28], provides a measure of the similarity between two strings by observing one or more features at the same positions in both strings that have the same value. It is sometimes referred to as a similarity template. The mechanisms of mutation and crossover can be described and analysed using schemata.

There are two basic principles for choosing a GA coding scheme[28]. The first is the principle of meaningful building blocks, which states:

The user should select a coding so that short, low-order schemata are relevant to the underlying problem and relatively unrelated to schemata over other fixed positions.

The second basic principle is the principle of minimal alphabets, which states:

The user should select the smallest alphabet that permits a natural expression of the problem.

An encoding was chosen that had two parts to the chromosome: the primary portion (head) and secondary portion (tail). The head contains information for each module
as to which group of modules it belongs; each group having a set of parameters for the modules comprising it. The tail contains the parameters indicated by membership of each group. The variables for each group of modules describe their data type, their implementation version, and whether they are a shared resource (implemented in a single physical instantiation).

One or more modules may belong to a single group—that is, have the aspects defined by that group. All the members of a group must perform identical functions (although possibly on data of different types and accuracies, and they can utilise different algorithms). Modules cannot be considered identical if they have additional design parameters that differ. For example, a constant multiplier is given its constant multiplicant at design time. Therefore, constant multiplier modules may be members of the same group only if their constants are the same; otherwise they cannot have their resources shared and therefore cannot be in the same grouping.

The information contained in the chromosome can be classified as follows:

**Referenced Value:** This is the index on which lookup in the head or tail sections of the chromosome is performed. In the head it is the module number, and in the tail it is the group name.

**A Variable:** A feature's value that can be modified by the genetic algorithm via the mechanisms of mutation and crossover.

The circuit of figure 5.2 is used to illustrate the encoding procedure. Initially, random groups in the chromosome tail are referenced by each module in the chromosome head. The variables of each group are chosen randomly from the available options for its associated module. That is, only viable data type and version values are chosen for each module. The encoded chromosome head is detailed in table 5.2, and the chromosome tail in table 5.3. Figure 5.3 gives a graphical representation of the circuit encoding. Note that in this example groups A4 and B2 are not referenced by any modules. Groups B1 and A2 each contain two modules, but the modules are not shared resources. Furthermore, groups A1 and A3 utilise resource sharing, but since they each only contain one module, the sharing is meaningless.

The chromosome head is ordered by module operation (making functionally equivalent modules contiguous) to minimise the destruction caused by crossover. This is because crossover must occur within (or between) one group—for the chosen, single-point, crossover. That is, modules performing the same operation are contiguous in the chromosome head. In terms of schemata, this ordering regime maintains similarity throughout the crossover process. In contrast, preserving no similarity would result in a random walk through the solution space.
Figure 5.2: The circuit used to demonstrate circuit encoding into its chromosome representation. The implementation version and data type were set arbitrarily. None of the modules utilise shared resources.

<table>
<thead>
<tr>
<th>Module No.</th>
<th>Module Operation</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reference)</td>
<td>(implied)</td>
<td>(variable)</td>
</tr>
<tr>
<td>1</td>
<td>Add</td>
<td>A1</td>
</tr>
<tr>
<td>3</td>
<td>Add</td>
<td>A2</td>
</tr>
<tr>
<td>7</td>
<td>Add</td>
<td>A3</td>
</tr>
<tr>
<td>6</td>
<td>Add</td>
<td>A2</td>
</tr>
<tr>
<td>2</td>
<td>Subtract</td>
<td>B1</td>
</tr>
<tr>
<td>4</td>
<td>Subtract</td>
<td>B1</td>
</tr>
<tr>
<td>5</td>
<td>Subtract</td>
<td>B3</td>
</tr>
</tbody>
</table>

Table 5.2: The primary portion of the chromosome representing an individual. The Group is the variable whose value is changed through mutation and crossover. As shown, the Group variable need not be unique. Example data is given for the circuit of figure 5.2.
<table>
<thead>
<tr>
<th>Group (reference)</th>
<th>Module Data Type (variable)</th>
<th>Module Version (variable)</th>
<th>Module Sharing (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Serial</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>A2</td>
<td>Parallel</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>A3</td>
<td>Serial</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>A4</td>
<td>Parallel</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>B1</td>
<td>Parallel</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>B2</td>
<td>Serial</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>B3</td>
<td>Parallel</td>
<td>1</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5.3: The secondary portion of the chromosome that represents an individual by defining the properties of the groups used in the primary individuals main chromosome. The Module Data Type and Module Version are the variables whose values are changed through mutation and crossover. Example data given corresponds to the circuit of figure 5.2. Two of the four addition modules and one of the three subtraction modules are implemented using serial data types.

Figure 5.3: Genetic algorithm encoded circuit representation of the circuit in figure 5.2 (See also tables 5.2 and 5.3).
5.4.2 Genetic Algorithm Parameters

There are a number of parameters associated with the genetic algorithm, which affect its convergence to a solution. The parameters are provided in a configuration file specifically for the GA. The syntax for the configuration file is given in appendix B. The following parameters are employed:

**Number of generations:** This integer value denotes the number of generations over which the population of solutions is to evolve. The greater the number of generations, the more likely a better solution will be found. This statement assumes fitness scaling is being used (see section 5.5.1). The number of generations is usually constrained by the length of time it takes to calculate the fitness of each individual.

**Population size:** This static integer value indicates the number of individuals (distinct circuit solutions) that make up the population. Population sizes that are too small do not exhibit enough genetic diversity to continue converging towards a solution, but they have low inertia—meaning that they can change more rapidly. Large populations have better genetic diversity, but are constrained by calculation time. The greater inertia of large populations also means that they have poor performance in early generations.

**Mutation probability:** This floating-point value is the probability that a mutation of any type occurs to a particular feature of a particular individual in each generation. Different types of mutation occur with the same probability. If this value is too low, new values for detectors may not be introduced and the population may stagnate (become more homogeneous). If this value is too high, good solutions (and their component detector values) may be prematurely destroyed.

**Crossover probability:** This floating-point value is the probability that crossover will occur between two parent individuals chosen at random from a newly selected mating pool. It is the probability of crossover occurring per mating event. The position (of which there may be more than one) where crossover occurs within an individual is chosen with uniform probability from along its length.

**Output frequency:** This integer value controls the number of generations that occur between reporting of the fitnesses of the individuals to a file (for subsequent analysis).

**Exhaustive search:** This is a boolean variable that indicates whether an exhaustive search of the design space is performed. (Exhaustive with respect to the optimisation parameters dealt with by the synthesiser.) It is only feasible for very small
circuits because the number of designs quickly becomes large as the number of modules in the circuit increases.

**Data types:** This parameter consists of a list of data types that are to be used for the modules. It allows certain data-types to be included or excluded to suit the nature of the circuit being optimised. For example, if accuracy of circuit operation is important, and it is desired that the results do not have probabilistic effects, then stochastic modules can be excluded from the search space. (Recall from sections 4.3.2 and 3.3 that stochastic modules operate on long probabilistic serial bit streams instead of standard binary encoded data.)

### 5.4.3 Other Optimisation Techniques

Any of the optimisation techniques that are based on the payoff information of the individual circuit solution are applicable to the search of the circuit design space. There are other appropriate methods apart from the genetic algorithm that are also randomised, though not directionless. One such method is *simulated annealing*, which uses random processes to search for minimal energy states[62].

Hill-climbing and derivative searches are not applicable due to the topology of the search space. The non-linearity of an individual’s fitness, with respect to its chosen module implementations, means there are many local maxima that can mislead slope-based search methods.

Analytic searches based on the qualitative suitability of parts of the circuit are not useful because of the severely non-linear solution space. Even in the simple case where the goal is to minimise the number of logic cells used, choosing all the modules with the smallest logic cell count would not necessarily produce the smallest circuit. The preservation of overall functionality may require the addition of data type converters and delay elements.

### 5.5 Fitness Function Derivation

The aim of circuit optimisation is to reduce both the compute-time for a given number of circuit iterations, and the hardware resource usage of a circuit. The fitness of an individual circuit is a function of the compute-time, average latency, number of logic cells, and number of memory cells. Overall fitness, $f_i$ of a circuit, as an individual in the population of solutions, was chosen to be:

$$f = -p_{lt}p_{mt}(n_lk_l + n_ck_c + n_mk_m)$$

(5.1)
where a numerically greater (less negative) value is better. The variable \( n_t \) is the calculated total compute time (in cycles) for the given number of iterations of the circuit. (Therefore, \( n_t \) implicitly includes information about the circuit’s latency.) The estimated number of logic cells is \( n_c \), and memory cells is \( n_m \). Each variable \( n_t, n_c, \) and \( n_m \) have an associated guiding weight \(( k_t, k_c, \text{ and } k_m \) respectively) used to indicate the relative importance of optimising for time, space or memory cell usage. Each multiplicative penalty \( p_t \) (time), \( p_c \) (logic cells), and \( p_m \) (memory cells) possesses the value of the raw penalty when \( n_t, n_c, \) and \( n_m \) respectively, are greater than predefined limits. Otherwise the penalty variables each possess a value of unity. The hard limits \( h_t, h_c, \) and \( h_m \) are arbitrary values used by the genetic algorithm, as are the raw penalty values \( P_t, P_c, \) and \( P_m, \) to guide the evolution of the solution away from undesirable circuits. Formally, the relationship between the raw penalty values and the penalties is:

\[

p_x = \begin{cases} 
  P_x & \text{if } n_x \leq h_x \\
  1 & \text{otherwise}
\end{cases} 
\]  

(5.2)

where \( x \) stands for \( t, c, \) and \( m \). The calculated and estimated circuit parameters are in their natural ranges:

\[ \{n_t, n_c, n_m\} \geq 0 \]  

(5.3)

The constraints on the penalties and hard limits are:

\[ \{p_t, p_c, p_m\} \geq 1 \quad \text{and} \quad \{h_t, h_c, h_m\} \geq 1 \]  

(5.4)

The constraints on the guiding optimisation weights are:

\[ 0 \leq \{k_t, k_c, k_m\} \leq 1 \]  

(5.5)

Therefore the fitness function—to be maximised—is always less than zero. Figure 5.4 shows how the penalty parameters affect the fitness function when it is a function of two variables; circuit size and compute time. The fitness function as given in equation 5.1 is a function of three variables (time, logic cells, and memory cells) and represents a four-dimensional surface.

### 5.5.1 Fitness Scaling

Each individual’s fitness is scaled before the selection process to control the rate of convergence to a solution. Bagley\(^2\) introduced a fitness scaling mechanism to address

---

\(^2\)Bagley is also attributed as the first to use the phrase “genetic algorithm”, in his pioneering dissertation in 1967.
Figure 5.4: The effect of hard limits and penalties on the fitness function. The shown fitness function is inverted and is only a function of two variables. Without the penalty multiplicants, the fitness function would be an extension of the plane as it is near the origin.

the need to maintain appropriate selection rates at the beginning and end of a GA run[28]. Without suitable scaling in place, “super” individuals can dominate in the early stages of the algorithm. Scaling is also useful near the end of an evolutionary algorithm to prevent the population from becoming too similar and ceasing to converge further. This happens to a converging population because there exist many highly fit, similar individuals, and fitness scaling is required to provide competition and maintain adequate genetic diversity.

There are three common forms of fitness scaling for obtaining a scaled fitness \( f' \) from the raw fitness \( f \)[28]:

1. Linear scaling
2. Sigma (\( \sigma \)) truncation
3. Power law scaling

Linear scaling is of the form:

\[ f' = af + b \]  \hspace{1cm} (5.6)

where \( a \) and \( b \) are usually chosen to cause the maximum scaled fitness to be twice the average scaled fitness of the population while keeping the average value static. This is
done to ensure that average individuals have one copy propagated into the subsequent generation, with the best individuals likely to have two copies survive. Linear scaling can produce negative scaled fitnesses that cause the reproduction operations (based on cumulative sums) to fail. This can be remedied by the use of sigma ($\sigma$) truncation. In this pre-scaling process, the standard deviation ($\sigma$) and the average ($f_{avg}$) of the pre-scaled fitnesses are utilised:

$$f' = f - (f_{avg} - \sigma)$$  \hspace{1cm} (5.7)

The value of $c$ is set at the outset of the GA run (1.0 $\leq c \leq 3.0$) and is chosen to provide a suitable multiple of $\sigma$. Negative scaled fitnesses ($f' < 0$) are arbitrarily set to zero.

Power law scaling is where the raw fitnesses are raised to the power of a value, $k$:

$$f' = f^k$$  \hspace{1cm} (5.8)

The value of $k$ is usually close to, but greater than, unity—such as 1.005, but is problem dependent and its value needs adjusting during the GA run.

A variation on linear scaling was chosen for the circuit optimisation, with appropriate choices of the values of $a$ and $b$ to ensure that negative fitnesses do not occur while trying to obtain a maximum scaled fitness of $C_{mult}$ times the value of the average scaled fitness. $C_{mult}$ represents the number of expected copies into a subsequent generation of an individual with the maximum fitness in the current generation. For small populations of between 50 and 100 individuals, values of $C_{mult}$ between 1.2 and 2 have been successful. Before the scaling process, a value $g$ is added to each fitness value to make them all positive and ensure that the minimum fitness is greater than zero to allow scaling to occur. The constraints on the scaled fitness values are:

1. $f'_{avg} = f_{avg} + g$
2. $f'_{max} = C_{mult} \cdot (f_{avg} + g)$
3. $f' > 0$

where the value $g$ is set for each generation and is equal to $-f_{min}$ for a particular generation where $f_{min} < 0$.

### 5.5.2 Derivation of a Hardware Compression Factor ($\gamma$)

The aim of deriving a hardware compression factor is to develop a model relating the logic cell usage of modules to the overall logic cell usage of a circuit containing multiple modules. The first approximation can be made by summing the number of logic cells
Figure 5.5: Estimated versus actual logic cell usage showing the linear relationship. The data was obtained by obtaining actual and estimated logic cell usages for random circuits as described on page 110.

required by each module to give the total for the circuit. The relationship between the summation and actual results is fairly linear as shown in figure 5.5. For this reason, it is appropriate to choose a constant multiplier, \( \gamma \) (the compression factor). This multiplier is used to adjust the estimated (summed) logic cell usage, so that it more closely approximates that of the actual implementation. This can be represented by:

\[
N = \gamma \sum_{j=1}^{k} n_j
\]

(5.9)

where \( N \) is the estimated number of logic cells for a circuit that contains \( k \) modules, and \( n_j \) is the logic cell count for each module.

A multiplicative factor does not affect the evolutionary process directly, but is important in the incorporation of penalties associated with hardware “hard limits”. If there are hard limits on the number of available logic cells, and the estimation is not accurate, infeasible solutions can result from the genetic algorithm.

The fitness of each individual solution in the GA is based solely on a function of calculated and estimated parameters determined by the synthesiser. The more accurate the estimate of a circuit’s fitness, the better the GA will work. The compute time and average latency of a circuit is determined precisely by the dataflow analysis algorithm within the synthesiser. The hardware resource usage estimate is very accurate
for memory cells because the information content stored in them can be accurately
determined by the synthesiser. The estimate for the logic cells, however, is often dif-
f erent to the actual hardware resource usage as determined after compilation of the
AHDL design. Reasons for this include the optimisation of boolean expressions by the
compiler, and certain limited resources (such as interconnects) forcing inefficient place-
ment of a circuit. Modules that are used in a circuit and are connected to I/O pins
have different input cell and output cell requirements to the same modules when they
are not connected directly to the I/O. In addition, if a module is defined in the initial
netlist as having an accuracy that is greater than necessary, the redundant portion of
it is removed during compilation, resulting in an inaccurate estimate.

For modules that are defined in AHDL, the estimation of their logic-cell usage was
 gained from the result of actual compilation. It is not feasible to test every case for
the module generators, and so only a subset of all possible generated modules were
compiled to determine their logic cell usage. The other estimates were interpolated
by empirically assessing how the Altera compiler actually instantiated the module.
This often resulted in a piece-wise function as shown by equation 5.10. This equation
describes the logic cell usage of “parallel, lower-half, constant” multipliers of A bits
accuracy. The multiplication is performed on a variable multiplicand of A bits with a
constant (C) also of A bits to produce the lower A bits of the result (where 2 ≤ A ≤ 32).
The formulae derived were:

\[
\text{logic cells required} = \begin{cases} 
1 + \frac{3A}{2} + 2M - L(M + 1) & \text{if } C = 3 \\
A - n & \text{if } C = 2^n \ (n \in 0, 1, 2 \ldots) \\
\frac{3A}{2} + 2M - L(M + 1) & \text{otherwise}
\end{cases}
\]

(5.10)

where

\[
H = \log_2(\text{value represented by most significant '1' in } C) \\
L = \log_2(\text{value represented by least significant '1' in } C) \\
M = \min(A - H, H)
\]

The circuit compute time was determined to be:

\[
\text{circuit compute time (cycles)} = \begin{cases} 
0 & \text{if } C = 2^n \ (n \in 0, 1, 2 \ldots) \\
1 & \text{otherwise}
\end{cases}
\]

(5.11)

Just as the results obtained through compilation are dependent upon the target
FPGA family (Altera 10K), they are also dependent upon the settings of the Max+Plus
II compiler (Version 9.11). The settings were kept the same throughout research,
Table 5.4: Hardware resource requirements for memory-cell based parallel lower-half multipliers implemented in Altera 10K FPLDs. These multipliers only output the lower half of the calculated multiplication product. The resource requirements shown are valid for both signed and unsigned numbers.

<table>
<thead>
<tr>
<th>Module Accuracy (bits)</th>
<th>Memory Cells</th>
<th>Logic Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>192</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1024</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1376</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>2176</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>2560</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>4096</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>4928</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>5888</td>
<td>50</td>
</tr>
<tr>
<td>11</td>
<td>7168</td>
<td>64</td>
</tr>
<tr>
<td>12</td>
<td>9216</td>
<td>77</td>
</tr>
</tbody>
</table>

development, testing, and results stages to provide an accurate comparison between the resulting data. The settings are given in table B.12 in appendix B. An integrated commercial system would grant the GA control over such internal compiler settings.

Module implementations can be based on lookup tables implemented in the special FPGA memory cells known as EABs (Embedded Array Blocks). For modules such as multipliers, the number of EABs required increases exponentially with increasing input accuracy. This limits the supported input accuracies, which means logic cell and memory cell requirements can be determined empirically (because a module for each accuracy can be compiled). Table 5.4 shows the resource requirements for memory-cell based “parallel, lower-half” multipliers. The values for some modules are different, depending on whether or not the inputs are to be treated as signed numbers (shown in table 5.5 for memory-cell based “parallel, full-width” multipliers).

The method used to determine the compression factor for an arbitrary circuit was to generate 50 circuits that used randomly chosen modules and interconnections. After synthesis, estimated and actual results for hardware resource usage were compared and a multiplicative compression factor ($\gamma$) chosen with the aim of reducing the total square error, the formula for which is given in equation 5.12. The process of randomly creating circuit netlists, synthesising into AHDL, compiling with the Altera compiler, and parsing the output files was automated. All the individual programs were launched from the operating system command line (DOS in Windows 95/98 and the Command
Table 5.5: Hardware resource requirements for memory-cell based parallel full-width multipliers implemented in Altera 10K FPLDs. These multipliers output the entire calculated multiplication product as the concatenation of two distinct outputs.

<table>
<thead>
<tr>
<th>Module Accuracy (bits)</th>
<th>Signed Memory Cells</th>
<th>Signed Logic Cells</th>
<th>Unsigned Memory Cells</th>
<th>Unsigned Logic Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>64</td>
<td>4</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>384</td>
<td>6</td>
<td>384</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2048</td>
<td>8</td>
<td>2048</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2592</td>
<td>35</td>
<td>2496</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>4480</td>
<td>43</td>
<td>4480</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>4224</td>
<td>60</td>
<td>4224</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>8192</td>
<td>72</td>
<td>8192</td>
<td>69</td>
</tr>
</tbody>
</table>

Prompt in Windows NT). Perl (Practical Extraction and Report Language\(^3\)) scripts were used to parse and extract data from the output files\([82]\). Perl provides the ability to write terse but very powerful parsing routines using a minimum of code. The following Perl script extracts the integer value, or multiple integers in the case that the design was partitioned over multiple FPGA devices, from the report file produced by the Altera compiler:

```perl
while(<>){chomp;if(/\d+|s+/s+(\d+)/){print"$1\n";}}
```

The total square error, \(E\) is given by

\[
E = \sum_{i=1}^{m} (n_i - N_i)^2
\]

(5.12)

where \(n_i\) is the estimated logic cell count and \(N_i\) is the actual logic cell count for each of the \(m\) random test circuits. The values of \(E\) for various compression factors ranging from 0.5 to 1.1 are given for the 50 randomly generated circuits in figure 5.6. The compression factor that gave the least total square error was:

\[
\gamma = 0.89
\]

(5.13)

Using a value for \(\gamma\) of 0.89, the mean logic cell estimation error for the randomly generated circuits was 20%. This appears high, but was raised by a few circuits where the Altera compiler was able to fit the circuit into a relatively small space. This is often due to redundant hardware as the result of operations on constant values that cause overflows or underflows. Other situations are caused by certain constant values

\(^3\)Pathologically Eclectic Rubbish Lister is also endorsed as a definition by Perl's creator, Larry Wall.
Figure 5.6: Total square error of logic cell estimation versus compression factor. The minimum total square error occurs when the compression factor is equal to 0.89.

(such as 0, 1, $2^n$, or $2^n-1$) that result in implementations differing from the general case assumed by the synthesiser's estimation algorithm. The worst case encountered in the random circuit sample was an estimation 157% higher than the actual logic-cell usage. Only 11% of the estimates were more than 20% (the mean) different to the actual. The median error was 10%. Of the circuit sample, 68% of the logic-cell-usage estimates were within 11% of the actual counts.

As mentioned earlier, memory cell usage can be more accurately estimated, therefore a compression factor is not required for it. In the sample of random circuits used to determine the compression factor, less than one in five of the circuits that used memory-cell-based modules had any discrepancy between the estimated and actual memory-cell usage. The mean absolute error for the memory-cell-based circuit sample was five memory cells per circuit (an average error of 2.3%).

5.6 Selection and Reproduction

Reproduction of individuals into the subsequent generation is based on roulette wheel selection. After an individual's fitness is calculated, the value is scaled as described in section 5.5.1. Individuals are randomly selected for survival into the next generation with a probability proportional to their fitness as a ratio of the total summed population fitness. This is akin to spinning a roulette wheel with each sector's size proportional to
an individual circuit’s fitness. The one difference is that there is at least one copy of the best parent reproduced forcibly. This is to ensure the survival of the best individual over all generations. It was introduced after empirically noting that occasionally the best individual of the run did not breed into the final generation.

There are many methods of selection[28], but roulette-wheel selection was arbitrarily chosen because there is no universally accepted ‘best’ method. Some of the alternative methods are modifications of roulette wheel selection, such as stochastic sampling with replacement, and remainder stochastic sampling. A second common selection technique is based on deterministic sampling where the number of offspring of an individual is directly calculated from its fitness as a proportion of the summed population fitness. Thirdly, there is the stochastic tournament selection regime. In this, pairs of individuals are selected using roulette wheel selection and the individual with the highest fitness is declared the winner and inserted into the new population.

Reproduction occurs by choosing pairs of parents from the new population pool and performing crossover between them with a user defined probability. Wijkman discusses the necessity of crossover in terms of sexual versus asexual reproduction[103]. Mutation is performed on each individual, according to another user defined probability, as it is reproduced.

5.7 Specific Crossover Mechanism

A single-point crossover mechanism is used. Once the genetic algorithm has selected two individuals, the position of the crossover (crossover point) is chosen from along the chromosome with uniform probability.

When crossover occurs in the head of the chromosome it can occur between sets of functionally equivalent modules or within such a set. The former provides a cleaner division, the result being that all the sets of functionally equivalent modules are implemented in the same groups, but the groups are distributed between the two new individuals. The parameters of each group may be altered because a group can be implemented in a different manner in each parent. When crossover occurs within a set of functionally equivalent modules, this allows parts of the set to belong to different groups. At one extreme, there is the possibility that each module belongs to a different group. At the other extreme, all modules that are functionally equivalent could be defined by the parameters of a single group. In this case, the information about unassigned groups is retained in the chromosome tail, but is not used in the calculation of the individual’s fitness. If the information was not retained, it would be lost and not propagated into subsequent generations.
Figure 5.7: Circuit abstraction of an arbitrary four-module circuit. This abstraction is used to demonstrate circuit crossover in figure 5.8. (a) shows the circuit being represented. (b) shows the module groups represented by the shaded regions; light for serial and dark for parallel. Module versions are ignored for simplicity. A white triangle indicates the group is to be implemented as a shared physical instance.

When crossover occurs in the tail portion of the chromosome, each of the individuals gain the attributes for a subset of module groups that the other individual possessed. That is, although each module in each individual still belongs to the same group, the attributes of the group (data type, implementation version, and resource sharing) are possibly different.

A four-module circuit abstraction is given in figure 5.7. This visual abstraction is used to demonstrate the crossover mechanism in figure 5.8.

### 5.8 Specific Mutation Mechanisms

The mutation mechanisms are defined such that they will not produce individuals that are semantically incorrect and cannot be mapped to a real implementation. The mutation occurs to a feature of the individual, with a user-defined probability, as each individual is selected for reproduction (from the pool of recently selected individuals). If the feature selected for mutation is in the head of the chromosome then the feature is a module. In this case the only possibility for mutation is module group mutation. If the feature chosen for mutation is in the tail of the chromosome, then it is a group. One of
Figure 5.8: Crossover demonstration using the circuit abstraction defined in figure 5.7. (a) and (b) show the parents that have the crossover mechanism applied to their chromosomes. From the resulting chromosomes, the children are shown in (c) and (d). Note the ordering of the modules in the chromosome—such that functionally equivalent modules are next to each other.
three possible mutation mechanisms is chosen at random, with equal probability. These are mutations of the group’s data type, implementation version or resource sharing.

5.8.1 Module Group Mutation

The group mutation occurs when the feature selected for mutation is in the head of the chromosome of an individual. This mutation selects a group from those associated with the module function, with uniform probability. In this way, the module being mutated can join a group with other modules, be assigned a group on its own, or stay in the group that it is currently in.

5.8.2 Group Data-Type Mutation

Given that the feature chosen for mutation is in the tail, the probability of choosing this type of mutation is one third. The data type of the selected module group is chosen randomly, with uniform probability, from the set of available data types for the associated functional module. This means that inappropriate (unavailable) module data types are not chosen.

5.8.3 Group Implementation-Version Mutation

This mutation also occurs with a one-third probability when the mutated feature chosen is in the chromosome tail. The way that each module in the group is implemented is chosen with uniform probability from among the available implementation versions. The implementation version being mutated describes the algorithm or method that the module uses. A multiplication module, for example, can be implemented as a combinational circuit, a lookup table, a combination of these, or a pipelined circuit calculating partial results at each stage. Often there is only one implementation of a module that is described by an HDL description, and in such a case this mutation has no effect.

The implementation needs to ensure that the module version indicated by a chromosome is appropriate for the desired module data type. Consider a module such as a multiplier that can be implemented in three different parallel ways (0 to 2) and only two serial ways (0 to 1). If the version is set to a value within the range 0 to 1 when the module is serial then that would unfairly favour these values compared with parallel version 2. Similarly, if the version is left in the chromosome as a value from 0 to 3, it can be interpreted during synthesis by taking its value modulo 3 or modulo 2 for parallel and serial respectively. This is unfairly biased toward version 0. To overcome these biases, the version number stored in the chromosome is taken from the first $V_t$
integers starting at 0. $V_t$ is given by:

$$V_t = V_p \cdot V_s$$

(5.14)

where $V_p$ is the number of parallel versions and $V_s$ is the number of serial versions. During synthesis, the value is interpreted as $(V_t \mod V_p)$ for parallel modules and $(V_t \mod V_s)$ for serial modules. There is no bias toward any particular version, regardless of data type, because $V_t$ is always an exact multiple of the number of module versions of a particular data type.

### 5.8.4 Group Resource-Sharing Mutation

This mutation occurs, as do the other two group (chromosome tail) mutations, with a one-third probability when the mutation is chosen to take place in the chromosome tail. The resource-sharing feature describes whether the modules in the group share the same physical instance. This sharing directly affects the compute time of the circuit and its average latency. This mutation, having two possible values, will not cause the chromosome to represent a meaningless individual solution, because the modules within a group are, by definition, identical.

### 5.9 Distributed Genetic Algorithm Processing

The nature of GA optimisation means that the calculation of an individual’s fitness can be performed independently from that of other individuals. This property means that a distributed implementation of the GA is a natural extension. The genetic operations of mutation are performed by a host application, which controls the evolutionary process. Similarly, the selection and reproduction mechanisms are performed by the host.

The fitness function is calculated by client modules running on the same computer as the host, or on different computers interconnected via a LAN (Local Area Network). The data is distributed amongst the client modules, and controlled by the host module, via a database. Each client module obtains the evolutionary parameters and circuit parameters from the database at start-up. Clients then wait for individuals to process, at which time information about the chromosome that represents the circuit to be analysed is presented to the respective client. On completion of the analysis of an individual, the resulting calculated dataflow and resource-usage parameters are written into the database for access by the host application. Implementation details of the distributed GA processing are given in section 7.7.

Distributed client modules cannot perform fitness function calculations while the host module is processing all the individuals at the end of each generation. This does not
cause a noticeable bottleneck because the time taken for end-of-generation processing is approximately five percent of that required to calculate the fitness of an individual, for any reasonable circuit.

All the client GA processing modules must refer to the same module versions and module generators when extracting information from the encoded individual. This is necessary so that the same individual is not processed to give different results. This was solved by having the host application provide a definitive index for modules and module generators, which the client modules reference.

5.10 Optimisation Algorithm

The optimisation algorithm detail is given in table 5.6, which combines the techniques given earlier in this chapter. Note that to reduce processing time, the parameters and calculated fitness of each individual are stored in the database. This quick-lookup facility means that if an individual has had its fitness calculated earlier in the GA run then it need not be calculated again. The overhead for storage and retrieval is minimal compared with the time saving achieved by avoiding one or more fitness calculations (except for very small circuits of only a few modules, or where the population size and number of generations are extremely large). There is at least one individual in every generation (except the first) that has been analysed earlier (see section 5.6) and the break-even point is usually exceeded through this time saving alone.

5.11 Conclusions

This chapter has presented details of the optimisation algorithm that was used in conjunction with the synchronous-module based synthesiser of chapter 4. The resulting design-space searches are presented in the following chapter.

The circuit optimisation problem was encoded into a form that the genetic algorithm could operate on. This encoding enabled the representation of a solution circuit as a string that was robust to the genetic operators of mutation and crossover. Problem-specific mutation and crossover operators were developed that provide the means to traverse the design space. A fitness function was presented that takes into account various user-defined parameters, which guide the convergence of the GA to an optimal solution. In particular, these parameters define the chosen importance of average latency, logic-cell usage, and memory-cell usage. Fitness scaling was employed to eliminate population stagnation and maintain genetic diversity. The actual hardware resource usage, calculated by the fitting algorithm of the Altera AHDL compiler,
5.11 Conclusions

Step 1 Manually choose optimisation goal and GA parameters (section 5.4.2)
Step 2 Presynthesis redundancy removal (algorithm in table 5.1)
Step 3 For each generation:
   3.1 For each individual in the current generation not stored for ‘quick-lookup’: (Client)
      3.1.1 Perform dataflow analysis of circuit (see chapter 4)
      3.1.2 Estimate time and resource usage
      3.1.3 Modify resource usage estimate with hardware compression factor ($\gamma$)
      3.1.4 Calculate fitness using equation 5.1
      3.1.5 Store fitness for ‘quick-lookup’ in a later generation
   3.2 Scale fitnesses (as shown in section 5.5.1)
   3.3 Perform GA roulette wheel selection and reproduction (section 5.6)
   3.4 GA crossover (section 5.7)
   3.5 GA mutation (section 5.8)
   3.6 Output generation results to a textual log file (example in table B.11)

Step 4 Synthesise best circuit solution in AHDL

Table 5.6: Complete optimisation algorithm used to search the design space. Steps marked with ‘Client’ are performed in the distributed GA client modules; the others are done in the parent application.

differed from that estimated. A “compression factor” was determined that compensates for this difference. Selection and reproduction are based upon the roulette wheel technique.

The optimisation algorithm software was implemented in a distributed format allowing processing to occur over a computer network. Each client process communicates with the host via the underlying database.
Chapter 6

Synthesiser Performance and Design-Space Analysis

6.1 Introduction

The aim of this chapter is to present and interpret the results obtained by applying the optimisation algorithm. This includes an analysis of the circuit design space for each of the circuits studied. The size of each design space depends on the number of different modules that each module can be replaced with while preserving circuit functionality. Of particular interest are the effects of different data types (especially parallel and serial) and the effects of resource sharing in the form of module time-division multiplexing. The encountered circuit solutions, from each searched design space, are directly dependent upon the methods of search and optimisation employed by the synthesiser and genetic algorithm. Therefore, the performance of the optimisation algorithm can be evaluated by interpreting the design-space search results.

The two case studies presented in this chapter are chosen to be data-path oriented circuits. They are made up of modules that perform only a few different functions (for example, a circuit made of multipliers and adders only). Each module is able to be implemented in multiple ways and using different data types.

The aim is to optimise each circuit according to various desired goals that are chosen by observing the all-parallel and all-serial solutions. Goals are chosen from these boundary conditions that range from ‘reasonable’ to ‘highly optimistic’. The optimisation process is analysed by noting the effects (convergence rate and best-circuit quality) of varying the parameters and goals. In addition, the correct operation of the synthesiser on these real-world circuits is verified through simulation of the synthesised HDL.
Each functional circuit design has a design space whose size depends on the parameters being manipulated during a search. The number of possible solution points in the design space can be estimated using the formulae derived in section 6.2.

6.2 Quantitative Design-Space Extent

An understanding of the design space requires an appreciation of the number of circuit solutions that comprise the design space. The first aspect of interest is the number of ways that similar modules can be grouped together. The second is where the different ways of implementing each group are considered. From these combinations, the extent of the design space can be estimated by considering it to be made up of multiple (functionally different) module-groups.

6.2.1 Module Grouping

Consider a circuit made up of functionally equivalent modules—such as adders or multipliers only. The nomenclature \( G(m, i) \) will be used to indicate the number of different arrangements of \( m \) modules into exactly \( i \) groups (where each group contains at least one module). Assuming there are \( m \) modules in the circuit, there is only one way of choosing one group of modules, \( G(m, 1) \), which is where all the modules belong to the lone group. Similarly, there is only one way of choosing a configuration of \( m \) groups of modules, \( G(m, m) \), which is where each module belongs to a unique group. These boundary conditions are given in equation 6.1.

\[
G(m, i) = 1 \quad \text{where } i \in \{1, m\} \tag{6.1}
\]

The general case of choosing \( i \) groups of modules from \( m \) modules can be derived by considering the group configurations of a circuit with one fewer \((m - 1)\) modules. There are two possible ways of adding a module to \((m - 1)\) modules to obtain \( i \) groups. These exist where there are already either \( i \) groups, or \( i - 1 \) groups, in the \((m - 1)\) module circuit. If there are more than \( i \) groups already, the number of groups cannot be reduced by the addition of a module to any group. Similarly, if there are less than \( i - 1 \) groups, then \( i \) groups cannot be obtained by the addition of a new group.

Where there are \( i - 1 \) groups in the \((m - 1)\) module circuit, there is only one way of creating \( i \) groups for \( m \) modules: by the addition of the new module in a new group. Where there are \( i \) groups already, there are \( i \) possible groups that the new module can become a member of to preserve the existing number of groups. The general case for number of group arrangements is given by equation 6.2, which is recursively defined
<table>
<thead>
<tr>
<th>Groups</th>
<th>Group Arrangements</th>
<th>Group Arrangements</th>
<th>Group Arrangements</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>5-Module Circuit, G(5,i)</td>
<td>8-Module Circuit, G(8,i)</td>
<td>10-Module Circuit, G(10,i)</td>
</tr>
<tr>
<td>1</td>
<td>G(5,1) = 1</td>
<td>G(8,1) = 1</td>
<td>G(10,1) = 1</td>
</tr>
<tr>
<td>2</td>
<td>G(5,2) = 15</td>
<td>G(8,2) = 127</td>
<td>G(10,2) = 511</td>
</tr>
<tr>
<td>3</td>
<td>G(5,3) = 25</td>
<td>G(8,3) = 906</td>
<td>G(10,3) = 9330</td>
</tr>
<tr>
<td>4</td>
<td>G(5,4) = 10</td>
<td>G(8,4) = 1701</td>
<td>G(10,4) = 34105</td>
</tr>
<tr>
<td>5</td>
<td>G(5,5) = 1</td>
<td>G(8,5) = 1050</td>
<td>G(10,5) = 42525</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>G(8,6) = 206</td>
<td>G(10,6) = 22827</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>G(8,7) = 28</td>
<td>G(10,7) = 5880</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>G(8,8) = 1</td>
<td>G(10,8) = 750</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>G(10,9) = 45</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>G(10,10) = 1</td>
</tr>
<tr>
<td>Sum</td>
<td>Total = 52</td>
<td>Total = 4140</td>
<td>Total = 115975</td>
</tr>
</tbody>
</table>

Table 6.1: The number of ways of arranging circuit modules into i groups of modules.

and makes use of the boundary conditions given in equation 6.1.

\[
G(m, i) = G(m - 1, i - 1) + iG(m - 1, i) \quad \text{where } i \notin \{1, m\} \quad (6.2)
\]

Table 6.1 shows the number of ways of forming module groups in circuits containing 5, 8, and 10 modules. The totals show the number of different group arrangements for each circuit. The special case of forming exactly two groups of modules is derived in appendix C.

### 6.2.2 Solutions for a Set of Functionally Equivalent Modules

There are many circuit solutions that can be produced by the combination of choosing different module implementations, grouping them together differently, and sharing the resources in some or all of the groups. The first stage of determining the number of circuit solutions is to consider a group of functionally equivalent modules.

In the absence of a formula to exactly calculate the number of solution module arrangements \((S_A)\), the exact number was determined through a software algorithm that exhaustively determined all the combinations. The algorithm could not feasibly be used for groups with more than about 9 modules and 9 implementations because it required too much processing time. Therefore, it was of interest to derive a formula that estimated the result, and could be calculated quickly. A comparison of the number of actual solutions versus the estimates is given in appendix C.
Simple Arrangement Estimation

A simple estimate of solutions that ignores resource sharing can be obtained by:

\[ S_S(m, u) = u^m \]  
(6.3)

where there are \( m \) modules that can be implemented in \( u \) ways. A simple way to include resource sharing is to assume that, because a module can be shared or not, the number of implementations is effectively doubled. This assumption gives a huge overestimate when calculated as:

\[ S_{S2}(m, u) = (2u)^m \]  
(6.4)

Enhanced Arrangement Estimation

The exact calculation relies on being able to determine the sizes of the groups as well as the number of groups. Of particular importance is the occurrence of single-module groups. This is because when there is only one module in a group, its resource sharing is not important.

The estimate of the number of circuit solutions for the set of \( m \) functionally equivalent modules (with \( u \) module implementations\(^1\)) is given by equation 6.5. This equation makes use the partial solutions given by equation 6.6. The derivation of these equations is given in appendix C. Equation 6.5 always overestimates the number of solutions.

\[ S_E(m, u) = \sum_{n=1}^{m} S_e(m, n, u) \quad \{m, n, u\} \geq 1 \]  
(6.5)

\[ S_e(m, n, u) = \begin{cases} 
  u & m = 1 \\
  2u & n = 1 \text{ and } m \geq 2 \\
  \prod_{j=0}^{n-1} (u - j) & n = m \text{ and } m \geq 2 \\
  2(u - 1) \sum_{i=1}^{m-n+1} S_e(m - i, n - 1, u) & \text{otherwise} 
\end{cases} \]  
(6.6)

Even when the number of module arrangements is determined exactly, there are probably far fewer distinct solutions. This is due to the possibility that many of the solutions, although made from different modules, will lie on the same point in the design space (and have the same time and resource requirements). Determining the extent of this overlapping must be done by analysing the data flow and resource usage of each circuit solution in turn.

\(^1\)For example, one parallel and three serial implementations would give \( u = 4 \).
## 6.2 Quantitative Design-Space Extent

<table>
<thead>
<tr>
<th>Modules</th>
<th>Solutions</th>
<th>Solutions</th>
<th>Solutions</th>
<th>Solutions</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>$S_E(m, 1)$</td>
<td>$S_E(m, 2)$</td>
<td>$S_E(m, 4)$</td>
<td>$S_E(m, 6)$</td>
<td>$S_E(m, 8)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>20</td>
<td>42</td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>16</td>
<td>104</td>
<td>312</td>
<td>688</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>52</td>
<td>656</td>
<td>2772</td>
<td>7744</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>156</td>
<td>4568</td>
<td>28432</td>
<td>102894</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>468</td>
<td>32120</td>
<td>309372</td>
<td>1478640</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1404</td>
<td>224840</td>
<td>3402372</td>
<td>22011600</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4212</td>
<td>1573880</td>
<td>37433292</td>
<td>320891776</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>12636</td>
<td>11017160</td>
<td>411766208</td>
<td>4.948336E+09</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>37908</td>
<td>77120120</td>
<td>4.529428E+09</td>
<td>7.422561E+10</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>113724</td>
<td>539840832</td>
<td>4.982372E+10</td>
<td>1.113384E+12</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>341172</td>
<td>3.778886E+09</td>
<td>5.480609E+11</td>
<td>1.670076E+13</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>1023516</td>
<td>2.645220E+10</td>
<td>6.028669E+12</td>
<td>2.505146E+14</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>3070548</td>
<td>1.851654E+11</td>
<td>6.631536E+13</td>
<td>3.757671E+15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>9211644</td>
<td>1.296158E+12</td>
<td>7.294691E+14</td>
<td>5.636507E+16</td>
</tr>
</tbody>
</table>

Table 6.2: The number of estimated circuit solutions for circuits consisting of 1 to 15 functionally equivalent modules. Chosen values for the number of ways of implementing each group, $u$, are 1, 2, 4, 6, and 8.

Table 6.2 shows the number of estimated circuit solutions for circuits containing various numbers of functionally equivalent modules, which can be implemented in different ways.

### 6.2.3 Solutions for Circuits with Functionally Different Modules

The extension of the derivation to circuits that contain functionally different modules makes use of the independence of each set of functionally equivalent modules. That is, the number of solutions for each set of functionally equivalent modules can be worked out from equation 6.5 and multiplied together. Formally, consider a circuit of $n$ sets of functionally equivalent modules. Each set contains $m_n$ modules that can be implemented in $u_n$ ways. The total number of circuit solutions $S_{E_{total}}$ in the design space is given by equation 6.7.

$$S_{E_{total}} = \prod_{i=1}^{n} S_E(m_n, u_n)$$ (6.7)

Equations 6.5 and 6.7 are used to calculate the extent of the design space being searched. Once the size of the design space is known, along with the approximate time
for analysing the circuit, the time taken for an exhaustive search can be estimated. It can be shown that, with the exception of very small circuits, exhaustive design-space searches are infeasible.

Alternatively, the equations can be used to compare the size of a search with the size of the entire design space. For example, this comparison can be used to ensure a certain proportion of the design space is being covered during a search.

6.3 Case Study 1: Matrix Exponentiation Circuit

The first example used in the analysis of the proposed optimisation method is a circuit that takes as its inputs the four fixed-point elements of a $2 \times 2$ matrix. The circuit raises the input matrix to the third power. Each circuit output represents a fixed-point element of the resulting matrix.

The matrix exponentiation circuit comprises 18 modules in addition to its input and output modules. Of these, 10 are multipliers and 8 are adders. The impetus for using a circuit made up of large sets of functionally equivalent modules is to see how each set is manipulated by the genetic algorithm, and what the resulting design space features are. Any data-path oriented circuit can be considered to be a number of groups of functionally equivalent modules, and so the evolutionary results for a single group form the building blocks for an analysis of the whole evolutionary design and optimisation method.

6.3.1 Circuit Topology and Operation

For the matrix-cubing circuit example, the inputs were arbitrarily set to an accuracy of 7 bits each, with no binary places. The desired number of pipelined circuit initiations was set to 5. The derivation of the circuit's design, built upon scalar modules, is given below.

The circuit output matrix $B$ is:

$$B = A^3 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^3$$

(6.8)

Therefore, to implement the circuit by decomposing the circuit functionality into scalar operations, the output can be written as:

$$B = \begin{bmatrix} a_{11}^3 + 2a_{11}a_{12}a_{21} + a_{12}a_{21}a_{22} & a_{11}^2a_{12} + a_{12}^2a_{21} + a_{11}a_{12}a_{22} + a_{12}a_{22}^2 \\ a_{11}^2a_{21} + a_{21}^2a_{12} + a_{11}a_{21}a_{22} + a_{21}a_{22}^2 & a_{22}^3 + 2a_{12}a_{21}a_{22} + a_{11}a_{12}a_{21} \end{bmatrix}$$

(6.9)
The most common terms that can be implemented by existing modules are extracted from the algebraic expansions and represented by a column vector \( \mathbf{K} \) as defined in equation 6.10.

\[
\mathbf{K} = \begin{bmatrix}
a_{11}^2 \\
a_{12}a_{21} \\
a_{11}a_{22} \\
a_{22}^2
\end{bmatrix}
\] (6.10)

The elements of vector \( \mathbf{K} \) can be substituted into equation 6.9 to give:

\[
\mathbf{B} = \begin{bmatrix}
k_1a_{11} + k_2(2a_{11} + a_{22}) & a_{12}(k_1 + k_2 + k_3 + k_4) \\
a_{21}(k_1 + k_2 + k_3 + k_4) & k_2(a_{11} + 2a_{22}) + k_3a_{22}
\end{bmatrix}
\] (6.11)

Figure 6.1 shows the functional structure of the matrix exponentiation circuit, which resulted from using the common scalar terms as the basis of the implementation. The accuracies for each module were calculated by deriving the value for the output node of each module and determining what its minimum and maximum values could possibly be, given the input extrema.

The example below shows the numerical functionality of the circuit. The chosen example input array is:

\[
\mathbf{A} = \begin{bmatrix}
-33 & -45 \\
12 & -61
\end{bmatrix}
\]

The output of the circuit is:

\[
\mathbf{B} = \mathbf{A}^3 = \begin{bmatrix}
-33 & -45 \\
12 & -61
\end{bmatrix}^3 = \begin{bmatrix}
32643 & -282735 \\
75396 & -143281
\end{bmatrix}
\]

Represented in 21 bits (modulo \( 2^{21} \)), the example input translates to:

\[
\mathbf{A} = \begin{bmatrix}
2097119 & 2097107 \\
12 & 2097091
\end{bmatrix}
\]

The output, represented in 21 bits, is:

\[
\mathbf{B} = \begin{bmatrix}
2097119 & 2097107 \\
12 & 2097091
\end{bmatrix}^3 = \begin{bmatrix}
32643 & 18144\text{1}7 \\
75396 & 195387\text{1}
\end{bmatrix}
\]

The operation of the circuit (including the synthesised control unit) was verified using Altera Max+Plus II. The simulator utilises accurate device models for its timing calculations. The simulation clock period was chosen to be 200ns; a frequency of 5MHz.
Figure 6.1: Circuit for raising a $2 \times 2$ matrix to the third power. The labels for each module are in the form ‘X:Y:Z’ where X is the module number, Y is the module function, and Z is the minimum required accuracy. Each matrix element $a$ in the input matrix is restricted to $-63 \leq a \leq 63$ to avoid overflow errors within the circuit. (The input accuracy is nonetheless shown as 21 bits to match the output accuracy.)
The multipliers used in the circuit are 'lower-half' multipliers (\texttt{mul}(lh)) in that they output only the lower \( L \) bits of the result (correctly signed), where each multiplicand is accurate to \( L \) bits. Two LAB-based data type implementations are able to be synthesised by the module generators: parallel and serial. A third implementation is a (fully-combinational) parallel implementation that uses the Altera 10K memory (EAB) cells. This is not included in the GA because of the limited accuracies it is able to be synthesised for. There is currently a single version of the serial multiplier and five versions of the parallel multiplier implementation.

The resource usage, and processing time, of the multiplier and adder modules were approximated empirically, as discussed in section 4.6.1. For an \( L \) bit serial multiplier module, the number of cycles is \( 2L \) and the number of logic cells is \( (6L - 1) \). The logic cell usage for a parallel multiplier is given in table 4.12. Its compute time ranges from 0 to 4 cycles and it can be reinitiated after 1 cycle. There is one adder implementation version for a parallel data type and one version for a serial data type. The logic cell usages are \( (2L - 2) \) and 2 cells respectively, where \( L \) is the desired module accuracy. Both implementations are combinational and so the number of cycles is constant at zero.

### 6.3.2 Analysis Approach

The circuit was initially synthesised four times in an attempt to gauge the extent of the design space. The four solutions considered were in two sets: all the modules parallel (with compute time equal to three cycles) and all the modules serial. For each data type, the circuit was synthesised with all the modules instantiated separately and then with as much module resource sharing as possible. In each of the four trials, parallel input and output modules were used, as this is not usually a design choice. Moreover, it provides a better comparison of the circuit resource usages by having consistent I/O data types (thus forcing the inclusion of converter modules where appropriate). Table 6.3 gives the results of these trials.

These preliminary designs were used to choose goals for the GA by providing a conceptual solution boundary. The three design goals are illustrated in relation to the initial estimated design space extent in figure 6.2. The linear interconnections on the graph give an indication of the tradeoff between cells and cycles. The three design goals were:

1. **Optimistic (2000 cells, 20 cycles):** Both the number of cells and the number of cycles are greater than the minimum values found in the initial designs. This solution lies outside the closed curve shown in figure 6.2.
<table>
<thead>
<tr>
<th></th>
<th>Parallel</th>
<th></th>
<th>Serial</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>None</td>
<td>All</td>
<td>None</td>
<td>All</td>
</tr>
<tr>
<td>Summed 10K Cells</td>
<td>4706</td>
<td>3700</td>
<td>1334</td>
<td>13959</td>
</tr>
<tr>
<td>Estimated 10K Cells</td>
<td>4188</td>
<td>3293</td>
<td>1365</td>
<td>12424</td>
</tr>
<tr>
<td>Actual 10K Cells</td>
<td>3899</td>
<td>3008</td>
<td>1536</td>
<td>12263</td>
</tr>
<tr>
<td>Compute Time (Cycles)</td>
<td>6</td>
<td>13</td>
<td>89</td>
<td>630</td>
</tr>
<tr>
<td>Average Latency (Cycles)</td>
<td>1.00</td>
<td>14.00</td>
<td>63.00</td>
<td>630.00</td>
</tr>
<tr>
<td>Synthesis Time (Seconds)</td>
<td>14</td>
<td>17</td>
<td>22</td>
<td>31</td>
</tr>
<tr>
<td>Compilation Time (Seconds)</td>
<td>508</td>
<td>195</td>
<td>58</td>
<td>643</td>
</tr>
</tbody>
</table>

Table 6.3: Salient results for matrix exponentiation circuit resource usage (exclusive of control-unit circuitry). ‘None Shared’ indicates that each multiplier and adder is implemented as a distinct module instance. ‘All Shared’ indicates that the circuit contains only one multiplier and one adder that are shared in time (at the maximum accuracy of the shared modules). ‘Synthesis Time’ is for the synchronous module-based synthesiser. ‘Compilation Time’ is for Altera Max+Plus II. Times given are for relative comparisons. ‘Summed 10K Cells’ does not account for the hardware compression factor (see section 5.5.2) whereas the ‘Estimated 10K Cells’ does. ‘Actual 10K Cells’ is the output from Altera Max+Plus II.

2. Very Optimistic (800 cells, 20 cycles): The number of cells is less than the minimum of the initial solutions, while the number of cycles is greater than the minimum number of cycles in the initial solutions. This solution also lies outside the closed curve shown in figure 6.2.

3. Realistic (3000 cells, 60 cycles): As with the optimistic goal, both the number of cells and the number of cycles are greater than the minimum values of the initial solutions. An important difference is that the solution lies within the closed curve of figure 6.2.

The GA was run three times for each of the three goals. An analysis of the optimisations and design spaces resulting from these runs is presented in the remainder of this section.

Modules that utilise memory cells were excluded from the optimisation process to provide tradeoffs that were easier to interpret (and suitable for three-dimensional surface representations). The elimination of the memory cells simplifies the fitness function (equation 5.1) to:

\[ f = -pp_{p_c}(n_{k_l} + n_{k_c}) \]  

(6.12)
Figure 6.2: Initial design space extent and target solutions for matrix circuit. The closed curve is formed by linearly connecting the four design-space solutions. This gives a conceptual representation of the effect of trading off cells versus cycles. The o, *, and + symbols represent the positions in the design space of the three evolutionary goals respectively.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Trial a</th>
<th>Trial b</th>
<th>Trial c</th>
</tr>
</thead>
<tbody>
<tr>
<td>[p.mutation]</td>
<td>0.03</td>
<td>0.30</td>
<td>0.09</td>
</tr>
<tr>
<td>[p.crossover]</td>
<td>0.80</td>
<td>0.90</td>
<td>0.60</td>
</tr>
<tr>
<td>[optimise.time]</td>
<td>0.100</td>
<td>0.100</td>
<td>0.100</td>
</tr>
<tr>
<td>[optimise.size]</td>
<td>0.004</td>
<td>0.004</td>
<td>0.001</td>
</tr>
<tr>
<td>[size.hard.penalty]</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>[time.hard.penalty]</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td><strong>Cells</strong></td>
<td>1521.01</td>
<td>1365.26</td>
<td>3105.21</td>
</tr>
<tr>
<td><strong>Cycles</strong></td>
<td>404.00</td>
<td>404.00</td>
<td>25.00</td>
</tr>
</tbody>
</table>

Table 6.4: Matrix exponentiation circuit results for optimisation goal 1.

All GA runs in this section were configured to analyse the pipelined target circuit for five iterations. Each population had 30 individuals, and each optimisation trial was run for 50 generations. Modules were chosen from various versions of parallel and serial implementations.

### 6.3.3 Optimisation Results

The following subsections present the results for the three design optimisation goals: optimistic, very optimistic, and realistic. The results are analysed in the subsequent sections. One of the most important outcomes of the analysis is a methodology for choosing the optimisation goal parameters. Initially the choice of design parameters was made empirically, through parameter modification after observing the results of prior runs. Analysis of the optimisation process is done to provide an interpretation of the design space. In addition, an approach is developed that can be used for choosing optimisation parameters for future runs.

#### Goal 1: Optimistic

This circuit target goal was 2000 cells and 20 cycles. The three optimisation runs are summarised in table 6.4. In the absence of modules that utilise memory cells, the values of the parameters optimise.time and optimise.size describe the tradeoff between cells and cycles. For example, parameter values of 0.100 and 0.004 respectively indicate that 4 cycles has the same fitness value as 100 cells. These were initially chosen by trying to estimate the importance of time and size, and what increase in circuit size would be subjectively ‘acceptable’ for an associated decrease in processing time (or vice versa).

Figure 6.3 shows the maximum, median and minimum fitnesses for each generation
Figure 6.3: Fitness optimisation curves for matrix circuit goal 1, trial a. The lines represent maximum, median, and minimum fitness from top to bottom.

of the first trial. The three lines formed are equivalent to three vertical slices of the subsequent fitness surface graphs. Figure 6.4 shows the fitness surface for the first trial, obtained by ordering each generation’s individual fitnesses numerically and graphing the changes through time. These graphs both show convergence to a near homogeneous population before generation 15. Note that the fitness values shown depend on the parameters used in the fitness function, and so runs with different parameters cannot be quantitatively compared in terms of their fitnesses. Figures 6.5 and 6.6 show similar graphs with cell count and cycle count, respectively, on the vertical axes. A mathematical combination of these two graphs is used to produce the fitness surface.

To prevent early convergence, the mutation probability was increased for the second trial. Figure 6.7 shows the fitness surface for the second trial of the first goal showing less convergence. The optimisation parameters were further adjusted to coerce the GA to converge to a more optimal solution.

Goal 2: Very Optimistic

This goal aims to evolve a circuit that has 800 cells and takes 20 cycles for the 5 circuit iterations. Table 6.5 summarises the results of the three runs for the second goal.
Figure 6.4: Fitness optimisation surface for matrix circuit goal 1, trial a.

Figure 6.5: Cell-count optimisation surface for matrix circuit goal 1, trial a.
Figure 6.6: Cycle-count optimisation surface for matrix circuit goal 1, trial a.

Figure 6.7: Fitness optimisation surface for matrix circuit goal 1, trial b.
Table 6.5: Matrix exponentiation circuit results for optimisation goal 2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Trial a</th>
<th>Trial b</th>
<th>Trial c</th>
</tr>
</thead>
<tbody>
<tr>
<td>[p.mutation]</td>
<td>0.00</td>
<td>0.25</td>
<td>0.35</td>
</tr>
<tr>
<td>[p.crossover]</td>
<td>0.70</td>
<td>0.70</td>
<td>0.90</td>
</tr>
<tr>
<td>[optimise.time]</td>
<td>0.100</td>
<td>0.100</td>
<td>0.100</td>
</tr>
<tr>
<td>[optimise.size]</td>
<td>0.001</td>
<td>0.010</td>
<td>0.010</td>
</tr>
<tr>
<td>[size.hard.penalty]</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>[time.hard.penalty]</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Cells</td>
<td>3550.21</td>
<td>1365.26</td>
<td>1380.39</td>
</tr>
<tr>
<td>Cycles</td>
<td>20.00</td>
<td>404.00</td>
<td>404.00</td>
</tr>
</tbody>
</table>

Figure 6.8: Fitness optimisation surface for matrix circuit goal 2, trial a.

Figures 6.8, 6.9, and 6.10 give the fitness surface, cell-count surface, and cycle-count surface for the first run respectively.

The mutation probability was increased slightly for each trial to assist in finding an optimal solution and to avoid early convergence. Similarly, the crossover probability, and time and size penalties were adjusted through subjective observation of each previous trial.

Goal 3: Realistic

Goal three sees the optimisation algorithm aiming for a circuit solution that has 3000 cells and takes 60 cycles. Table 6.6 summarises the three trials for goal three. Figure 6.11 shows the fitness surface for trial one. Figures 6.12 and 6.13 show the cell-count
Figure 6.9: Cell-count optimisation surface for matrix circuit goal 2, trial a.

Figure 6.10: Cycle-count optimisation surface for matrix circuit goal 2, trial a.


<table>
<thead>
<tr>
<th>Parameter</th>
<th>Trial a</th>
<th>Trial b</th>
<th>Trial c</th>
</tr>
</thead>
<tbody>
<tr>
<td>[p.mutation]</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>[p.crossover]</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>[optimise.time]</td>
<td>0.100</td>
<td>0.100</td>
<td>0.100</td>
</tr>
<tr>
<td>[optimise.size]</td>
<td>0.001</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>[size.hard.penalty]</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>[time.hard.penalty]</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>Cells</strong></td>
<td>1375.05</td>
<td>1375.94</td>
<td>3761.14</td>
</tr>
<tr>
<td><strong>Cycles</strong></td>
<td>404.00</td>
<td>404.00</td>
<td>14.00</td>
</tr>
</tbody>
</table>

Table 6.6: Matrix exponentiation circuit results for optimisation goal 3.

Figure 6.11: Fitness optimisation surface for matrix circuit goal 3, trial a.

surface for trials two and three respectively, which have identical optimisation parameters. The second and third trials show how a GA with identical parameters can converge to markedly different solutions.

### 6.3.4 Design Space

Each multiplier module can be implemented either as parallel or serial. The parallel multipliers can be implemented in five ways, with compute times ranging from zero cycles (purely combinational) to four cycles. This produces six implementations \((u = 6)\). There are ten multiplier modules in the circuit \((m = 10)\), so the application of equation 6.5 yields:

\[
S_E(10,6) = 4.529428 \times 10^9
\]
Figure 6.12: Cell-count optimisation surface for matrix circuit goal 3, trial b.

Figure 6.13: Cell-count optimisation surface for matrix circuit goal 3, trial c.
Figure 6.14: Explored design space for matrix circuit goal 1, trial a. The ellipses denote design clustering and are described in section 6.6.

Similarly, the eight adder modules can each be implemented in two ways, giving \( S_E(8,2) = 4212 \) arrangement combinations. The total number of circuit solutions is therefore:

\[
S_E(10,6) \cdot S_E(8,2) = 1.907795 \times 10^{13}
\]

During a typical GA run of 30 individuals and 50 generations, a maximum of 1500 designs would be encountered—equating to a very small proportion of the whole design space.

Figure 6.14 shows the points in the design space representing solutions found during the evolution of the matrix exponentiation circuit goal 1, trial a. These design-space scatter plots are analysed in section 6.6.

### 6.4 Case Study 2: Artificial Neural Network Circuit

The ANN circuit was designed to analyse real echocardiogram data from a well-known corpus.\(^2\) The corpus describes health features and survival information for a group of

\(^2\)The echocardiogram data (echocardiogram.data) is available from many locations on the internet. One such source is http://www.ics.uci.edu/pub/machine-learning-databases/echocardiogram/
132 post heart-attack patients. Eleven of the attributes were used as inputs to the ANN and one was the binary output describing whether an individual survived for at least one year after the heart attack.

The attributes describe physiological aspects of the patient such as age at heart attack, whether there was fluid around the heart (pericardial effusion), various dimensional abnormalities, and a measure of how the segments of the left ventricle were moving.

The aim was to train an artificial neural network to be able to analyse echocardiogram data from a heart attack patient and determine whether the individual will survive for at least twelve subsequent months. This circuit was then optimised using a genetic algorithm and synthesised for implementation in an FPGA.

The first step was to train a multi-layer perceptron network using SNNS (Stuttgart Neural Network Simulator). This required each input and output to have its training data normalised between 0 and 1. Of the data vectors for the 132 patients, 120 were used to train the ANN. The remaining 12 were used to test the ability of the neural network to generalise.

The output from SNNS was a neural network with four hidden neurons and one output neuron. Figure 6.15 shows the neuron interconnections and network topology as produced by SNNS. Note that the eleven input neurons are only connections and do not have weights and do not implement an activation function.

### 6.4.1 Circuit Topology and Operation

The ANN circuit was implemented as a netlist of 214 modules. The circuit was mostly made up of multipliers, adders and constants to form the multiply-and-sum function of each of the five hidden and output neurons. In addition there was a sigmoidal activation function module for each of these neurons.

The weights and bias values were stored as constant modules, which were then multiplied by variable inputs in multiplier modules. The reason for not choosing the more compact multiply-by-constant modules was to allow the GA to share the resources of the multipliers. Such resource sharing is not possible with multipliers that are defined to multiply by a single constant value (where the constants differ between multipliers). In addition, the whole neural network was exploded into one netlist from its hierarchical design consisting of neuron circuits that themselves consist of synapse and weight circuits. Again, this was done to provide the greatest design space scope for the GA to search within.

The module accuracies were chosen empirically through simulation by a simple
Figure 6.15: Artificial neural network topology as output from SNNS. Each square represents a neuron; input, hidden, and output from left to right. The number above each neuron is its identifier. The number below each neuron is its output value for a certain input data set. (The input data set appears as the outputs of the input layer. The output neuron output is the ANN's overall output.) The lines interconnecting the neurons show the physical connections, which are assigned weights during the training process.
C++ program by analysing the output error introduced into the circuit. The program emulated the fixed point arithmetic that would exist in the final circuit implementation. The inputs and outputs of each neuron had real values between 0 and 1 and were chosen to have 6 bp (binary places—to the right of the binary point). The constants were represented by 12 bit values, also with 6 bp. The lowest 6 binary places were truncated after the first multiplication of these two values leaving 12 bits with 6 bp. The internal circuitry had accuracies up to 18 bits (6 bp) to prevent numerical overflows.

The sigmoidal activation function module was implemented in EABs, and the number of values in the lookup table was found to affect the performance of the ANN. The activation function used by SNNS during ANN training is given in equation 6.13, where $net$ is the bias constant plus the sum of each neuron input multiplied by the corresponding weight constant. With the constants derived by SNNS, $|net|$ for any of the five neurons cannot be greater than 61.8. It was decided that where $net \leq -4$, and $net \geq 4$ the output of the activation function could be approximated to 0 and 1 respectively. This introduces a range error outside the domain $\pm 4$ of less than 1.8%.

$$Output = \frac{1}{1+e^{-net}} \quad (6.13)$$

The remaining domain of $-4 < net < 4$ was divided into $2^n$ equal segments to optimise LUT size versus the error introduced by the approximations. A separate C++ simulation of the neural network was used to analyse the effect of different numbers of LUT segments on the output of the network.\textsuperscript{3} The results are summarised in table 6.7. From these simulations it was found that using $2^5 = 32$ segments had a minimal impact on the accuracy of the neural network, while producing a relatively compact LUT. This equates to a numerical input resolution of 0.25. In this regime, the sigmoid output ranged from (binary) 0000 to 1111, and represented a fixed-point number between 0 and 1. This value had two binary placeholders appended to match the input expected by the subsequent neuron (6 bits to the right of the binary point).

The circuit operation was verified, using the *Max+Plus II* simulation environment, by the application of binary encoded normalised test inputs.

### 6.4.2 Analysis Approach

The ANN circuit was optimised for five iterations of the target circuit. Modules were chosen from parallel and serial implementations. The input and output modules were parallel, as were the constants. The implementation of the sigmoidal activation function was parallel and was based on EABs.

\textsuperscript{3}In a similar vein, Holt and Baker\textsuperscript{[35]} discuss the effects of limited precision calculations during the backpropagation training stage.
<table>
<thead>
<tr>
<th>Input Accuracy (Bits)</th>
<th>Output Accuracy (Bits)</th>
<th>Correct $\epsilon \leq 5%$</th>
<th>Not Exact $5% &lt; \epsilon &lt; 40%$</th>
<th>Wrong $\epsilon \geq 40%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
<td>129</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>129</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>126</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>124</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>122</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>121</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>122</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>122</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>118</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>114</td>
<td>0</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 6.7: The effect of different neuron activation function resolutions on the neural network output. The absolute percentage error in the network output is denoted by $\epsilon$. The error categories were chosen arbitrarily (in that 5% error was deemed acceptable, while 40% was not). The number given is the number of outputs in the various error categories for the 132 input vectors. The 'float' accuracies are for a floating point simulation of the activation function. Note that a static network output of 0 (as in the 1,1 case) gives the maximum of 18 wrong outputs due to the distribution of the test data.
<table>
<thead>
<tr>
<th></th>
<th>Parallel</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summed 10K Logic Cells</td>
<td>25787</td>
<td>9042</td>
</tr>
<tr>
<td>Estimated 10K Logic Cells</td>
<td>22933</td>
<td>8848</td>
</tr>
<tr>
<td>Actual 10K Logic Cells</td>
<td>7771</td>
<td>8851</td>
</tr>
<tr>
<td>Estimated 10K Memory Cells</td>
<td>800</td>
<td>800</td>
</tr>
<tr>
<td>Actual 10K Memory Cells</td>
<td>800</td>
<td>800</td>
</tr>
<tr>
<td>Compute Time (Cycles)</td>
<td>8</td>
<td>155</td>
</tr>
<tr>
<td>Average Latency (Cycles)</td>
<td>1.00</td>
<td>57.00</td>
</tr>
</tbody>
</table>

Table 6.8: Salient results for artificial neural network circuit resource usage (exclusive of control-unit circuitry). ‘Summed 10K Logic Cells’ does not account for the hardware compression factor whereas the ‘Estimated 10K Logic Cells’ does. The serial implementation uses some parallel modules as discussed in the accompanying text.

Two initial designs were synthesised, one using all parallel modules, and the other using serial modules where possible. This gave a starting reference in the design space. The results of these initial trials are given in table 6.8.

From the initial synthesis, it can be seen that the synthesiser produced a very close approximation of the serial circuit. Conversely, the estimation for the parallel circuit was considerably larger than the actual resource usage. This was due to using multiplier modules and constant modules instead of multiply-by-constant modules. The effect of this is high degree of redundancy, with the particular constants used. Consider a weight constant with a value of 2. It makes the immediately subsequent multiplier redundant (it can be implemented as a static connection shift arrangement). In this case, the shift operation makes the lowest bit redundant, which affects subsequent circuitry. This inaccuracy affects the estimation of a particular circuit with given constant weights and biases. If the final weight constants have not been chosen, or could change with subsequent retraining of the network, then the actual compilation is not representative of the general case. Effectively the synthesiser is giving a worst case estimation in its analysis.

The evolution approach was to choose a target for the GA to evolve towards and to adjust the optimisation parameters to modify the search performance. The initial GA run was on a population of 12 individuals for 15 generations. The reason these were chosen to be less than for the matrix exponentiation circuit was because the larger ANN circuit required about 22 hours on the synthesis PC to analyse the resulting 180 solution points. The goal was to evolve the circuit to comply with the (estimated)
Table 6.9: Optimisation parameters and results for the artificial neural network circuit.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Trial a</th>
<th>Trial b</th>
<th>Trial c</th>
<th>Trial d</th>
</tr>
</thead>
<tbody>
<tr>
<td>[population]</td>
<td>12</td>
<td>12</td>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>[generations]</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>[p.mutation]</td>
<td>0.05</td>
<td>0.18</td>
<td>0.25</td>
<td>0.35</td>
</tr>
<tr>
<td>[p.crossover]</td>
<td>0.85</td>
<td>0.85</td>
<td>0.80</td>
<td>0.60</td>
</tr>
<tr>
<td>[optimise.time]</td>
<td>0.100</td>
<td>0.100</td>
<td>0.100</td>
<td>0.100</td>
</tr>
<tr>
<td>[optimise.size]</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0003</td>
</tr>
<tr>
<td>[optimise.mem]</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>[size.hard.limit]</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>[size.hard.penalty]</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>[time.hard.limit]</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>[time.hard.penalty]</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Logic Cells</td>
<td>28749.67</td>
<td>27012.39</td>
<td>26622.57</td>
<td>27953.12</td>
</tr>
<tr>
<td>Cycles</td>
<td>449.00</td>
<td>471.00</td>
<td>728.00</td>
<td>425.00</td>
</tr>
<tr>
<td>Memory Cells</td>
<td>640</td>
<td>640</td>
<td>800</td>
<td>640</td>
</tr>
</tbody>
</table>

6.4.3 Optimisation Results

The circuit was evolved four times with different optimisation parameters. The results are summarised in table 6.9. The fitness surfaces describing the circuit evolutions are given for each evolution run. The scatter plots of the design spaces are discussed in section 6.6.

The first evolutionary run (trial a) produced the fitness surface shown in figure 6.16. The optimisation parameters for the second trial were modified to reduce the rate of convergence to a homogeneous population that the first trial exhibited. Specifically, the mutation rate was increased from 0.05 to 0.18. The fitness surface for the second evolutionary trial is shown in figure 6.17.

To further increase the diversity of the population, the population size was increased to 18 for the third trial. The crossover rate was reduced to 0.8 and the mutation rate was increased to 0.25. The resulting fitness surface is shown in figure 6.18.

The fourth trial was performed with a population size of 24 individuals. The mutation rate was increased to 0.35 and the crossover rate was reduced to 0.60 to increase population diversity. The hard penalties were also both reduced from 3 to 1 to allow extended exploration into undesirable—but possibly useful—areas of the design space.

\footnote{The discussion of cells in this artificial neural network section implies 10K logic cells.}
Figure 6.16: Fitness optimisation surface for neural network circuit, trial a.

Figure 6.17: Fitness optimisation surface for neural network circuit, trial b.
Figure 6.18: Fitness optimisation surface for neural network circuit, trial c.

It is possible for solutions with very undesirable fitness measurements to contain very desirable subcircuit characteristics. The fitness surface for the fourth trial is shown in figure 6.19.

6.4.4 Design Space

The ANN circuit comprises 214 modules (before synthesis and the addition of auxiliary modules such as data type converters and delays). The input, output, and const modules can each be implemented in only one way (and cannot utilise resource sharing), so they are ignored by the design space calculation.

Each sigmoidal activation function requires 160 EABs. The maximum requirement is 800 EAB cells for the entire network. This can be minimised during optimisation through resource sharing, although the GA was not guided towards such minimisation, as it was assumed that there were enough EAB resources available.

The modules that can be implemented in various forms are 48 multipliers (mult(1h)), 48 adders (add), 48 divide-by-constants (divconst) and 5 bus-splitters (splitbus). The number of ways of implementing each group for each of these module functions are \( u = 6, u = 2, u = 1, \) and \( u = 1 \) respectively. The total number of circuit solutions, emphasising the extent of the design space, is:

\[
S_E(48,6) \cdot S_E(48,2) \cdot S_E(48,1) \cdot S_E(5,1)
\]

which calculates to a value much greater than \( 10^{100} \).

\[5\] The const modules are used to store numerical constants for synaptic weights and biases.
Figure 6.19: Fitness optimisation surface for neural network circuit, trial d.

The design space is analysed in section 6.6, with scatter plots showing the subset of designs that were encountered during the four evolution runs.

6.5 Optimisation Performance

The optimisation performance of the genetic algorithm can be discussed in terms of convergence to, and proximity to, the target design constraints. To reduce the barrage of insignificant digits, cycle and cell counts in this section are rounded to the nearest integer.

The first matrix exponentiation circuit goal was 2000 cells and 20 cycles. The first trial produced the closest cell count with 1521 cells and 404 cycles. The third trial produced the closest cycle count with 3105 cells and 25 cycles. The second trial exhibited the most diversity in the population throughout the evolution, but did not produce any exceptional solutions.

The second matrix circuit goal was 800 cells and 20 cycles. The solution having the closest cell count to the desired count was produced by the second trial, with 1365 cells and 404 cycles. The desired cycle count was achieved by the first trial, with a cell count of 3550. Again, the trial with the most diversity did not produce either of the more optimal solutions.
The third matrix circuit goal was 3000 cells and 60 cycles. The third trial produced the closest circuit solution both in terms of cells and cycles, with 3761 cells and 14 cycles. This trial had the slowest convergence to the fitness surface plateau of the third goal.

The artificial neural network circuit goal was 10000 cells and 30 cycles. The third trial produced the closest cell count with 26623 cells and 728 cycles. The closest cycle count was produced in the fourth trial with 27953 cells and 425 cycles. As shown by figures 6.16 through 6.19, the fourth trial had much more diversity than the others. This was due to the modified optimisation parameters—largely the increase in population size.

The graphs for matrix circuit goal 1, trial a (figures 6.4, 6.5, and 6.6) show that individual solution points with more optimal cell counts or cycle counts need not necessarily produce better fitnesses than solutions with a more optimal mix. Although this is clear from the fitness function equation, graphical representation of individual components of the fitness function can be misleading. A similar situation occurs in matrix circuit goal 2, trial a (figures 6.8, 6.9, and 6.10).

Of particular interest are the last two trials for optimisation goal 3 for the matrix circuit. These two optimisations had identical optimisation parameters but yielded greatly different results due to the random nature of the optimisation algorithm. The two solutions occur while the algorithm is converging towards local minima. The two cell-count surfaces produced by these trials (figures 6.12 and 6.13) are quite different in their structure.

### 6.5.1 Determining Optimisation Parameters

The choice of optimisation parameters provides the designer with the ability to guide the evolution and tweak its performance. The initial values for the optimisation parameters were set subjectively, using values obtained through trial and error and by observing the values used by other authors[28].

In general, the aim was to maintain diversity in the population by adjusting the various parameters, while ensuring convergence to a solution. Mutation rates and population sizes were increased to enhance population diversity. Crossover rates were chosen to ensure genetic information was shared throughout the population, while not stagnating the population with homogeneity. The guiding parameters—the hard penalties—were also modified to guide the evolution. It was observed that increasing them forced the solutions towards the desired area of the design space, while reducing them allow solution diversity.
For the matrix circuit optimisation, the closest optimised solutions occurred in the first trial in two cases, in the second trial in one case, and in the third trial in three cases\textsuperscript{6}. It might be expected that the genetic algorithm would work better in later trials, as the optimisation parameters were adjusted. The artificial neural network evolution improved in the latter trials. The closest circuits were in trials three and four for cell count and cycle count respectively. Many more trials would be necessary before conclusions can be drawn from such statistical evidence.

Genetic algorithm mutation rates are often in the order of 0.01, depending on the application. The mutation rates used in this thesis were higher (0.05 to 0.35) to maintain the population diversity. The effects of using different mutation rates were observed from the fitness surface graphs. The mutation rates had to be relatively high due to the large number of mutations that did not affect a circuit's fitness. For example, the mutation of a module version or data type that only has a single version or data type causes no physical change to the circuit. Similarly, the mutation of a property to its current value has no effect.

It would be possible to computationally analyse the homogeneity of the population during the GA run and modify the optimisation parameters on-the-fly, but this would involve a separate study. The diversity cannot be maintained in all situations through simple fitness scaling (for example, in homogeneous populations).

### 6.5.2 Design-Space Fitness-Surface Transformation

Once the design space has been searched, the best individual—the one that is most suitably matched to the design criteria—can be chosen. Graphically, the fitness of each design can be represented by a vertical bar rising from the associated solution point on the base plane formed by the cell-versus-cycle design space. Visually, these graphs are difficult to interpret because of the clusterings of the designs in the design space, and the fitnesses that are not simply correlated with each solution's position on the design-space plane. Moreover, the surface formed by joining the fitness peaks together is very complex.

If the design criteria change, the associated fitness function is modified. A new fitness surface can be produced through the application of the new fitness function to each previously calculated point in the design-space plane. This forms a new fitness surface from which the solution with the maximum fitness can be chosen. The advantage this approach gives is that the time-consuming algorithms for determining each

\textsuperscript{6}Counting goal 3, trial a, as two best-cases as it was the closest in terms of cell count and in terms of cycle count.
solution do not have to be re-run. The ‘search’ is effectively a random walk, and does not replace goal-seeking search methods, but can be used in conjunction with them. In particular, if similar goals are being used in successive evolutionary searches, the results from previous searches can have their solution fitnesses transformed to supplement the current design space.

The advantage of calculating the fitness of each point, rather than just exhaustively traversing the design space and comparing values of cell-count and cycle-count to some desired values, is that it provides an objective representation of the design goal—necessary for an automated process.

6.6 Design-Space Feature Analysis

The subsets of the design space encountered during each evolutionary run were analysed in terms of the clustering of the solution points. The clusterings were considered as clusters within the design space (macroscopically) and as individual designs within the clusters (microscopically). These two aspects of the analysis are presented in sections 6.6.1 and 6.6.2 respectively. The full design spaces being searched are themselves subsets of an infinitely large conceptual design space for the design of a circuit with a given functionality.

Some of the matrix exponentiation circuit’s design-space subsets are presented in figures 6.20 to 6.23. The subsets encountered in the ANN’s design-space are shown in figures 6.24 to 6.27. The ellipses on the planes were added to emphasise design clustering by grouping visual clusters.

6.6.1 Macroscopic Features

The macroscopic features describe the spread of the clusters in the design space. These clusters are obvious from the design space plots of figures 6.20 to 6.27.

The clusters are generally vertical on the plots of the design space. This indicates they contain a wide range of cell counts while having very similar cycle counts.

The first two evolutionary trials of the ANN circuit had no visually discernable clusters. These trials were run with the smallest numerical product of population size and number of generations.

Each point on the design space scatter plot may have occurred repeatedly during the GA run. The number of times each solution point occurred during the evolution of the matrix circuit goal 3, trial a, is shown in figure 6.28. This shows a typically observed distribution, where most of solution points are encountered only once, but a
Figure 6.20: Explored design space for matrix circuit goal 1, trial b.

Figure 6.21: Explored design space for matrix circuit goal 1, trial c.
Figure 6.22: Explored design space for matrix circuit goal 2, trial a.

Figure 6.23: Explored design space for matrix circuit goal 3, trial a.
Figure 6.24: Explored design space for neural network circuit, trial a.

Figure 6.25: Explored design space for neural network circuit, trial b.
Figure 6.26: Explored design space for neural network circuit, trial c.

Figure 6.27: Explored design space for neural network circuit, trial d.
Figure 6.28: Solution repetitions for matrix circuit goal 3, trial a. This is a typical plot showing the number of times each distinct solution point was encountered during the genetic algorithm run.

few are encountered hundreds of times.

Figure 6.23 showed three clusters (of which the rightmost one is, perhaps, questionable). Partitioning the design space into bands of 100 cycles in width, a distribution of the solutions can be obtained. In this case, 1398 of the 1530 solutions (91%) occur between 400 and 500 cycles—the leftmost cluster. This includes the two most highly repeated solution points, encountered 262 and 154 times during the evolution. A further 53 solutions (3%) lie between 700 and 800 cycles—the middle cluster. Many of the 100-cycle bands contain no solutions.

The first explored design space to have its features analysed is matrix circuit goal 3, trial a. Overall, the design space utilised 27% parallel modules and 4% of the modules shared their resources. There was a general trend of decreasing use of parallel modules with increasing cycle count from about 80% to 30% across the design space. This is shown in figure 6.29. There was no obvious correlation between resource sharing and cycle count. As expected, higher cell counts were produced with the use of more parallel modules. The proportion of parallel modules versus cell count increased from 20% to 70% about two thirds of the way up the design space and then decreased to 50%. The
Figure 6.29: Spread of module data-type across design space with respect to cycle-count. Data extracted from the resulting design space of the evolution of matrix circuit goal 3, trial a. The explored design space was partitioned into bands of 50 cycles.

A correlation between resource sharing and cell count was indistinct. Approximately, module resource sharing increased from 0% to 40% up the design space. This is shown graphically in figure 6.30.

Not all the evolutionary runs for the same circuit exhibited the same trends. Matrix circuit goal 2, trial a, showed a decreasing use of parallel modules with increasing cell count. This reduced from 100% to 60% about halfway up the design space and then leveled. In addition, the overall features of the design space were quite different to the previous example. Resource sharing was used by 32% of the modules and 91% of the modules were parallel.

The neural network circuit evolutionary goal 3 showed a decrease of module resource-sharing from 26% to 20% with respect to increasing cell-count. Module resource-sharing was static across the design space at 22%. Parallel module usage did not alter with respect to either axis of the design space and was static at 74%. The overall design space features were 74% parallel modules and 22% resource sharing.

The neural network goal 4 had overall features of 75% parallel modules and 15% module resource-sharing. Parallel module usage was fairly static at 75% over both axes. Resource sharing increased from 10% to 20% with respect to cycles, and from 15% to 30% with respect to cell count.
Figure 6.30: Spread of module-sharing across design space with respect to cell-count. Data extracted from the resulting design space of the evolution of matrix circuit goal 3, trial a. The explored design space is partitioned into bands of 200 cells.

6.6.2 Microscopic Features

The microscopic features describe the spread of circuit designs within the clusters in the design space. Selected clusters were analysed by comparing their circuit features (cell-count and cycle-count) with those of their relevant design spaces.

The leftmost cluster in figure 6.23 showed a similar relationship between parallel module usage and cell-count to that observed in the whole design space. There was a correlation between resource sharing and cell count at about one third the magnitude of the whole design space. The middle cluster showed very similar trends to the whole design space.

The leftmost cluster in figure 6.27 exhibited the same distribution of parallel module usage versus cell-count as the whole design space—albeit over a smaller domain. Similarly, module resource-sharing for the cluster was simply a subset of the results for the whole design space.

Most of the clusters did not display any clear trend for parallel module usage or module resource-sharing with respect to cycle-count. The clusters’ vertical orientations meant they were narrow and so there were small domains over which to observe any trends.
Table 6.10: Best-circuit features for matrix circuit optimisation goal 2, trial b. Usage is shown as a ratio of the number of modules of the given function. The nomenclature \( \text{V}_x=y/z \) means that \( y \) of the \( z \) modules were implemented as version \( x \).

<table>
<thead>
<tr>
<th>Module</th>
<th>Modules Used</th>
<th>Shared Modules</th>
<th>Parallel Modules</th>
<th>Module Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>8</td>
<td>0/8</td>
<td>0/8</td>
<td>V0=8/8</td>
</tr>
<tr>
<td>input</td>
<td>1</td>
<td>0/1</td>
<td>1/1</td>
<td>V0=1/1</td>
</tr>
<tr>
<td>mult (1h)</td>
<td>10</td>
<td>0/10</td>
<td>0/10</td>
<td>V0=3/10, V2=1/10, V3=3/10, V4=3/10</td>
</tr>
<tr>
<td>output</td>
<td>1</td>
<td>0/1</td>
<td>1/1</td>
<td>V0=1/1</td>
</tr>
</tbody>
</table>

Table 6.11: Best-circuit features for matrix circuit optimisation goal 3, trial c. Usage is shown as a ratio of the number of modules of the given function.

Analysis of the Winning Solutions

The best circuits from each case study were selected. The features of these circuits are described below.

The best circuit evolved for the matrix exponentiation circuit goal 2, trial b, is outlined in Table 6.10. It does not make use of resource sharing and utilises serial modules where possible. Module versions were chosen fairly uniformly from those available, except no use was made of version 1 of the multiplier module. The best circuit evolved for goal 3 during trial c is described in Table 6.11. This circuit differs from that just discussed in that it uses all parallel modules. Two of the ten multiplier modules shared their resources. Versions 1 and 4 of the multipliers were not utilised.

The best solutions created during the evolution of the neural network circuit trials c and d are presented in Tables 6.12 and 6.13 respectively. Trial c used more resource sharing than did trial d. The most common multipliers in trial c were version 4, while trial d favoured versions 1, 2 and 3 similarly. Slightly more of the multipliers were parallel in the latter trial.
### Table 6.12: Best-circuit features for neural network optimisation trial c. Usage is shown as a ratio of the number of modules of the given function.

<table>
<thead>
<tr>
<th>Module Function</th>
<th>Modules Used</th>
<th>Shared Modules</th>
<th>Parallel Modules</th>
<th>Module Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>48</td>
<td>13/48</td>
<td>14/48</td>
<td>V0=48/48</td>
</tr>
<tr>
<td>const</td>
<td>53</td>
<td>0/53</td>
<td>53/53</td>
<td>V0=53/53</td>
</tr>
<tr>
<td>divconst</td>
<td>48</td>
<td>24/48</td>
<td>48/48</td>
<td>V0=48/48</td>
</tr>
<tr>
<td>function</td>
<td>5</td>
<td>0/5</td>
<td>5/5</td>
<td>V0=5/5</td>
</tr>
<tr>
<td>input</td>
<td>1</td>
<td>0/1</td>
<td>1/1</td>
<td>V0=1/1</td>
</tr>
<tr>
<td>mult(lh)</td>
<td>48</td>
<td>9/48</td>
<td>23/48</td>
<td>V0=8/48, V1=7/48, V2=10/48 ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V3=4/48, V4=19/48</td>
</tr>
<tr>
<td>output</td>
<td>1</td>
<td>0/1</td>
<td>1/1</td>
<td>V0=1/1</td>
</tr>
<tr>
<td>presig</td>
<td>5</td>
<td>0/5</td>
<td>5/5</td>
<td>V0=3/5, V1=2/5</td>
</tr>
<tr>
<td>splitbus</td>
<td>5</td>
<td>2/5</td>
<td>4/5</td>
<td>V0=5/5</td>
</tr>
</tbody>
</table>

### Table 6.13: Best-circuit features for neural network optimisation trial d. Usage is shown as a ratio of the number of modules of the given function.

<table>
<thead>
<tr>
<th>Module Function</th>
<th>Modules Used</th>
<th>Shared Modules</th>
<th>Parallel Modules</th>
<th>Module Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>48</td>
<td>0/48</td>
<td>16/48</td>
<td>V0=48/48</td>
</tr>
<tr>
<td>const</td>
<td>53</td>
<td>0/53</td>
<td>53/53</td>
<td>V0=53/53</td>
</tr>
<tr>
<td>divconst</td>
<td>48</td>
<td>15/48</td>
<td>48/48</td>
<td>V0=48/48</td>
</tr>
<tr>
<td>function</td>
<td>5</td>
<td>2/5</td>
<td>5/5</td>
<td>V0=5/5</td>
</tr>
<tr>
<td>input</td>
<td>1</td>
<td>0/1</td>
<td>1/1</td>
<td>V0=1/1</td>
</tr>
<tr>
<td>mult(lh)</td>
<td>48</td>
<td>7/48</td>
<td>30/48</td>
<td>V0=13/48, V1=6/48, V2=12/48 ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V3=12/48, V4=5/48</td>
</tr>
<tr>
<td>output</td>
<td>1</td>
<td>0/1</td>
<td>1/1</td>
<td>V0=1/1</td>
</tr>
<tr>
<td>presig</td>
<td>5</td>
<td>0/5</td>
<td>5/5</td>
<td>V0=3/5, V1=2/5</td>
</tr>
<tr>
<td>splitbus</td>
<td>5</td>
<td>3/5</td>
<td>1/5</td>
<td>V0=5/5</td>
</tr>
</tbody>
</table>
6.7 Conclusions

This chapter initially presented a method for quantitatively determining the extent of circuit design spaces searched by the genetic algorithm.

A case study was then presented, which constituted a circuit for raising a $2 \times 2$ matrix of fixed point numbers to the third power. The second case study was an artificial neural network for determining life expectancy after a heart attack. These circuits were evolved, given various constraints and various optimisation parameters. The optimisation parameters were modified according to subjective evaluation in subsequent evolutionary trials to maintain diversity while still converging to a good solution.

The searched design spaces for both circuits exhibited clustering of solution design points. The only resulting design spaces showing no evident clusters were those with the lowest numbers of searched points. The clusters generally covered a narrow range of cycles and a wide range of cells.

The evolution runs showed convergence towards the desired goals, although most did not fully attain them. The fitness surface graphs showed the diversity throughout the optimisation runs, and how the diversity could be maintained through appropriate choices for the optimisation parameters. The fitness function was effective, as it allowed the searches to be guided towards the desired solutions. An empirical analysis of the fitness function, and the possible development of a better one, would require a separate study. Fitness functions could be compared in terms of their time to converge, quality of the best found solution, and the robustness of convergence in a wide variety of search spaces.

The resulting design spaces showed varying levels of correlation between parallel module usage and resource sharing with respect to cell-count and cycle count. Some were too noisy to determine trends, particularly in the cases having the fewest points in the design space.

The best solutions found by the genetic algorithm, for each circuit goal, favoured a mixture of module implementations. As expected, optimisations constrained by circuit run-time favoured parallel modules. Those constrained by circuit size favoured serial modules and resource sharing. Most solutions were evolved to make use of parallel and serial modules of different versions, and to include some degree of module resource sharing.
Chapter 7

Synthesiser Software
Development and
Implementation Discussion

7.1 Introduction

This chapter discusses the synthesis and optimisation software, in addition to other associated software, that has been presented in this thesis, from a software development perspective. The maturation of the source code and the progression of the designs are presented. Development paradigms are discussed, as are the choices of development tools. Issues related to software design and coding are also raised. These issues are important because software development is at the critical phase of becoming a formal discipline with many new methods and techniques becoming available. Often the process is ignored and the result deemed predominantly important. Naturally this is a dangerous approach as developers are now creating single systems consisting of nearly thirty million lines of code.¹

The level of software engineering design performed at the outset of a project, in addition to the choices of tools and supported operating systems, dramatically affect the life of a software project. In the synthesiser software presented in this thesis, a combination of compiled object oriented and run-time interpreted languages were used to produce the HDL code to be further compiled. Many levels of debugging were necessary, which highlighted the complexity of software development of this type.

¹The figure associated with Microsoft's Windows 2000.
7.2 Software Engineering Issues

This following four subsections discuss some of the points that relate to competent software engineering.

7.2.1 Software Engineering

As more people are given access to software development through more technically accessible tools, especially people from non-technical backgrounds, software integrity must be considered. The origin of software becomes an issue, and as software development slowly matures into an engineering discipline, the issue of licensing software developers needs to be raised—it was technically illegal to use the title “software engineer” in the United States before August 1996[14].

The distinction should be emphasised between programmers and software engineers, as they are responsible for different tasks. Software engineers are responsible for the project (cost models, quality control, and program design), while the programmers are in charge of writing the source code and implementation of the required design. The tasks often do overlap, and in small companies the software engineer is often a programmer as well. Dorchester points out that in a more structured regime “code writers” will not necessarily be licensed as professional engineers, and that qualified software engineers may—or may not—be highly competent at writing code[19].

An extension to certification of the engineer is certification of the software quality. Software would incite more confidence if an independent review of the software certified claims[100] such as:

- The code has no malicious behaviour (including Trojan Horses, Easter Eggs, viruses, or trap doors).
- Up-to-date development and testing processes were employed.
- The developers have passed professional exams demonstrating programming proficiency.

Such certification standards would reduce the hidden costs of poor software quality. These standards could be defined by companies or by professional organisations. The drawback of companies, or consortiums of companies, defining standards, is that they will tend to favour their own products and make things harder for their competitors. Arguably, the role of defining standards should be given to non-commercial professional organisations[58], such as the IEEE (Institution of Electrical and Electronic Engineers) or ISO (International Organization for Standardization).
7.2.2 Software Design and Quality Management

The licensing of Software Engineers should go some way towards promoting quality software, but more fundamental are the design methodologies and quality assurance schemes that are in place. As part of the quality certification process, a project should be well specified. In addition to this fundamental requirement, the software project management process[72] continues as:

**Estimation:** Estimates of the project using past measurements, estimates of aspects of the software to be produced, and preparation of the requirements specification.

**Risk Analysis:** Identification, assessment, and prioritisation of the project and technical risks associated with the project.

**Scheduling:** Similar to scheduling in any other engineering project.

**Tracking and Control:** Monitoring conformance to the project schedule and the effects of tasks falling behind schedule.

All these steps can be estimated and given numerical values (metrics). The importance of having metrics for various aspects of a project, such as productivity and quality, is that it is very difficult to analyse and compare these aspects otherwise. With metrics, analysis can be performed to see if quality is being improved for each subsequent project, for example. Once there are ways to define the quality of software produced, methods exist that ensure that the required quality standards are met[10]. One example is the presentation by Tackett et al. of a state-based development process that assists in the control and development of error-free software[91].

In relation to software engineering economics, the costs associated with a project can be estimated; one of the most popular models being Barry Boehm's Constructive Cost Model (Cocomo)[72]. Using attributes pertaining to the product, operating hardware, personnel, and project, the Cocomo model can be used to estimate the number of people required for the project as well as its duration. (Only to a limited extent can duration and number of people be traded off against each other.) The analysis of real-world projects, such as that performed by the US Army Missile Command, shows that real organisations can benefit from a well-organized metrics program[30].

One proposed advantage of the publicity surrounding the Year 2000 date problems is that it should make industry more aware of the need to apply formal processes in the future[58] (in this case including the specification and documentation of software operating boundaries).
7.2.3 Development-Team Issues

With a multi-person software development team, human factors begin to play a role in
the development of the software product. These human factors need to be considered
along with the purely theoretical calculations of the number of programmers per line of
code. Weinberg discusses the ramifications that human ego has on software development
teams, concluding that it is valuable for programmers to accept their humanity and
that they are prone to making errors. This allows egoless peer review of software and
the associated improvement in quality that this review process brings[101].

Programming teams, as with other professional teams, are fundamentally based on
either hierarchical, or egalitarian, team structures. The issue of interacting personalities
can, on the one hand, be disastrous for a project, but can also, on the other hand,
benefit a project. The aim for the members of programming team is to grow, starting
from dependency through conflict to cohesion and interdependency[51]. In this way, a
successful team can be built.

7.2.4 Coding Issues

There are a number of issues that should be considered during the coding phase of
software development. Some are outlined in this section that directly or indirectly
relate to the development of the synthesiser code.

The specifications of languages such as C and C++ leave many details dependent
upon the implementation so as to make the best use of available hardware[22, 67]. To
a certain extent this reduces the portability of code. For example, the result of the %
operator is implementation defined, as are the number of bits in long, int, and short
integers (with certain constraints).

Passing by (hidden) reference should be avoided. That is, a C++ function with
the prototype: int add(int& a, int& b); could modify the values of a and b as well
as returning a result. It is possible to call this function with numerical constants as
in c=add(5,6);. The Borland C++ Builder warning-message description states “The
warning means that the reference variable or parameter does not refer to what you
expect ... ” It is generally clearer to make the programmer aware that his or her
arguments may be modified by forcing a pointer to be passed. The prototype should
thus be: int add(int* a, int* b);. Wiener argues that C++ is not representative
of object-oriented languages, and that better ones (such as Eiffel) exist and are prefer-
able. Moreover, it is suggested that C++ is not suited to large projects because of its
hybrid nature that allows several programming styles: procedural, object-based, and
object-oriented[102].
Overloaded operators can cause concern in regard to deep copying or shallow copying. On assignment of one object to another, the data members could be copied in full, or their pointers could be copied (as in a bitwise copy of the object). Moreover, problems can be caused when a programmer ignores conventional wisdom, such as by returning from many points within a function. Other problems can be caused by undue optimisation—the introduction of global variables, large functions, or overly-complex pointer arithmetic[81].

Third-party debugging tools should be considered as part of the programmers toolkit. Tools such as Memory Sleuth monitor the dynamic memory allocation and deallocation of class objects, arrays, and variables. It warns of non-freed memory and tools such as this provide a great resource in tracking down stray memory-leak bugs. Other tools such as Lint give a thorough analysis of the code for possible errors, and many issues are raised as warnings. Some compilers include this type of checking with a strict compiler-option.

Finally, mention must be made of the choice of programming language and how it affects the overall project. Powerful, expression-based languages such as C and C++ allow the programmer a great deal of freedom, but at the expense of being weakly-typed languages[67]. C programmers can write terse, even obfuscated, code.2 This allows errors to seep in for the unwary programmer. More verbose languages, such as Pascal, that have strong type-casting, reduce these types of errors but increase coding time. Stroustrup comments that “C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg.”[98]

7.3 Development Paradigms

As the synthesiser software matured, many development paradigms were considered. At different stages in the development, different criteria became important, and so different methodologies were appropriate. This section outlines some of the software development paradigms that were encountered in the evolution of the synthesiser software; in particular issues of portability, database usage, and program interface.

7.3.1 Portability Issues

The stochastic neural network synthesiser was written to be the most portable of all the presented synthesisers. It was written in ANSI C++ and used the command-line based standard input/output library. It runs on many platforms (Unix and Dos

2There is even an International Obfuscated C Code Competition (IOCCC).
based), but the code-base was relatively complex as no advantage could be made of development-tool specific, or operating-system specific, high-level features. This can be contrasted with the final version of the synchronous module based synthesiser that utilised many available high-level features, but was limited to operating under 32 bit Microsoft Windows environments.

The final version of the synthesiser used many Windows-based and C++ Builder-based features as discussed on page 174 (BDE, SQL, VCL, DLLs, and the Win32 Registry). In addition to these, there is another more subtle portability issue: the source code utilises extensions that Borland have made to C++. These language extensions are optional, but are necessary to realise the full power of Rapid Application Development (RAD) environments. Reisdorph debates the use of language extensions[73] arguing that some should be part of the C++ standard. Common examples of language extensions in Borland's C++ are:

 closures This keyword is used for (class) method pointers. It associates an object instance pointer with a method pointer so that the correct method (using data from the appropriate instance) can be called.

catch This keyword is equivalent to the finally keyword in Java. The code in this block is always called whether or not a try block exits normally or whether control is intercepted by a catch clause exception handler.

property This keyword is used to describe a cross between a variable and a class. It effectively overloads the assignment operator so that a method is called when the variable’s value is set or retrieved.

An associated issue to portability is that of code re-usability. If code is designed to be used in a number of applications, developed with different tools and for different operating systems, then this affects how portable the code should be. It also affects the level of documentation required for the source code.

7.3.2 Relational Database Engines

The use of relational databases as opposed to object oriented (manually programmed) storage for small applications polarises the software development community. There are legitimate advantages and disadvantages to using a relational database at the heart of a small application; some of these are outlined below.

Relational databases are generally manipulated using SQL (Structured Query Language), which has become the standard relational database language[32]. SQL plays many roles including being:
7.3 Development Paradigms

An interactive query language: To retrieve and display data on screen in an ad-hoc manner.

A database programming language: For embedding SQL into applications or report-writing and data-entry utilities.

A database administration language: For database management tasks such as for defining the structure of a database, and user access to the data.

A client/server language: Client computers using SQL to access data on a server database.

A distributed database language: Multiple systems communicating via SQL for data access requests.

A database gateway language: For interfacing different database products and brands.

SQL is a powerful query language, described as a fourth-generation language (4GL). The higher the language generation, the greater the level of abstraction at which code can be written[72]. Other 4GLs include formal specification languages and prototyping languages. Less abstract third-generation languages include Pascal, Modula-2, C, and Ada. Second-generation languages include Fortran and Cobol. The least abstract, first-generation languages, are the closest to the actual physical implementation of the computer running the software. Examples of first generation languages are Machine Code and Assembly Language.

It should be noted that not all SQL implementations are created equal. The generic subset of SQL, called SQL92, that was used for the synthesiser application did not have many of the powerful commands found in an implementation such as Oracle (such as the null-value conversion command: NVLO, or a command that performs like an ‘if-then-elseif’: DECODE()). This lack of commands had the effect of increasing the required code-size of SQL statements. It also required an increased interaction between the C++ code and the embedded SQL queries. (Embedded SQL is written within another programming language such as C or Pascal and data is transferred from the programming language code to and from the database.) The advantage of using a generic subset of SQL is that it is database engine independent. This means that the database engine can be changed in the background—providing scalability for small applications[37].

The use of a database means that memory management is automatically handled by the database engine. Multi-level loops to search for data stored in arrays are removed. Subtle effects, such as those caused by accessing all the elements in an array while its length is changing, are also handled by the database engine when using tables as
a storage medium. SQL interaction with a database also allows a log of all database commands to be stored, such that the database can be accurately recreated to any point in time. Database integrity keys in a single table or between multiple tables automatically issue an error at run time when incongruent data is being inserted into the database.

Databases are optimised to be efficient at processing and manipulating large amounts of data. They are not necessarily fast at performing in embedded environments, where the overhead of parsing the SQL code by the database engine is great in comparison to the (often single row of) data processed by each query in a loop. Embedded SQL effectively introduces large volumes of global data into an application. Every time the data is required, it must be re-queried from the database, because the application cannot be sure that the data has not changed. Finally, the use of database engines alters the way debugging is performed because the SQL queries are, in most cases, only parsed and interpreted at run-time. This means that not only semantic, but syntactic errors are not found until run-time.

Overall, the use of a database engine and SQL-based manipulation of the data simplified the coding and debugging of the synthesiser application. It enabled easy manual interaction with the database during operation to check the integrity of stored data. It also enabled readily interpretable code to be written and debugged. On the down-side, it slowed the performance of the synthesiser. Kernighan argues that optimisation of software should be attempted only if the program is genuinely too slow, and adds that "A fast program that gets the wrong answer doesn't save any time."[40]

### 7.3.3 Graphical versus Command Line Interfaces

Both Graphical User Interfaces (GUI) and Command Line Interfaces (CLI) play important roles in the interaction between user and application. During the testing phase of software, each of these interfaces becomes important when various aspects are being scrutinised. The GUI is important near the beginning of the application testing cycle—with manual design entry, while the CLI becomes important in advanced, automated testing.

The GUI is important for visual interpretation and verification of input to an application, so that erroneous input is not mistaken for a software bug early in the testing phase. The synchronous-module based synthesiser has a graphical entry tool, which is discussed in detail in section 4.8. It was written in 32 bit C++ code for Microsoft Windows platforms, utilising the general interactive features of the Windows environment. Actions such as moving modules, and making and breaking connections are simple click
and drag operations. A simple algorithm is used to draw the connections, such that they endeavour to avoid being drawn over modules or other connection lines. This type of feature is fundamentally important for a graphical interface because the input must remain interpretable to the user even when it becomes relatively complex. Similarly, hierarchical constructs are allowed for design entry; allowing detail to be abstracted from the designer.

The CLI is important for the automation of testing. The synchronous-module based synthesiser utilised a CLI that is described in section B.2 of appendix B. A separate utility was developed that generated random circuits and then executed the main synthesiser application. This was useful, for example, in the derivation of a hardware compression factor (as discussed in chapter 5). Other automation that utilised the CLI was the repeated synthesis and optimisation of a circuit given various design and optimisation parameters and criteria.

### 7.4 Development Tools

As software development tools become increasingly powerful, they hide many of the details of software development from the software engineer. This abstraction has obvious positive benefits, allowing more people to write code (or drag software components) and develop software utilities. The drawback of evermore (internally) complex development tools is that they can hide important sinister interactions between different code fragments. Moreover, sometimes the development tools themselves contain bugs, which can severely increase source code debugging time. This section does not intend to slur any particular development tool, environment, operating system or company.

The need for Rapid Application Development coerces software developers to use the high-level features of a development tool. These features are usually proprietary, and may be software objects or compiler settings. The use of these features creates a high degree of reliance upon a particular development tool. It is not usually trivial to port a large base of code from a development environment, such as Borland C++ Builder to another, such as Microsoft Visual C++ (which is an advantage for the development tool companies). Problems created by this reliance upon a particular development tool occur when unsupported functionality is required or bugs are found in the tool. As mentioned earlier in this chapter, these bugs do happen, and when bug-fixes are not available this can cause problems for a software project. Boyle et al. propose formal methods for guaranteeing that a program continues to do what it should after the compilation process; from specification to assembly language[11].

A more subtle problem than development tool bugs are low-level interactions that
are hidden from the software developer. This is a price that is paid for the ability to program and describe software operation at a very high level, with much of the detail abstracted. Often these interactions are dependent upon the operating system flavour, and the versions of dynamically linked libraries.

Software developers should accept bugs in development tools as they have come to accept bugs in other software, even, occasionally, operating systems. The prudent programmer should always try to be aware of available patches, bug-fixes, and updated notes for developers.

7.5 The Stochastic Neural Network Synthesiser

The synthesiser software was written in C++ and tested under Microsoft DOS and SunOS UNIX operating systems. An object oriented approach was taken where each class represents a specific piece of the neural network hardware such as a neuron, synapse or the entire network. Each of these classes contain private data members that store parameters relevant to the class, such as an array of synapses included in a particular neuron, its activation function type, and its bias value. These private data members are accessible via public methods. This method of abstraction allows run-time error checking to be performed that can identify illegal value assignments.

The software core utilises a two pass algorithm. During the first pass, the input text file is read and parsed, and an array of neurons and interconnections is created. Also, the integrity of the network structure is checked to ensure that the description conforms to fundamental requirements such as the correct number of inputs and outputs.

The second pass through the algorithm is where the AHDL text design files are created. Each software class that represents a hardware block contains a public method (visible to non-derived classes) to start the synthesis of the block description. In addition, private methods (hidden to other non-derived classes) exist specifically for the synthesis of defined portions of the block. For example, in the neuron class, one specific method is used to synthesise the activation function (depending upon the type chosen). Another method synthesises the connections with the synapses and biases. At an even lower level, there exist private methods to synthesise the prototypes of blocks included from further down the AHDL project hierarchy, and the input and output descriptions of the AHDL text design file being created.
7.6 Evolution of the Synthesiser Source Code

There were three major versions of the synchronous-module based synthesiser. Each version had increased functionality over the previous and took advantage of the latest development tools. The three versions are:

**Version 1: Platform-independent object-oriented C++ code**

This version of the synthesiser uses predefined synchronous modules and a netlist of modules interconnected such that there were no feedback connections. The C++ source was coded in a portable way so that no operating-system-specific features were used. In addition, no development-tool dependent features were used, as these are often also platform dependent. The synthesiser consisted of 69KB of source code, or 2.3 KLOC (thousand lines of code).

The advantage of this method is that the code can be ported to a number of platforms that had C++ development support. The disadvantage is that, of the three versions, the coding time per line-of-code was the greatest. In addition, debugging time was lengthy because advantage was not taken of encapsulated system-dependent classes, so all low level functions such as dynamic memory management had to be explicitly performed by the code.

**Version 2: VCL-based object-oriented C++ code**

This version of the synthesiser was created to include hardware pipelining of data. This time, Borland's Visual Component Library (VCL) was utilised, which greatly reduced the portability of the code. Objects such as AnsiString were used to encapsulate data and methods. This particular class performed string functions and handled all associated dynamic memory management, thus reducing the number of pointer and array errors, and memory leaks. The synthesiser consisted of 329KB (9.7 KLOC) of source code.

The resulting code ran slower than its counterpart written at a low level would. However, the relative coding and debugging times were reduced. The disadvantage was that a good understanding of the VCL objects had to be gained. Moreover, some of the low level control was seized from the programmer, and where the low-level functionality was not completely documented, tests on the classes had to be performed. Introducing the code to another programmer also required their gaining familiarity with the VCL.

Issues such as shallow versus deep copying of classes became an issue and errors in interpretation had disastrous effects on the program's stability. Probably the greatest
problem with using a library such as the VCL is when it contains bugs. These can be very difficult to locate. In fact, the subscript operator ([ ]) of AnsiString achieved fame as the “Windows Developer’s Journal ‘Bug of the Month’.” Effectively, the copy-on-write technique (used to create efficient copy and assignment operations) did not work for the subscript operator such that when two strings point to the same data (supposedly temporarily) and one modifies the contents using the subscript operator, the strings are not differentiated!

**Version 3: BDE-based SQL and object-oriented C++ code (utilising DLLs)**

This third version of the synthesiser provides the greatest functionality of the three. It is able to process pipelined circuits that contain feedback. In addition, parameterised module generators are supported. These are implemented in Dynamic Linked Libraries (DLLs), which are used by the main synthesiser to produce the HDL for an appropriate module given various parameters (such as data type, accuracy in bits, and implementation version). As with the second version of the synthesiser, this version also utilises the VCL to simplify coding, debugging, and maintenance of the code.

The synthesiser also utilises the Win32 Registry for storing configuration data during synthesis and between synthesis runs. This had the advantage of using standard VCL and API (Application Programming Interface) calls—thus simplifying code writing and debugging, while providing storage of the configuration in a form that any programmer can access easily. The configuration data effectively becomes global though, and so programmers must use caution when updating the data.

The major difference in implementation between this synthesiser and previous versions is its foundation upon the Borland Database Engine (BDE). Powerful SQL commands were issued that handled the manipulation of complex data with only a few lines of code. The third version of the synthesiser consisted of 951KB (25.3 KLOC) of C++ and SQL92 source code, including the module generators’ source code.

### 7.6.1 Synthesiser Debugging

There were many distinct levels of debugging required for the final version of the synchronous-module based synthesiser (using the BDE). These debugging levels can be summarised as follows:

**Synthesiser compiled-source:** These errors are caused by C++ code that cannot be compiled. They are found at compile-time.

**Synthesiser interpreted-source:** These errors are caused by incorrect SQL—often caused by tables not being available at appropriate times (because they have not
been created elsewhere in the program, for example). These errors are found at run-time.

**Synthesiser crashing:** These are fatal run-time errors in the C++ (and possibly SQL) code.

**Synthesiser functionality:** These errors or warnings are raised by the software itself. They dictate that the program has entered a state that it should not have. This can be due to erroneous input files or semantic code errors (C++ or SQL).

**AHDL interpretation:** These errors are interpreted by the synthesiser developer when analysing the AHDL output. They indicate that the AHDL will not perform with the desired functionality.

**AHDL compilation:** These errors are found during compilation of the synthesised AHDL. They indicate that the synthesiser has produced syntactically incorrect AHDL code.

**AHDL functionality:** This is where the final errors are evident. AHDL code that has correctly compiled shows it is not performing correctly during functional tests (via simulation or on-device experimentation).

For the debugging of the AHDL functionality and the AHDL interpretation errors, it is important that the synthesised AHDL code is human readable. This takes extra effort during writing of the synthesis routines, but pays off in the long run. To enable easily readable synthesised AHDL code, features are included such as inline documentation, indented code, and neatly laid out code that is purposefully non-obfuscated.

### 7.6.2 Module Generators

Only the final version of the synchronous-module based synthesiser utilised module generators as were discussed in section 4.6.

To reduce the need for a large number of AHDL files, provision has been made for extensions to the main synthesiser in the form of Dynamic Link Libraries (DLLs). A DLL is a program, or collection of functions, that must be called by another program. It can not be executed on its own. Each DLL is called by the main synthesiser and is capable of synthesising one or more functional modules for a range of various parameters. There are general parameters, such as accuracy and data type, as well as module specific ones, such as delay duration and whether a number should be treated as being signed. All of the appropriate DLLs are loaded into memory when the main synthesiser is initialised. After this, their functions can be called with no additional time penalty.

---

3 One for each function, accuracy, data type combination, for example.
The main advantages of using DLLs are that features can be added to the main program without having to recompile it, and each DLL can be developed using a different development tool to that used for the main program. The use of DLLs also reduces the complexity of managing a multi-group programming team by allowing each group to supply an upgraded DLL independently of other groups. Where standard Windows, VCL, or BDE DLLs are used by a program, they need only be installed if they (or later versions) do not exist on the target system. The programming interface to the module synthesis DLLs is described in terms of its exported functions in section B.5 of appendix B.

There are associated disadvantages with using DLLs. The first is that other programs can install newer (or as often is the case, older) DLLs that are not totally compatible[98]. Similarly, different versions of a DLL can exist on the same computer or network. This means that the correct DLL search path may be important for the application to perform correctly—all relevant DLLs must be present and able to be located. Another problem noted recently is that DLLs built using a different compiler version to the application may be incompatible (depending on the operating system). This is due to compatibility errors in the different compiler versions. The last major problem with DLLs experienced by developers is that they do not support the separate memory spaces required by different threads—global data in the DLL is global to all threads. That is, when different (32-bit) threads of a single process load the same DLL, they are loaded into the same absolute memory space[13]. (16-bit threads are shared between processes as well as threads[69].) The unwary programmer can be caught with these fiendish artefacts of DLL usage.

7.7 Distributed Genetic Algorithm Implementation

The Genetic Algorithm was implemented in a distributed manner such that each client process or thread could work independently upon the calculation of an individual circuit's fitness. These clients could be run any number of times on any number of computers connected via a local area network.

The database choice was to use Paradox tables accessed through the Borland Database Engine (BDE). This limited the sharing policy to table-level locking, which is slower than the more preferable row-level locking. In addition, the only level of transaction isolation provided by the BDE with Paradox tables is dirty read, which

4Discussions of such errors, and possible 'work-arounds,' are usually given on the respective web-sites of the compiler developers, with URLs such as http://www.microsoft.com and http://www.borland.com.
7.8 Conclusions

means that uncommitted changes are read by subsequent accesses to the database; in effect multi-statement transactions are not supported[13]. In databases that support transactions, changes are not written to the database until a commit command is processed. Conversely, changes made within a transaction can be undone with a rollback command. The highest level of transaction-level isolation is transaction-level read consistency, which is supported by high-end databases such as Oracle and which ensures a consistent view of the database throughout a transaction—ignoring the effects of other simultaneous transactions even when they are complete and committed[43]. The cost of such a high level of isolation comes in the amount of computing resources (memory, drive-space and CPU time) required in the processing and logging of all the separate transactions and states of the database so that rollbacks can be made. The effect that the limited transaction support of Paradox tables had on the distributed GA was that a form of transaction needed to be implemented manually such that any single thread would take ownership of one or more rows by inserting the thread's unique identifier. Other threads knew not to touch these rows until the ownership identifier was removed, and the owner thread could update the rows at its leisure (independently of the table-level locking).

The table level locking, controlled by the BDE, is effected by defining the current client connection to the database that has control over each table. These definitions are contained in a globally shared file (pdousrs.net)[37] for which each client thread requires read and write privileges.

7.8 Conclusions

Many different software development paradigms were considered and analysed during the evolution of the synthesiser source code. As the synthesiser application matured and became more functional, it moved to a platform dependent, development tool dependent application relying upon an underlying database. Analysis of the different available options is important, especially as software development matures into an engineering discipline. The main development issues raised during this research were: design for portability, use of relational database engines, reliance upon development tools, and miscellaneous software development approaches.

This chapter has discussed various points currently being considered by bodies and authorities around the world when looking at formalising software engineering. As software development continues to mature, there will be many more formal approaches to software design and coding that will possibly change the way software engineers and code writers perform.
Chapter 8

Conclusions

8.1 Introduction

With the increase in FPGA density and reduction in their cost, the popularity of FPGAs has increased greatly over the last decade. FPGAs are emerging as a ubiquitous digital technology. Development tools have had to keep up with the ever-increasing size and complexity of designs being implemented within the target devices.

This thesis has presented a synthesis system that allows large designs to be implemented and managed. Optimisation was presented as an integral part of the design-and-synthesis process. A detailed description of the synthesis and optimisation methodologies was given, along with a literature review of both utilised and associated techniques.

The thesis is summarised in the following section. Following this, conclusions to the thesis are drawn. The last section provides some suggestions for the direction of future research.

8.2 Thesis Summary

Chapter 1 presented technologies relevant to this thesis. Programmable logic, pipelining techniques, genetic algorithms and stochastic numbers were briefly introduced. Field programmable logic was presented in much greater depth in chapter 2. A review of literature was made that covered FPGA design tools, hardware description languages, custom-configurable computing, high-level design synthesis, and design optimisation.

The specific-purpose synthesis of stochastic neural networks was presented in chapter 3. Various hardware neural network implementations were discussed. Stochastic neural networks were presented, which created the need to investigate the generation of random numbers in digital hardware. Specialised techniques were discussed that make efficient hardware implementation possible. In conclusion, an example network was
synthesised and its correct operation confirmed. Some synthesised AHDL listings are
given in appendix A. These show the textual descriptions of various physical modules
that compose the stochastic neural network.

The implementation of the synchronous module-based synthesiser was discussed in
detail in chapter 4. Initially the foundation was set with nomenclature definitions. The
design-space searches were anticipated with the discussion of the modifiable synthesis
variables. The synthesis algorithm was presented, with special focus on the handling of
module feedback. A discussion was then presented on the methods of analysing data
flow through the resulting pipelined circuits. To finish, a discussion of the synthesised
circuit control-and-scheduling units was provided.

Supplementary information is given in appendix B, including the synthesiser's
command-line parameters and usage. The syntax specifications of the input files are
given. This includes the files for the module database, circuit netlists, and genetic
algorithm parameter files. The interface to the module generator DLLs is also speci-
ified. Finally, synthesised example output files are provided to show the structure of
the synthesised AHDL code and report files.

The optimisation algorithm employed by the synchronous module based synthesiser
was discussed in detail in chapter 5. The genetic algorithm was described, including
problem encoding and the parameters used by the optimisation algorithm. The fitness
function was described along with fitness scaling techniques. A hardware compression
factor was derived that accounted for errors in the estimation of hardware resource
usage. Lastly, the genetic operations of selection, reproduction, mutation and crossover
were described in terms of the problem encoding used. These were specifically chosen
to match the physical problem of module-based circuit optimisation.

The software development was discussed in chapter 7, giving details of the meth-
ods used in the design and implementation of the synthesis applications. Software
engineering issues including software design, quality management, and development
paradigms were presented. The maturation of the synthesiser source code was dis-
cussed. Coding techniques that were used were discussed along with their advantages
and disadvantages. The features of the distributed genetic algorithm implementation
were also presented.

The techniques developed in the thesis were combined so as to observe the actual
performance of the synthesis and optimisation on circuits with real-world application.
These results were described and discussed in chapter 6, and showed the feasibility of
the method.

The first circuit developed as a case study was for numerically cubing a $2 \times 2$
fixed-point matrix. The design of the circuit was presented, and its design space was calculated. It was subsequently optimised, using the genetic algorithm, and synthesised. Results from the optimisation were provided numerically and graphically.

The second circuit was a neural network for analysing post-heart-attack patient cardiogram data. The circuit was discussed, with a focus on the salient features of the design. The effect of fixed-point arithmetic on the accuracy of the sigmoidal activation function was one aspect in particular that was given close scrutiny.

The optimisation algorithm performed correctly and it was shown that its solution convergence and population diversity could be controlled through the choice of optimisation parameters. Features of the resulting design spaces were analysed, such as the clustering of solutions, and other trends that were exhibited.

A formula for determining the maximum design space size for a circuit implementation was derived. This was based on the number of ways similar modules could be grouped together, and the number of ways that each group could be implemented.

8.3 Discussion and Conclusions

The overall objective of the thesis was to develop a methodology to assist digital circuit designers in the tasks of design and optimisation. This was partitioned into three smaller objectives, each of which has been achieved. These can be summarised as follows:

1. The development of a scalable, high-level, hierarchical, module-based synthesiser that supports multiple data types, and is extensible.

2. The development of a specific genetic algorithm implementation for circuit optimisation for use in conjunction with the synthesiser.

3. Analysis of the explored circuit design spaces and of the optimisation algorithm's performance.

The general purpose synthesiser produces AHDL from the high-level hierarchical description of a circuit. Several module data-types are supported, including parallel, serial, and stochastic, and conversions are handled automatically. Module resource sharing is also implemented as requested by the designer. Modules can be defined manually through various design methods by including a standard module interface syntax. Modules can also be defined as DLL-based parametric synthesiser using a predefined programming interface. These DLLs extend the synthesiser core.

The genetic algorithm implementation was designed for the specific purpose of optimising circuits, as part of the synthesis process. This required the inclusion of mecha-
nisms for handling multiple data types, module resource sharing, and different module implementation versions. The two-part (head/tail) encoding regime was developed to represent these circuit features in a simple way. Moreover, the chromosome structure preserves general circuit features while being manipulated by the various genetic operators. One of the contributions of this thesis was the development of an encoding mechanism that allows the mapping of a circuit design to a representation that is able to be manipulated by the GA. The GA's inherent parallel nature was exploited through an implementation that supports a distributed processing environment.

One problem-specific crossover operator was developed along with four problem-specific mutation operators. The circuit features altered by the four mutation operators are module group membership, module data type, module implementation version, and module resource sharing. These operators allowed new circuits to be created, thus effecting a traversal of the design space.

The optimisation is guided by various parameters that are used in a fitness function to evaluate the quality of a particular circuit solution, with respect to the desired goal. A contribution of this research was the development of an objective fitness function that can be used for the circuit quality evaluation. The fitness function's parameters include logic cell usage, memory cell usage, and circuit run time. The relative importance of these parameters is defined, allowing objective trade-offs to be made by the optimisation algorithm.

The genetic algorithm was run on two circuits to confirm that the evolution process worked. It was observed that the explored design spaces exhibited clustering of design points. The clusters generally covered a narrow cycle-count range and a wide cell-count range. These were indicative of different circuit implementations that had the same critical data-flow path but had redundancy in other parts of the design. Individual evolution runs showed trends across the design spaces of data type and resource sharing with respect to cell count and cycle count. There were, however, no concise generalisations drawn as most runs showed different trends.

The overall objective of this thesis is an amalgamation of the three preceding objectives. It is the development of a complete methodology for the design, optimisation and synthesis of module-based digital circuits. The methodology was formed as a novel combination of new and existing techniques. The novel aspects include the specific genetic operators, the genetic problem encoding, the concept of a hardware compression factor, the synthesis methodology, and the design space analyses.

An investigation into stochastic neural network synthesis was made to highlight the advantages and disadvantages of special-purpose synthesis. This was an introductory
work that preceded the main research. This investigation included a detailed analysis of the implementation of pseudo-random number generation in digital hardware. These random number generation techniques were used in the subsequent synthesis algorithms.

The advantage of special-purpose synthesis is that the resulting circuitry can be tailored to the target hardware, and the exact requirements of the problem. This is demonstrated well with the design of the 'summation and sigmoidal activation function' for the stochastic neural network.

The main disadvantage is that a specific synthesiser must be developed for each target application—in this case stochastic neural networks. A compromise can be made, by using special-purpose synthesis techniques to create modules. These modules can then be used in general-purpose synthesis. One example is the stochastic sigmoidal activation function that can be implemented in a self-contained module. Other special-purpose techniques are not so applicable to general purpose synthesis. One example is the technique of producing non-correlated binary sequences through simple delays. This approach is fundamental to the whole structure of the synthesised circuit and is not easily implemented without changes to the general-purpose synthesiser.

The methods and findings presented in this thesis are not solely limited to FPGA applications, although there is one main aspect that would require modification before targeting VLSI and other digital platforms. The hardware resource usage estimations would need to be derived for the target implementation technology, and perhaps generalised. This would include an analysis of the effects of subsequent compilation and fitting of a synthesised design into a target device, and coincident optimisations. A hardware compression factor, or a similar concept, would have to be determined dependent on the physical target. Furthermore, the synthesis of VHDL or Verilog would increase the suitability for a range of devices.

In the analysis of the pipelined data flow of the circuit being synthesised, a greedy strategy was used to determine reinitiation timing. The resulting initiation latency can sometimes be improved through various techniques, such as the appropriate insertion of delays. The improvement obtained through using better algorithms must be traded off against the increased computational time for the analysis, which increases the time taken by the genetic algorithm.

The hardware compression factor that was derived to account for effects during the AHDL compilation did not always produce exact estimations. Its accuracy is dependent on the level of redundancy in the circuit, which is itself dependent on the combinations of modules used and their parameters. The parallel neural network case-
study in chapter 6 is an example of inaccurate estimation. A more accurate model to estimate actual hardware resource usage would require a deep knowledge of the compilation process and the target structure. The former of these is often proprietary and highly confidential.

The design and implementation of software for the synthesiser was discussed in detail in the thesis body. Diverse issues were raised regarding software engineering styles, and coding approaches. Some subjective conclusions were drawn from observing the quality, development time, and speed of the various versions of the synthesiser software. The applied development approach was to reduce coding time, reduce debugging time, and produce more robust code. The penalty was a reduced run-time application speed, which was a calculated cost. This penalty is more pertinent to commercial software than to research based software.

The developed methodology has been shown capable of application to medium sized circuits. The scalability of the of the synthesiser, especially through its support of hierarchical design, means that it is also applicable to arbitrarily large circuits. The open-architecture nature of the system allows for extension to new data types, and modules implementing newer hardware algorithms. The system can be readily applied to practical circuit design, or used in further analysis of design features and optimisations.

The optimisation algorithm has been developed such that it is suitable for incorporation into an integrated CAD system. Such inclusion would allow circuits to be optimised towards desired goals in a seamless process. A designer using such as system would not even need to be aware of the structures or internal data-types used in the resulting implementation.

8.4 Proposed Future Extensions

As with any research topic, the synthesis and optimisation methodology presented in this thesis was subject to certain constraints and criteria. A number of proposed extensions to this research are outlined below. These can build upon the results already obtained to create an even more powerful, general purpose, and commercially useful design automation application.

Use of the HDL-Compiler Output by the Genetic Algorithm

The genetic algorithm could use the report files produced by different HDL compilers. These would provide the exact cell count for a compiled design given a specific FPGA
device. This approach has the advantage of eliminating the need to estimate cell counts. This, in turn, makes the derived hardware compression factor (γ) redundant. The major disadvantage of this approach is the time taken for HDL compilation (usually also including some level of optimisation). As computing power becomes ever-greater, this approach becomes a more feasible option.

Module Propagation Delays as a Circuit Analysis Parameter

As discussed in section 4.10, the genetic algorithm could utilise module propagation-delay information in its analysis of a circuit’s performance. This extension would mean that a circuit’s clock frequency would become part of the optimisation goal. The current assumption is that the clock period is great enough so as not to cause problems in any of the modules. Allowing faster clocking where permitted would alter the design space being searched and possibly provide more optimal circuit solutions. For instance, a certain fully parallel multiplier could be replaced by a two-cycle pipelined implementation. Not only is there a trade-off between compute cycles and resource usage, but the maximum allowable clock frequency becomes a variable. In this example the pipelined multiplier would probably cope with a higher clock frequency due to its lower propagation delay.

Power Consumption as a Circuit Analysis Parameter

Another important design criterion is a circuit’s power consumption. This is related to frequency, area, and the levels of dynamic reconfiguration. There are two obvious approaches for the inclusion of power consumption in the fitness function. The first is to empirically derive a metric for a particular family that relates various circuit features to the power requirement. The second is to use an accurate model of the configured target device—using tools such as those produced by Synopsys.

Addition of Floating-Point Data Representation

The addition of a floating-point representation of varying accuracy would allow a different class of problems to be implemented and optimised using the approach given herein. Currently, floating-point operations can be performed, hidden inside a module, but the inputs and outputs to the module must appear as a simple binary encoding (which may be interpreted as a fixed-point number). For example, an 8 bit representation of all ones could be interpreted as values such as 255, 127.5, 63.75 etc. Accuracy conversion is currently performed by appending or truncating bits.
The synthesiser would require modification to support a mantissa and exponent representation. The automatic accuracy conversion in the synthesiser would need to be modified to account for floating-point numbers.

Resource-Sharing of Multi-Module Subcircuits

Currently, only multiple instances of single modules can be used in resource sharing. If a circuit has a repeated multi-module subcircuit that would be beneficial to share, the present synthesiser requires the subcircuit be included as a separate module (itself a multi-module circuit). The optimisation algorithm would have a wider range of designs if it was the choice of the optimiser, and not the designer, as to which modules to group together as subcircuits.

Additional Implementation-Specific Design Parameters

It would be possible to alter the optimisation algorithm, and the synthesiser, to include extra parameters that control the direction of the goal-seeking. One proposed optimisation parameter is the allowable circuit re-initation times or minimum re-initiation time. This would make the solution circuits easier to integrate with existing circuitry that has its own timing constraints. For instance, a microprocessor may have pre-defined op-code durations for accessing the FPGA-implemented functions and other devices such as memory. This would assist with the integration of the synthesiser and optimiser into a hardware/software co-design system.

Incorporation of an Evolution ‘Wizard’

Extra features could be incorporated into the genetic algorithm. These could include an automatic analysis of boundary-condition circuits. Circuits with all parallel modules and those with all serial modules could be synthesised and analysed with full resource sharing and with resource sharing disabled. This would provide an initial guide to the designer as to what the actual target constraints on the circuit evolution should be.

In addition, the circuit could be monitored for homogeneity during the run and the optimisation parameters automatically updated. These parameters could include the number of generations, the population size, the penalties, the mutation rate, crossover rate, and other optimisation settings. The target values for logic-cell count, cycle count, and memory-cell count would remain unaffected.
Statistical Analysis of Optimisation Parameters

A wide variety of circuits could be repeatedly evolved with different optimisation parameters. Given enough trials, it may be possible to deduce 'good' optimisation parameters that suit general hardware evolution of this type or that relate to specific circuit subsets.

Statistical Design-Space Generalisations

An analysis could be performed on a large number of combinations of circuit topology and optimisation parameters. With an automated analysis of enough design optimisations, it may be possible to produce some quantitative empirical results on the designs and parameters. A statistical study would require a large base of data, which, in turn, would require some clever data-mining and feature-analysis techniques. The outcome could be that, for certain classes of problems, an analytic optimisation method may be found, such that an evolutionary approach is not needed.

Migration to VHDL as the Core Language

Using VHDL as the synthesised language would provide a broader base of supported devices for the synthesiser and optimiser. The main language-specific aspects of the synthesiser are the synthesis of auxiliary modules and the module interconnectivity of the main circuit. All of the synthesis routines are currently implemented in cleanly defined sections of the application's source code. In addition to the changes within the main synthesiser, the module generators would need to be redefined specifically for VHDL module synthesis. The synthesis of VHDL would be a major step towards providing support for other brands of FPGA, such as Xilinx and Atmel. Estimating logic cell usage for different (general) target devices remains a difficulty.

8.4.1 Suitable Types of Target Application

The applications that are most appropriate for the approach presented in this thesis are those that are inherently data-path oriented. In addition, designs made up of repeated processing modules are particularly suitable.

Two areas that suit these criteria well are signal processing and neural processing. Applicable signal processing circuits include image processing and digital filtering (such as Finite Input Response (FIR) and Kalman filters), and matrix/vector manipulation. The highly repetitive structure in most neural network architectures—and their inherent datapath orientation—makes them ideal application candidates.
Large Circuits

The synthesis and optimisation techniques presented in this thesis are applicable to circuits much larger than the examples presented. As the circuit size increases, however, the fitness function becomes more complex to calculate. This is largely dependent on the pipeline analysis stage.

One option to reduce processing time is to partition the circuit by first optimising regions of the circuit and then including each of these optimised blocks as a module in the overall circuit. The drawback of this procedure is that it eliminates some of the solutions that may otherwise be encountered during the design-space traversal. The number and makeup of the modules retained in the top-level design should be determined from two main considerations. The first is the reusability of the modules, which directly affects the levels of resource sharing that can be appraised during the design space search. Secondly, too great a reduction of the number of top-level modules will result in a highly limited search space. This would negatively affect the optimisation result quality.

8.4.2 Application to Dynamically Reconfigurable Circuits

The simplest way to apply the developed method to dynamic reconfiguration is to manually partition the circuit into blocks. Each of these can be optimised to fit within the desired hardware resource. If partial reconfiguration is required, the area to be occupied by each block (its optimisation target) is less than the size of the device.

Ideally, the choice of which parts of a design are instantiated at any one time would be an automated decision. The inclusion of this feature would require a change to the optimisation algorithm. Specifically, a chromosome representation would need to be maintained that included time-dependent instantiation information. The pipeline analysis stage would process the resulting circuit as if it had additional delays introduced, as data waits until specific processing modules are available on the device.

Online-Modifiable Artificial Neural Networks

For ANN circuits to have wide application, they need to be online-modifiable. That is, their synaptic weights, and even their neuron interconnections, need the ability to be retrained after the initial implementation is made.

With simple ANN structures, the interconnections can be considered static, and only the weights need adjusting. (Assigning a “zero” weight can effect a connection removal.) The training circuitry, such as that to implement backpropagation, can be instantiated on the device along with the ANN. The optimisation algorithm can attempt
the fitting of both of these within the allowable hardware resource. Operating-speed penalty calculations can be used to determine a desirable partitioning of the resource between the training circuitry and the neural network itself.

For more amorphous ANN structures, the simplest application of the synthesis methodology would be to rerun the optimisation algorithm and reconfigure the device. The natural extension of this is dynamic reconfiguration, such that only a subset of the neurons and synapses are instantiated at any one time. With the pipeline analysis algorithm modified as discussed above, the partitioning could be performed automatically to optimise operating speed given a specific hardware resource.

Alternatively, a modular approach could be taken in which the inputs and outputs of all instantiated neurons are connected via a data bus to some form of memory storage. The synaptic weights would be updated to produce a "new" neuron, and the appropriate assignment of data to the inputs effects the interconnections. The required control system would use up resource, and updating the weights would take up clock cycles. However, this scalable structure would allow many neurons to operate in parallel (allowing the serialised implementation of large ANNs), and the optimisation and synthesis needs to be performed only once.
Appendix A

Stochastic Neural Network Synthesiser Reference

A.1 Synthesised AHDL Output

This section gives sample synthesised AHDL from the stochastic neural network synthesiser. The first two listings describe the circuitry to implement the linear-feedback shift register and generate the weighted stochastic bit streams. The remaining listings present the hierarchical components of the stochastic neural network in order of increasing design detail: an entire neural network, a neuron, a neuron synapse and a neuron bias.

Stochastic Bit-Stream Generator and Address-Line Modulator

This description generates three maximal length pseudo-random binary sequences from a linear-feedback shift register. Then, using bit-stream modulators, the address line bit-streams are weighted (probability of a bit being a ‘1’) to 2/3, 4/5, and 16/17 for use in the distributed generation of stochastic synaptic weights. Table A.1 shows the synthesised AHDL implementation of a stochastic bit-stream generator.

Stochastic Bit-Stream Modulator

These bit-stream modulators are cascaded together to modify the probability of a ‘1’ occurring in a stochastic bit stream. Table A.2 gives the synthesised AHDL bit-stream modulator.
TITLE "Weighted random bit-stream address generator.";
CONSTANT seed = B"001000110101100011011011";
CONSTANT weightA = B"11010101"; % 2/3 (Reverse order) %
CONSTANT weightB = B"10110011"; % 4/5 %
CONSTANT weightC = B"10001111"; % 16/17 %

FUNCTION addrmod(clock, reset, prevstg, modbit, carrier)
  RETURNS (nextstg);
SUBDESIGN addrgen(
  clock : INPUT;
  reset : INPUT;
  randA : OUTPUT;
  randB : OUTPUT;
  randC : OUTPUT;
)

VARIABLE
  lfsr[24..0] : DFF;
  addrA[7..0] : addrmod;
  carrA[7..0] : DFF;
  addrB[7..0] : addrmod;
  carrB[7..0] : DFF;
  addrC[7..0] : addrmod;
  carrC[7..0] : DFF;

BEGIN
  lfsr[].clk = clock;
  carrA[].clk = clock; carrA[].clrn = !reset; addrA[].clock = clock;
  addrA[].reset = reset; addrA[].modbit = weightA; carrA[7..1].d = carrA[6..0].q;
  addrA[].prevstg = (B"0", addrA[7..1].nextstg);
  addrA[].carrier = carrA[].q;
  carrB[].clk = clock; carrB[].clrn = !reset; addrB[].clock = clock;
  addrB[].reset = reset; addrB[].modbit = weightB; carrB[7..1].d = carrB[6..0].q;
  addrB[].prevstg = (B"0", addrB[7..1].nextstg);
  addrB[].carrier = carrB[].q;
  carrC[].clk = clock; carrC[].clrn = !reset; addrC[].clock = clock;
  addrC[].reset = reset; addrC[].modbit = weightC; carrC[7..1].d = carrC[6..0].q;
  addrC[].prevstg = (B"0", addrC[7..1].nextstg);
  addrC[].carrier = carrC[].q;
  IF (reset) THEN
    lfsr[].d = seed;
  ELSE
    lfsr[24..1].d = lfsr[23..0].q;
    lfsr0.d = (lfsr24.q XOR lfsr21.q);
  END IF;
  carrA0 = (lfsr3.q XOR lfsr11.q XOR lfsr15.q);
  carrB0 = (lfsr6.q XOR lfsr22.q XOR lfsr23.q);
  carrC0 = (lfsr4.q XOR lfsr7.q XOR lfsr23.q);
  randA = addrA0.nextstg; randB = addrB0.nextstg; randC = addrC0.nextstg;
END;

Table A.1: Synthesised random number generator that produces the address streams used by the neuron synapses. The random bit streams are produced with different stochastic values (probabilities of each bit being a '1').
TITLE "Stochastic bit-stream modulator."

SIGNAL addrmod
(
  clock : INPUT;
  reset : INPUT;
  prevstg : INPUT;
  modbit : INPUT;
  carrier : INPUT;
  nextstg : OUTPUT;
)

VARIABLE
  bit : DFF;
  temp[2..0] : NODE;
BEGIN
  bit.clk = clock;
  bit.clrn = !reset;
  nextstg = bit.q;
  bit.d = !(temp0 AND temp1 AND temp2);
  temp0 = (modbit NAND prevstg);
  temp1 = (modbit NAND carrier);
  temp2 = (prevstg NAND carrier);
END;

Table A.2: Synthesised random bit-stream modulator that can be cascaded with others to modify the value of a stochastic bit stream.
Stochastic Neural Network Structure

The neural network design file is the topmost in the AHDL project. It utilises the address-line generation module and the neurons, which in turn use the synapses and biases. Table A.3 is an example synthesised AHDL stochastic neural network.

Stochastic Neuron (Summation and Activation)

The neuron utilises the bias and synapse modules. It also performs input summation and instantiates the sigmoidal activation function. The method used for the summation and activation function in this example is by formula, allowing the Altera compiler to implement it and partition it in what it deems as the most efficient manner. Table A.4 is an example listing of a synthesised AHDL stochastic neuron.

Neuron Synapse (Stochastic Weight Storage and Multiplication)

This synapse description stores the synaptic weight in a table which is referenced from the weighted address lines. The stochastic stream that is generated from this look-up table is multiplied with the inbound bit stream using bipolar stochastic multiplication (XNOR). Table A.5 gives an example synthesised AHDL stochastic synapse.

Neuron Bias (Stochastic Bias Storage and Multiplication)

The biases are simply synapses where there is no input stochastic bit stream, only the weighted address lines. The output is the weighted stochastic bit stream generated from the look-up table. Table A.6 gives an example synthesised AHDL stochastic neuron bias.
TITLE "Stochastic Neural Network with 5 Neurons and 6 Synapses";
% 2 input neurons %
% 2 hidden neurons %
% 1 output neurons %

FUNCTION neuron02(in[1..0], addrA[2..0], addrB[2..0], addrC[2..0])
  RETURNS (out);
FUNCTION neuron03(in[1..0], addrA[2..0], addrB[2..0], addrC[2..0])
  RETURNS (out);
FUNCTION neuron04(in[1..0], addrA[2..0], addrB[2..0], addrC[2..0])
  RETURNS (out);
FUNCTION addrgen(clock, reset)
  RETURNS (randA, randB, randC);

SUBDESIGN network
(
in[1..0] : INPUT;
clock : INPUT;
reset : INPUT;
out : OUTPUT;
)

VARIABLE
  nd[4..0] : NODE;
  randA : NODE;
  addrA[8..0] : DFF;
  randB : NODE;
  addrB[8..0] : DFF;
  randC : NODE;
  addrC[8..0] : DFF;

BEGIN
  addrA[0].clk = clock;
  addrA[0].clrn = !reset;
  addrA[8..1].d = addrA[7..0].q;
  addrA0.d = randA;
  addrB[0].clk = clock;
  addrB[0].clrn = !reset;
  addrB[8..1].d = addrB[7..0].q;
  addrB0.d = randB;
  addrC[0].clk = clock;
  addrC[0].clrn = !reset;
  addrC[8..1].d = addrC[7..0].q;
  addrC0.d = randC;
  (randA, randB, randC) = addrgen(clock, reset);
  nd0 = in0;
  nd1 = in1;
  nd2 = neuron02(nd1, nd0, addrA[8..6].q, addrB[8..6].q, addrC[8..6].q);
  nd3 = neuron03(nd1, nd0, addrA[5..3].q, addrB[5..3].q, addrC[5..3].q);
  nd4 = neuron04(nd3, nd2, addrA[2..0].q, addrB[2..0].q, addrC[2..0].q);
  out = nd4;
END;

Table A.3: Synthesised stochastic neural network that contains three hidden neurons and six synapses.
TITLE "Synthesised AHDL hidden neuron #3 with 2 inputs";

% Neuron activation is by formula %

CONSTANT ph = B"0";

FUNCTION bias03(addr[2..0])
    RETURNS(out);
FUNCTION syn0300(instream, addr[2..0])
    RETURNS(out);
FUNCTION syn0301(instream, addr[2..0])
    RETURNS(out);

SUBDESIGN neuron03
(  
in[1..0] : INPUT;
addrA[2..0] : INPUT;
addrB[2..0] : INPUT;
addrC[2..0] : INPUT;
out : OUTPUT;
)

VARIABLE
  prA[2..0] : NODE;
  sum[1..0] : NODE;

BEGIN
% Accurate formula code for activation function %
  sum[] = (ph,prA0)+(ph,prA1)+(ph,prA2);
  IF (sum[]>=2) THEN
      out = VOC;
  ELSE
      out = GND;
  END IF;
  prA0 = syn0300(in0, addrC0, addrB0, addrA0);
  prA1 = syn0301(in1, addrC1, addrB1, addrA1);
  prA2 = bias03(addrC2, addrB2, addrA2);
END;

Table A.4: Synthesised hidden stochastic neuron. This version uses the formula for calculating the threshold as opposed to look-up tables.
TITLE "Synapse #1 for neuron #2 with weight 0.990";

SUBDESIGN syn0201
(
   instream : INPUT;
   addr[2..0] : INPUT;
   out : OUTPUT;
)

VARIABLE
   weight : NODE;

BEGIN
   TABLE
      addr[] => weight; % Stored Weight %
      0 => 0;
      1 => 1;
      2 => 1;
      3 => 1;
      4 => 1;
      5 => 1;
      6 => 1;
      7 => 1;
   END TABLE;
   out = (weight XNOR instream); % Bipolar multiplication %
END;

Table A.5: Synthesised stochastic synapse. The synaptic weighting is stored as a bipolar representation of a number between +1 and -1.
TITLE "Neuron bias for neuron #4 with value -0.990"

SUBDESIGN bias04
(  
addr[2..0] : INPUT;
out : OUTPUT;
)

BEGIN
  TABLE
    addr[] => out;
    0 => 1;
    1 => 0;
    2 => 0;
    3 => 0;
    4 => 0;
    5 => 0;
    6 => 0;
    7 => 0;
  END TABLE;
END;

Table A.6: Synthesised stochastic neuron bias. The bias weighting is stored as a bipolar representation of a number between +1 and -1.
Appendix B

Synchronous-Module Based HDL Synthesiser Reference

B.1 Introduction

This appendix provides a technical reference for the synthesiser described in chapter 4. The operation of the command-line user interface is detailed. The formats of the synthesiser input and output files are described. The programming interface to the module generators is given so that further modules can be developed and added to the existing base. Lastly, use of the genetic algorithm client processors is described.

B.2 Command Line Interface

The synthesiser, by default, expects manual starting of the repair algorithm or the genetic algorithm and subsequent synthesis. To run the synthesiser, the command syntax is:

```
synthbase [-o[cdgrms]] [-Nfile] [-Gfile]
```

Options for -o are:

c Close program on completion.
d Allow distributed genetic algorithm client processors.
g Automatically start genetic algorithm.
m Automatically load predefined AHDL modules.
r Automatically initiate repair (ignored if g also used).
s Automatically start synthesiser.
The file given for the -N option is the netlist file to process. If this option is not
given then the previously loaded netlist file is used. If given, the argument should
include the filename (with suffix).

The file given for the -G option is the genetic algorithm parameter file to use. If
this option is not given then the previously loaded genetic algorithm parameter file is
used. If given, the argument should include the filename (with suffix).

An example command that uses `genalg.ga` GA parameters to optimise the circuit
described by `circuit.net`, utilising GA clients and automatically synthesising the
result, is:

```
synthbase -ogcdms -Ne:\data\circuit.net -Ge:\data\genalg.ga
```

### B.3 Input Files

The synthesiser uses three types of textual input files; two for synthesis and one for
optimisation. These are in addition to the module generators dealt with in sections 4.6
and B.5. For synthesis, there are text design files that describe modules—effectively
producing a database of usable modules. Also, there is the netlist that describes the cir-
cuit to be synthesised or optimised. The third input file describes the genetic algorithm
parameters, which are themselves described in section 5.4.2.

The BNF (Backus-Naur Form) syntax designation used for recurrence is:

- `{X}` indicates X can be repeated any number of (including zero) times.
- `{Y}:n` indicates Y must be repeated at least n times with no maximum.
- `{Z}:n:m` indicates Z must be repeated at least n times and at most m times.

### B.3.1 Module Database and Interface Shell

Each module that is designed by hand can consist of any number of files. At its highest
level, the module must consist of a text design file that contains a `module` interface
shell section within an AHDLS comment. The BNF syntax of the interface is given in
table B.1, with an overview provided in section 4.2.

### B.3.2 Circuit Netlist Description

The circuit netlist uses interconnected modules to describe a circuit. There are three
main parts to the netlist: overall circuit parameters, modules used, and module inter-
connections. The BNF syntax specification of the netlist file is given in table B.2. An
example netlist file is given in table B.3.
<interfacesshell> ::= %{{"\n":1<line>}}{"\n":1%
<line> ::= <extline>|<inline>|<boolline>|<vectorline>|<controlline>
    |<extcontrolline>|<datatypeline>
<extline> ::= ["<textparam>"]="<string>";
<inline> ::= ["<intparam>"]="<integer>";
<boolline> ::= ["<boolparam>"]="<bool>";
<vectorline> ::= ["<vectorparam>"]="<vector>";
<controlline> ::= ["<controlparam><integer>"]="<control>";
<extcontrolline> ::= ["<extcontrolparam><integer>"]="<extcontrol>";
<datatypeline> ::= ["<datatypemenu>"]="<datatype>";
<bool> ::= y|Y|n|N|true|false
<vector> ::= {<vectorchar>}
<control> ::= <integer>,<inrange>
<extcontrol> ::= <inrange>,<integer>
<vectorchar> ::= x|.
<textparam> ::= function|description|ext_desc|integer
<intparam> ::= 10k_cells|cycles|inputs|outputs|parameters
    |inaccuracy|inbuswidth|outaccuracy|outbuswidth|external_lines
    |external_pulses|control_lines|control_pulses
<boolparam> ::= clocked|clm|converter|can_share
<vectorparam> ::= collision_vector
<controlparam> ::= control
<extcontrolparam> ::= ext_control
<datatypemenu> ::= indatatype|outdatatype
<intrange> ::= <integer>|<integer><integer>
<datatype> ::= par|ser|stoch
<string> ::= one or more characters
<integer> ::= {<digit>}:1
<digit> ::= 0|1|2|3|4|5|6|7|8|9

Table B.1: BNF syntax of the module interface shell. Some characters are given in double quotes to improve readability.
\begin{verbatim}
<netlist> ::= \{"\n":1|<line>\}
<line> ::= <textline>|<intline>|<datatypeline>|<cominline>
<textline> ::= [\"<textparam\\"]=\"<string\\");
<intline> ::= [\"<intparam\\"]=\"<integer\\");
<textparam> ::= description|function|module{<integer>}:1|desc{<integer>}:1
               |param{<integer>}:1:index| associated_files
<intparam> ::= accuracy{<integer>}:0:1
<datatypeline> ::= [\"<datatype\\"]{<integer>}:0:1\"=\"<datatype\\");
<cominline> ::= [\"<comn\\"]{<module>\"<module\"<index\">\"<integer\">\"F\">0:1;
<datatype> ::= par|ser|stoch
<module> ::= {<integer>}:2:2
<string> ::= one or more characters
<integer> ::= {<digit>}:1
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<index> ::= a|b|c|...|y|z
\end{verbatim}

Table B.2: BNF syntax of the circuit netlist file. Some characters are given in double quotes to improve readability.

### B.3.3 Genetic Algorithm Parameters

The genetic algorithm parameter file describes aspects of the GA optimisation process as discussed in section 5.4.2. The BNF syntax specification of the GA parameter file is given in table B.4. An example genetic algorithm parameter file is given in table B.5.

### B.4 Output Files

There are a number of different types of text files produced by the synthesiser during the synthesis and optimisation processes. Table B.6 gives an example directory listing of the output files produced by the synthesiser, and their descriptions. The subsequent subsections detail each of the file types.

### B.4.1 Synthesised Modules

During the synthesis process, AHDL files that represent the circuit component modules are put into the output directory. These are either synthesised by the synthesiser or module generators, or they are copied from the library of predefined modules. In addition to text design files (TDF), there can be many other file types as discussed in chapter 4. An example synthesised module is shown in table B.7. Note that there is no module interface shell, as the module generator automatically provides all the parameters directly to the synthesiser's database.
[description]=Generic test circuit;
[function]=func2;
[datatype]=par;
[accuracy]=5;

[module0]=input;
[desc0]=Input Module;
[accuracy0]=5;
[datatype0]=par;
[param0a]=3;

[module1]=add;
[desc1]=LLL;
[datatype1]=ser;
[accuracy1]=5;

[module2]=output;
[desc2]=Output Module;
[accuracy2]=5;
[param2a]=2;

...

[module5]=addconst;
[desc5]=Constant adder 2;
[datatype5]=ser;
[accuracy5]=8;
[param5a]=17;

[module6]=addkext;
[desc6]=Adder #3 (external);
[datatype6]=par;
[accuracy6]=5;

[module7]=addconst;
[desc7]=Constant adder 4;
[datatype7]=ser;
[accuracy7]=7;
[param7a]=21;

[comm0]=00a-01a;
[comm1]=00b-01b;
[comm2]=00c-04a;
[comm3]=01a-06a;
[comm4]=01a-03a;
[comm5]=04a-03bf;
[comm6]=05a-02a;
[comm7]=03a-06a;
[comm8]=06a-07a;
[comm9]=07a-02b;

Table B.3: Example circuit netlist file.
<gapparameters> ::= {"\n":1\n}</gapparameters>
<line> ::= <boolline>|<intline>|<floatline>|<typesline>
<boolline> ::= "[exhaustive]="<bool>; "true"|"false"
<intline> ::= "["<intparam>"YNAMIinteger>; "false"
<floatline> ::= "["<floatparam>"YNAMIfloat>;
<typesline> ::= "["<types>"YNAMItrue[
<datatypes> ::= {<datatype>,}0<datatype>
<bool> ::= yY|nN|true|false
<floatparam> ::= p_mutation|p_crossover|optimise_time|optimise_size
 |size_hard_penalty|mem_hard_penalty|time_hard_penalty
<float> ::= floating point number in fixed or scientific notation
<intparam> ::= generations|population|output_gens|size_hard_limit
 |mem_hard_limit|time_hard_limit|iterations
<datatype> ::= par|ser|stoch
<integer> ::= {<digit>}; 1
<digit> ::= 011213141516171819

Table B.4: BNF syntax of the genetic algorithm parameter file. Some characters are
given in double quotes to improve readability.

[generations]=5;
[population]=4;
[p_mutation]=2.0000E-2;
[p_crossover]=.031;
[output_gens]=2;
[exhaustive]=n;
[optimise_time]=0.7;
[optimise_size]=0.6;
[size_hard_penalty]=3.2;
[size_hard_limit]=1000;
[mem_hard_penalty]=13.2;
[mem_hard_limit]=256;
[time_hard_penalty]=2.3;
[time_hard_limit]=32;
[iterations]=5;
[types]=par,ser;

Table B.5: Example genetic algorithm parameter file.
<table>
<thead>
<tr>
<th>Filename</th>
<th>File Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixels.tdf</td>
<td>Main circuit design</td>
</tr>
<tr>
<td>control.tdf</td>
<td>Control signal generator design</td>
</tr>
<tr>
<td>_add12p.tdf</td>
<td>Synthesised (functional) module</td>
</tr>
<tr>
<td>_divk12p1.tdf</td>
<td>Synthesised (functional) module</td>
</tr>
<tr>
<td>_comm10p1.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_pipelay10p1.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_uacc12p8.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_pipelay12p1.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_pipelay12p2.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_pipelay12p3.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_pipelay12p4.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_cd12p2.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_uacc10p12.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_uacc12p10.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>_uacc12p12.tdf</td>
<td>Synthesised (auxiliary) module</td>
</tr>
<tr>
<td>Pixels.rep</td>
<td>Design report file</td>
</tr>
<tr>
<td>ga.txt</td>
<td>Optimisation results file</td>
</tr>
</tbody>
</table>

Table B.6: Example of the output directory after successful optimisation and synthesis.

### B.4.2 Main Circuit Design Listing

The synthesiser produces a project of files for compilation. At the top of the project is the main circuit (although this can be included in the controller design if an integrated controller is desired). The file consists of a number of distinct sections. Firstly, there is a list of the prototypes of modules that are included in the project. Next is the module interface shell, so that the design can be used as a module in higher-level designs. The circuit inputs and outputs follow this, before the variable section. The variable section defines instantiations of lower-level modules. The main circuit structure is then defined by global signal connections, control line connections, and lastly, the module interconnections. Table B.8 shows an example listing of the main synthesised design file. (Each ellipsis indicates where code was removed to fit on the page).

### B.4.3 Control Signal Generator Design Listing

The control signal generator, as its name suggests, generates the control signals required by the synthesised circuit. It is the highest level in the project design hierarchy, although it does not need to be included if the main circuit is going to be used as a module in a further design. An example of the synthesised controller is given in table B.9. It includes the main circuit as a function, which it makes one instance of in a variable. The main circuit's input and output names are preserved (except for CONTROL[]), which is no longer required). A state machine is used to implement the
TITLE "Synthesised multiply (lower half only) module";

SUBDESIGN _mult32s
(
  clk : INPUT;
  clrn : INPUT;
  InA : INPUT;
  InB : INPUT;
  Control[1..0] : INPUT;
  OutA : OUTPUT;
)

VARIABLE
  RegA[31..0] : DFF;
  RegB[31..0] : DFF;
  RegOut[31..0] : DFF;

BEGIN
  RegA[31..0].(clk,clrn) = (GLOBAL(clk),GLOBAL(clrn));
  RegB[31..0].(clk,clrn) = (GLOBAL(clk),GLOBAL(clrn));
  RegOut[31..0].(clk,clrn) = (GLOBAL(clk),GLOBAL(clrn));

  IF(Control0) THEN
    RegA[31..0].d = (InA,RegA[31..1].q);
  ELSE
    RegA[31..0].d = (RegA[30..0].q,0);
  END IF;

  RegB[31..0].d = ((Control0 AND InB), RegB[31..1].q);

  OutA = RegOut[0].q;

  IF (Control0) THEN
    RegOut[31..0].d = '0;
  ELSIF (Control1) THEN
    RegOut[31..0].d = (0,RegOut[31..1].q);
  ELSIF (RegB[0].q) THEN
    RegOut[31..0].d = RegOut[31..0].q+RegA[31..0].q;
  ELSE
    RegOut[31..0].d = RegOut[31..0].q;
  END IF;
END;

Table B.7: AHDL multiplier module synthesised by a module generator DLL. Note the layout is human-friendly to assist in the debugging process. A module interface shell is not required, but the I/O labels must conform to the standard.
TITLE "None";
% Function prototypes %
FUNCTION _cvnv2p32(clk,clrn,InA)
    RETURNS(OutA[31..0]);
FUNCTION _cvnv2s32(clk,clrn,InA[31..0],Control)
    RETURNS(OutA);
...
% Module parameter information
[function]=multipliers;
[description]=None;
[10k_cells]=1148;
[10k_mem_cells]=0;
...[control14]=14,0;
[control15]=15,0;
%
SUBDESIGN multipliers
(  clk : INPUT;
  clrn : INPUT;
  InA[31..0] : INPUT;
  ...
  Control[15..0] : INPUT;
  OutA[31..0] : OUTPUT;
  ...
)
% Modules used in circuit %
VARIABLE
  ModuleC : _mult32s; % LAB=191 EAB=0 %
  ...
  ModuleR : _cvnv2p32; % LAB=32 EAB=0 %
  % Totals  LAB=1148 EAB=0 %
% Circuit implementation %
BEGIN
% Global clock and clear connections %
ModuleC.(clk,clrn) = (GLOBAL(clk),GLOBAL(clrn));
...% Module control line connections %
ModuleC.Control[1..0] = Control[1..0];
ModuleD.Control[1..0] = Control[3..2];
...% Module interconnections %
OutA[31..0] = ModuleR.OutA[31..0];
OutB[31..0] = ModuleP.OutA[31..0];
... ModuleR.InA = ModuleF.OutA;
END;

Table B.8: Main AHDL circuit synthesised by the synthesiser. This TDF file is at
the top of the project hierarchy (with the exception of the control circuit if one exists).
controller function, and it can handle multiple circuit initiations.

B.4.4 Design Report File

At the conclusion of every successful synthesis, a report file is generated that describes the salient features of the synthesised circuit. It indicates the control signals required by the circuit. These are both the control signals produced by the controller circuit as well as the external control signals to be provided to the circuit. The latter are represented hierarchically and this is the only place where they can be obtained.

In addition to the control signals, the pipelined nature of the circuit is described. This includes the initiation sequence—when data should be provided to the circuit, and the (possibly variable) delays between initiations. The collision vector of the circuit is also given.

Table B.10 shows an example generated report file.

B.4.5 Optimisation Results File

The optimisation results file logs the fitnesses and structures of individuals at given generations throughout the optimisation process. This is useful for the analysis of the design space and the optimisation process. The data about each individual is written to the results file at the beginning of the optimisation process, after the last generation, and every $n$ generations—where $n$ is given by the output.gens parameter in the GA configuration file.

An example optimisation results file is given in table B.11. The HEADER line describes each functional module comprising the circuit in the form:

Module_Number(Function, Minimum_Accuracy);

Following this, there is a line for each individual, for each generation being output (broken into three lines in table B.11 to fit on the page). Each line describes the chromosome representing each circuit, and consists of the following fields concatenated together (See section 5.4.1):

Generation: Completed generation number, starting at zero.

Individual: Individual (circuit) number.

Groups: Describes the group that each module in order belongs to (separated by commas and terminated with a colon).

Group_Number(Data_type, Version, Sharing): Describes each group in terms of its data type, implementation version, and whether it is a shared resource.
% Function prototypes %
FUNCTION multipliers(clk,clrn,InA[31..0],InB[31..0],InC[31..0],InD[31..0],
   InE[31..0],InF[31..0],InG[31..0],InH[31..0],Control[15..0])
RETURNS(OutA[31..0],OutB[31..0],OutC[31..0],OutD[31..0]);

... Module interface shell and Subdesign sections go here

% Modules and variables used in circuit %
VARIABLE
   ModuleC : multipliers;
   Counter[6..0] : DFF;
   OutNode[16..0] : NODE;
% Circuit implementation %
BEGIN

% Global clock and clear connections %
ModuleC(clk,clrn) = (GLOBAL(clk),GLOBAL(clrn));
Done = OutNode0;
ModuleC.Control[15..0] = OutNode[16..1];

Counter[6..0](clk,clrn) = (GLOBAL(clk), GLOBAL(clrn));

IF (Start) THEN
   Counter[6..0].d = 1;
ELSIF (Counter[6..0].q=96) THEN
   Counter[6..0].d = 1;
ELSIF (Counter[6..0].q>1) THEN
   Counter[6..0].d = Counter[6..0].q + 1;
ELSE
   Counter[6..0].d = GND;
END IF;

TABLE
   Counter[6..0].q,Start =>OutNode[16..0];

   0, GND => B"000000000000000000";
   B"xxxxxxx",VCC => B"1111111010101010";
   1, GND => B"00000000010101010";
   2, GND => B"00000000010101010";
   ... 
   95, GND => B"000000000101010100";
   96, GND => B"1111111101010111";
END TABLE;

% Module interconnections %
OutA[31..0] = ModuleC.OutA[31..0];
OutB[31..0] = ModuleC.OutB[31..0];
OutC[31..0] = ModuleC.OutC[31..0];
...
ModuleC.InG[31..0] = InG[31..0];
ModuleC.InH[31..0] = InH[31..0];
END;

Table B.9: Controller AHDL circuit synthesised by the synthesiser. The module interface shell and the subdesign section are not shown due to space limitations.
Report generated for multipliers.tdf
Description: None

OVERVIEW
  Estimated number of 10k LAB cells: 1148
  Estimated number of 10k EAB cells: 0
  Cycles for 1 iteration (compute time): 96
  Associated file dependencies:

EXTERNAL CONTROLS AND INPUTS
  There are no external controls.

CONTROL SIGNALS GENERATED BY CONTROLLER
  Control lines are: Control[15..0]

  11111 // 111111111111
    // 888888888999999900000 // 8888888889

  Cycle->  012345 // 123456789012345678901234 // 012345678901

  Control10 = 111111 // ...................111111 // ...........

  Control11 = ........ // 1111111111111111111111 // 11111111111111111111111

  Control12 = 111111 // ...................111111 // ...........

  Control13 = ........ // 1111111111111111111111 // 111111111111111111111111

  Control14 = 111111 // ...................111111 // ...........

  Control15 = ........ // 1111111111111111111111 // 1111111111111111111111111

  Control16 = 111111 // ...................111111 // ...........

  Control17 = ........ // 1111111111111111111111 // 1111111111111111111111111

  Control18 = 1..... // ...................1..... // ...........

  Control19 = 1..... // ...................1..... // ...........

  Control20 = 1..... // ...................1..... // ...........

  Control21 = 1..... // ...................1..... // ...........

  Control22 = 1..... // ...................1..... // ...........

  Control23 = 1..... // ...................1..... // ...........

  Control24 = 1..... // ...................1..... // ...........

  Control25 = 1..... // ...................1..... // ...........

  Done signal= ........ // ...................1..... // ...........

  Repeat-> // 

  *********** // ***********

PIPELINING AND LATENCY INFORMATION

  Collision vector = xxxxxxxxxxxxxxxxxx // xxxxx
  Average reinitialization = 96.00 cycles (for 5 iterations)
  Circuit compute time = 96 cycles
  Initiation sequence has 1 steps (# = repeating cycle)
  # Initiation delay 1 is for 96 cycles

End of report.

Table B.10: Synthesis report file describing the main features of the synthesised circuit. It describes the functionality of the controller and the pipelining attributes of the synthesised circuit. The characters // indicate where part of a line has been removed to fit on this printed page.
*Cells10k* The number of logic cells required.

*MemCells10k* The number of memory cells required.

*Cycles* The number of cycles to complete the given number of iterations.

*Fitness* The numerical fitness of the individual.

*Rating* Indicates whether the individual's fitness is top-equal (BEST) or bottom-equal (WORST) for the current generation.

## B.5 DLL Synthesiser Extensions (Module Generators)

The programming interface to the DLL extensions uses non-mangled names. Function name encoding (name mangling) is used in C++ to implement the overloading of functions[22]. For example, in respect to class `x` the function `x::f(int)` gets mangled to `f_1xFi`, while `f(x,int)` becomes `f_F1xi`. By using the `_stdcall` modifier, the name is not mangled. This makes the DLL usage and implementation easier across the boundaries of different programming languages.

The three exported functions that provide the programming interface to each DLL are described below:

```c
extern "C" char* _export _stdcall Operations(void); This function returns a string that lists the names of the functions implemented in the DLL. The names are separated by a carriage-return character (`\n`). An example, for a DLL that implements addition and subtraction might be: "add\nsubtract"

extern "C" char* _export _stdcall AvailableTypes(char* Operation); This function describes the ability of the DLL to synthesise different versions of each of the modules it is capable of. The argument to the function is a string that indicates which function information is required for. This must be one of the functions returned by the Operations() function. The return value defines the data-types and bit accuracies that the module can be generated with. The string consists of a definition for each available type, separated by carriage return characters. Each type definition consists of the data type (`par`, `ser`, or `stoch`) followed by a colon and then the accuracies available for that type. The accuracies are numbers, or number ranges (using a minus to indicate a range), separated by commas. As an example, the string "par:8,16-32\nser:2-999" indicates that the module in question can be generated for parallel 8 bit and 16 to 32 bits. In addition it can be generated in serial from 2 to 999 bits.
```
HEADER: 1(add, 12); 2(add, 12); 3(add, 12); 5(add, 12);
4(connect, 10); 6(divconst, 12); 0(input, 8); 7(output, 8);
0:0:0, 1, 0, 4, 5, 6, 7: 0(par, 0, Y); 1(par, 0, N); 2(ser, 0, N); 3(ser, 0, Y);
4(par, 0, Y); 5(par, 0, Y); 6(par, 0, N); 7(par, 0, N);
*183.339969* 0.000000* 1.000000** 123.600006*BEST
0:1:0, 1, 0, 4, 5, 6, 7: 0(par, 0, Y); 1(ser, 0, Y); 2(ser, 0, N); 3(ser, 0, N);
4(ser, 0, Y); 5(par, 0, N); 6(par, 0, N); 7(par, 0, N);
*606.089966* 0.000000* 14.000000** 416.300018*
0:2:2, 1, 3, 2, 4, 5, 6, 7: 0(ser, 0, N); 1(ser, 0, Y); 2(ser, 0, Y); 3(par, 0, N);
4(par, 0, Y); 5(par, 0, N); 6(par, 0, N); 7(par, 0, N);
*525.099976* 0.000000* 24.000000** 368.700012*
0:3:2, 0, 0, 2, 4, 5, 6, 7: 0(ser, 0, Y); 1(par, 0, N); 2(par, 0, N); 3(par, 0, N);
4(par, 0, N); 5(par, 0, Y); 6(par, 0, N); 7(par, 0, N);
*773.409973* 0.000000* 24.000000** 536.100037*WORST
1:0:0, 1, 0, 4, 5, 6, 7: 0(par, 0, Y); 1(par, 0, N); 2(ser, 0, N); 3(ser, 0, Y);
4(par, 0, Y); 5(par, 0, Y); 6(par, 0, N); 7(par, 0, N);
*183.339966* 0.000000* 123.600006*BEST
1:1:0, 1, 0, 4, 5, 6, 7: 0(par, 0, Y); 1(ser, 0, N); 2(ser, 0, Y); 3(ser, 0, N);
4(ser, 0, Y); 5(par, 0, N); 6(par, 0, N); 7(par, 0, N);
*606.089966* 0.000000* 14.000000** 416.300018*WORST
1:2:0, 1, 0, 4, 5, 6, 7: 0(par, 0, Y); 1(par, 0, N); 2(ser, 0, N); 3(ser, 0, Y);
4(par, 0, Y); 5(par, 0, Y); 6(par, 0, N); 7(par, 0, N);
*183.339966* 0.000000* 123.600006*BEST
1:3:2, 1, 3, 2, 4, 5, 6, 7: 0(ser, 0, N); 1(ser, 0, Y); 2(ser, 0, Y); 3(par, 0, N);
4(par, 0, Y); 5(par, 0, N); 6(par, 0, N); 7(par, 0, N);
*525.099976* 0.000000* 24.000000** 368.700012*
2:0:2, 1, 3, 2, 4, 5, 6, 7: 0(ser, 0, N); 1(ser, 0, Y); 2(ser, 0, Y); 3(par, 0, N);
4(par, 0, Y); 5(par, 0, N); 6(par, 0, N); 7(par, 0, N);
*525.099976* 0.000000* 24.000000** 368.700012*WORST
2:1:0, 1, 0, 4, 5, 6, 7: 0(par, 0, Y); 1(par, 0, N); 2(ser, 0, N); 3(ser, 0, Y);
4(par, 0, Y); 5(par, 0, Y); 6(par, 0, N); 7(par, 0, N);
*183.339966* 0.000000* 123.600006*BEST
2:2:0, 1, 0, 4, 5, 6, 7: 0(par, 0, Y); 1(par, 0, N); 2(ser, 0, N); 3(ser, 0, Y);
4(par, 0, Y); 5(par, 0, Y); 6(par, 0, N); 7(par, 0, N);
*183.339966* 0.000000* 123.600006*BEST
2:3:0, 1, 0, 4, 5, 6, 7: 0(par, 0, Y); 1(par, 0, N); 2(ser, 0, N); 3(ser, 0, Y);
4(par, 0, Y); 5(par, 0, Y); 6(par, 0, N); 7(par, 0, N);
*183.339966* 0.000000* 123.600006*BEST

Table B.11: File showing the fitnesses and features of each circuit throughout the genetic optimisation process. This data can be used to monitor the optimisation process and analyse features of the design space. The actual files produced by the optimisation of the case study circuits in this thesis were between 400kB and 1.3MB in size.
extern "C" bool _export _stdcall Synthesise(bool Make, char* ModName,
    char* Operation, char* DataType, int Accuracy, TStringList* Params);

This function synthesizes the synchronous AHDL module, or determines the pa-
rameters of the module and inserts them into the database. The success of the
function is returned as a boolean value (true on success). The six arguments are:

Make: This boolean describes if the AHDL file should be written (true) or
whether the database should be informed of the parameters of the requested
module (false).

ModName: The data that this character pointer is referencing is updated with the
name prefix of the synthesised AHDL file (such as _mult32s). It is imple-
mented so that the synthesiser can keep track of all the synthesised modules
and their unique filenames. It is usually a concatenation of a shortened form
of the function, its accuracy, and type. Prefixing with an underscore reduces
the likelihood that it will be named identically to a user-written AHDL file.

Operation: This is the function name of the module (such as multiply). It
must be one of the items returned by the Operations() function.

DataType: Describes the data type of module as a string (par, ser, or stoch).

Accuracy: Describes the bit accuracy of the module as an integer.

Params: Provides any module-dependent parameters. For example whether the
module assumes signed data, the delay duration for a delay module, or the
constant of a constant multiplier.

B.6 Distributed Genetic Algorithm Client Processors

The GA client processors calculate the fitness function of the circuit being optimised
given certain parameters by the main synthesiser. These clients can operate on a
different PC to the main synthesiser as long as the PCs are connected via a Local Area
Network (LAN). There are a few things that must be configured so that the clients
perform correctly.

1. The Borland Database Engine must be installed on the client machine. This is
installed automatically if the GA client installation program is run.

2. The clients should all be run from the same directory so that all the (latest)
module-generator DLLs can be found. This is easily achieved by using a Windows
'shortcut' on each machine to the one executable.
3. The same `pdoxusr.s` file must be pointed to by all database engines. That is, it must exist on one machine only. All BDEs must have read and write access to it. It is configured in the ‘BDE Administrator’ utility in the Control Panel of each machine with the BDE installed. The path for the file is known as the NET DIR parameter found under Configuration→Drivers→Native→PARADOX in the administration utility.

When these constraints are met, the client processors can be run before or during an optimisation run. More can be added during a run if more PCs become available, for example. Avoid using computers with a wide range of processing speeds, because the main process must wait for all clients to finish. If one client takes more than twice as long as another, the faster client could have already finished both before the slower client finishes. Shutting down a client process during a GA run should be avoided, as the main synthesiser may wait indefinitely for it to finish its calculation.

### B.7 Settings for the Altera Max+Plus II Environment

The settings for the Altera compiler were kept constant throughout the gathering of results. Table B.12 gives a list of the parameter settings that were used.
<table>
<thead>
<tr>
<th>Synthesis Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global Project Logic Synthesis</strong></td>
<td></td>
</tr>
<tr>
<td>Style</td>
<td>NORMAL</td>
</tr>
<tr>
<td>Optimize Area(0)/Speed(10)</td>
<td>5</td>
</tr>
<tr>
<td>One-Hot State Machine Encoding</td>
<td>Off</td>
</tr>
<tr>
<td>Automatic Fast I/O</td>
<td>Off</td>
</tr>
<tr>
<td>Automatic Register Packing</td>
<td>Off</td>
</tr>
<tr>
<td>Automatic Open-Drain Pins</td>
<td>On</td>
</tr>
<tr>
<td>Automatic Implement in EAB</td>
<td>Off</td>
</tr>
<tr>
<td>Automatic Global Clock</td>
<td>On</td>
</tr>
<tr>
<td>Automatic Global Clear</td>
<td>On</td>
</tr>
<tr>
<td>Automatic Global Preset</td>
<td>On</td>
</tr>
<tr>
<td>Automatic Global Output Enable</td>
<td>On</td>
</tr>
<tr>
<td><strong>Synthesis Style</strong></td>
<td></td>
</tr>
<tr>
<td>Slow Skew Rate</td>
<td>Off</td>
</tr>
<tr>
<td>Ignore SOFT Buffers</td>
<td>On</td>
</tr>
<tr>
<td>Use LPM for AHDL Operators</td>
<td>Off</td>
</tr>
<tr>
<td>Minimization</td>
<td>Full</td>
</tr>
<tr>
<td>Cascade Chain</td>
<td>Ignore</td>
</tr>
<tr>
<td>Carry Chain</td>
<td>Ignore</td>
</tr>
<tr>
<td><strong>Advanced Options</strong></td>
<td></td>
</tr>
<tr>
<td>Decompose Gates</td>
<td>On</td>
</tr>
<tr>
<td>Reduce Logic</td>
<td>On</td>
</tr>
<tr>
<td>Duplicate Logic Extraction</td>
<td>On</td>
</tr>
<tr>
<td>NOT Gate Push-Back</td>
<td>On</td>
</tr>
<tr>
<td>Refactorization</td>
<td>On</td>
</tr>
<tr>
<td>Subfactor Extraction</td>
<td>On</td>
</tr>
<tr>
<td>Multi-Level Factoring</td>
<td>On</td>
</tr>
<tr>
<td>Resynthesize Network</td>
<td>On</td>
</tr>
<tr>
<td>Register Optimization</td>
<td>On</td>
</tr>
</tbody>
</table>

Table B.12: Settings for Max+Plus II compiler that were used throughout the research and testing of the synthesizers.
Appendix C

Circuit Module Grouping

C.1 Introduction

This appendix presents some supplementary material regarding the number of circuit solutions as discussed in chapter 6. The special case of grouping functionally equivalent modules into exactly two groups is covered. Next, the derivation is given of an estimate for the number of circuit solutions made from functionally equivalent modules. Lastly, a comparison is presented between the estimate ($S_L$), the lower bound (the number of module arrangements), and the actual number of solutions, $S_A$ (gained empirically).

C.2 Grouping of Modules into Two Groups

The particular case $G(m,2)$ describes grouping $m$ modules into two groups, and is a useful example for describing the group arrangements. If the two target groups are labeled 0 and 1, binary representations can be used to describe group membership. The number of ways of grouping $m$ modules into exactly two groups is equivalent to the number of arrangements of $m$ 0s and 1s less the two cases where all binary bits are 0s or all are 1s. There are $2^m - 2$ such arrangements. The group labels are unimportant, because the group features are arbitrary. For example, 00111 and 11000 represent the same grouping arrangement. Therefore, only half the groupings are unique. The general case for $i = 2$ is given in equation C.1. Examples can be seen in table 6.1 (on page 123).

$$G(m,2) = \frac{2^m - 2}{2} = 2^{m-1} - 1$$  \hspace{1cm} (C.1)
C.3 Design Space Size Estimation Formula Derivation

This section describes the derivation of the design-space size estimation formulae. The two equations (from chapter 6) are:

$$S_E(m, u) = \sum_{n=1}^{m} S_e(m, n, u), \quad \{m, n, u\} \geq 1 \quad (C.2)$$

and

$$S_e(m, n, u) = \begin{cases} 
  u & m = 1 \\
  2u & n = 1 \text{ and } m \geq 2 \\
  \prod_{j=0}^{n-1}(u-j) & n = m \text{ and } m \geq 2 \\
  2(u-1) \sum_{i=1}^{m-n+1} S_e(m-i, n-1, u) & \text{otherwise}
\end{cases} \quad (C.3)$$

Equation C.2 observes that the total number of solutions is simply the sum of the number of particular solutions of n module-groups. The sum is taken over the possible number of groups: $1 \leq n \leq m$. There can be, at the very least, one group containing all m modules. At most there can be m groups containing one module each. Remember that all the module implementations within each module group are identical.

Equation C.3 describes the number of solutions for m modules that are grouped into n groups, and can each be implemented in u ways. Each of the four cases are dealt with separately.

The first case in equation C.3 ($m = 1$) observes that when there is one module, there are u ways of implementing it. Resource sharing does not have any effect.

The second case in equation C.3 ($n = 1$ and $m \geq 2$) observes that when there is only one group, made up of two or more modules, there are u choices for the module implementation. In addition, the modules can have their resources shared or not. This gives rise to 2u solutions.

The third case in equation C.3 ($n = m$ and $m \geq 2$) is where the number of groups and the number of modules are the same. The first group can be implemented in u ways, the second in $u-1$ ways, and so on. Resource sharing does not affect any of the groups since they each contain a single module. In the case where $n > u$, the equation equates to zero. This zero indicates that no solutions occur that match the criterion $n = m$. Any arrangement would produce fewer than m groups.

The final case in equation C.3 is built from the idea that the number of solutions for n groups can make use of the number of solutions for $n-1$ groups. Consider the

\[1\text{This is similar to } u!/(u-n)!, \text{ except that } n \text{ can be greater than } u.\]
choice for the leftmost group. At its smallest, it can contain one module. At its largest, it can contain $m - n + 1$ modules. If it contained more, there would not be enough modules left to form into $n - 1$ groups.

For each of the ways of choosing the leftmost group, it can be implemented in $2u$ ways.\(^2\) The number of ways of implementing the leftmost group is multiplied by the number of solutions made from the remaining modules (with exactly one fewer groups). So far, this argument yields:

$$S_c(m, n, u) = 2u \sum_{i=1}^{m-n+1} S_c(m - i, n - 1, u)$$

(C.4)

The last ingredient arises from the observation that if the leftmost group of the remaining $(n - 1)$ groups is the same as that chosen for the leftmost group, then it is dealt with by another of the cases. That is, the group is actually larger than that for which the calculation was being performed. Therefore, only $(u - 1)/u$ of the $(n - 1)$ group solutions needs to be included. The multiplication by this term gives the final case that appears in equation C.3.

The estimate always produces an overestimate because there is an overlap of calculated solutions. In general, the overlap is produced because it is not clear in all cases whether there are groups of only one module, in which case the extra (module sharing) solutions are not distinct.

### C.4 Size Estimate Accuracy

Tables C.1 and C.2 show various estimated numbers of circuit solutions and, for each, the corresponding actual number of circuit solutions.

---

\(^2\)Where the size of the leftmost group is one module, this could be $u$, but if the same implementation appears in the remaining groups (which is not explicit), sharing is possible.
<table>
<thead>
<tr>
<th>(m modules)</th>
<th>2 Modules</th>
<th>3 Modules</th>
<th>4 Modules</th>
<th>5 Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 module implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $1^m$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Actual: $S_A(m,1)$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Estimated: $S_E(m,1)$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2 module implementations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $2^m$</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Actual: $S_A(m,2)$</td>
<td>6</td>
<td>16</td>
<td>44</td>
<td>104</td>
</tr>
<tr>
<td>Estimated: $S_E(m,2)$</td>
<td>6</td>
<td>16</td>
<td>52</td>
<td>156</td>
</tr>
<tr>
<td>3 module implementations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $3^m$</td>
<td>9</td>
<td>27</td>
<td>81</td>
<td>243</td>
</tr>
<tr>
<td>Actual: $S_A(m,3)$</td>
<td>12</td>
<td>48</td>
<td>198</td>
<td>786</td>
</tr>
<tr>
<td>Estimated: $S_E(m,3)$</td>
<td>12</td>
<td>48</td>
<td>234</td>
<td>1194</td>
</tr>
<tr>
<td>4 module implementations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $4^m$</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>1024</td>
</tr>
<tr>
<td>Actual: $S_A(m,4)$</td>
<td>20</td>
<td>104</td>
<td>560</td>
<td>3008</td>
</tr>
<tr>
<td>Estimated: $S_E(m,4)$</td>
<td>20</td>
<td>104</td>
<td>656</td>
<td>4568</td>
</tr>
<tr>
<td>5 module implementations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $5^m$</td>
<td>25</td>
<td>125</td>
<td>625</td>
<td>3125</td>
</tr>
<tr>
<td>Actual: $S_A(m,5)$</td>
<td>30</td>
<td>190</td>
<td>1250</td>
<td>8330</td>
</tr>
<tr>
<td>Estimated: $S_E(m,5)$</td>
<td>30</td>
<td>190</td>
<td>1450</td>
<td>12570</td>
</tr>
<tr>
<td>6 module implementations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $6^m$</td>
<td>36</td>
<td>216</td>
<td>1296</td>
<td>7776</td>
</tr>
<tr>
<td>Actual: $S_A(m,6)$</td>
<td>42</td>
<td>312</td>
<td>2412</td>
<td>19032</td>
</tr>
<tr>
<td>Estimated: $S_E(m,6)$</td>
<td>42</td>
<td>312</td>
<td>2772</td>
<td>28452</td>
</tr>
<tr>
<td>7 module implementations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $7^m$</td>
<td>49</td>
<td>343</td>
<td>2401</td>
<td>16807</td>
</tr>
<tr>
<td>Actual: $S_A(m,7)$</td>
<td>56</td>
<td>476</td>
<td>4214</td>
<td>38234</td>
</tr>
<tr>
<td>Estimated: $S_E(m,7)$</td>
<td>56</td>
<td>476</td>
<td>4802</td>
<td>56546</td>
</tr>
<tr>
<td>8 module implementations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $8^m$</td>
<td>64</td>
<td>512</td>
<td>4096</td>
<td>32768</td>
</tr>
<tr>
<td>Actual: $S_A(m,8)$</td>
<td>72</td>
<td>688</td>
<td>6848</td>
<td>70016</td>
</tr>
<tr>
<td>Estimated: $S_E(m,8)$</td>
<td>72</td>
<td>688</td>
<td>7744</td>
<td>102384</td>
</tr>
</tbody>
</table>

Table C.1: Solutions of circuits made from functionally identical modules. The table shows a comparison between the actual module combinations, $S_A(m,n)$, and estimates, $n^m$ and $S_E(m,n)$. The number of modules is in the range $2 \leq m \leq 5$. 
<table>
<thead>
<tr>
<th>(m modules)</th>
<th>6 Modules</th>
<th>7 Modules</th>
<th>8 Modules</th>
<th>9 Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 module implementation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $1^m$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Actual: $S_A(m,1)$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Estimated: $S_E(m,1)$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>2 module implementations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $2^m$</td>
<td></td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>Actual: $S_A(m,2)$</td>
<td></td>
<td>228</td>
<td>480</td>
<td>988</td>
</tr>
<tr>
<td>Estimated: $S_E(m,2)$</td>
<td></td>
<td>468</td>
<td>1404</td>
<td>4212</td>
</tr>
<tr>
<td><strong>3 module implementations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $3^m$</td>
<td>729</td>
<td>2187</td>
<td>6561</td>
<td>19683</td>
</tr>
<tr>
<td>Actual: $S_A(m,3)$</td>
<td>3018</td>
<td>10926</td>
<td>37506</td>
<td>124218</td>
</tr>
<tr>
<td>Estimated: $S_E(m,3)$</td>
<td>5970</td>
<td>29850</td>
<td>149250</td>
<td>746250</td>
</tr>
<tr>
<td><strong>4 module implementations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $4^m$</td>
<td>4096</td>
<td>16384</td>
<td>65536</td>
<td>262144</td>
</tr>
<tr>
<td>Actual: $S_A(m,4)$</td>
<td>15992</td>
<td>82832</td>
<td>415832</td>
<td>2012960</td>
</tr>
<tr>
<td>Estimated: $S_E(m,4)$</td>
<td>32120</td>
<td>224840</td>
<td>1573880</td>
<td>11017160</td>
</tr>
<tr>
<td><strong>5 module implementations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $5^m$</td>
<td>15625</td>
<td>78125</td>
<td>390625</td>
<td>1953125</td>
</tr>
<tr>
<td>Actual: $S_A(m,5)$</td>
<td>55650</td>
<td>368490</td>
<td>2402170</td>
<td>15324130</td>
</tr>
<tr>
<td>Estimated: $S_E(m,5)$</td>
<td>113010</td>
<td>1018050</td>
<td>9162450</td>
<td>82462048</td>
</tr>
<tr>
<td><strong>6 module implementations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $6^m$</td>
<td>46656</td>
<td>279936</td>
<td>1679616</td>
<td>10077696</td>
</tr>
<tr>
<td>Actual: $S_A(m,6)$</td>
<td>151692</td>
<td>1210032</td>
<td>9597492</td>
<td>81448808</td>
</tr>
<tr>
<td>Estimated: $S_E(m,6)$</td>
<td>309372</td>
<td>3402372</td>
<td>37433292</td>
<td>411766208</td>
</tr>
<tr>
<td><strong>7 module implementations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $7^m$</td>
<td>117649</td>
<td>823543</td>
<td>5764801</td>
<td>40353608</td>
</tr>
<tr>
<td>Actual: $S_A(m,7)$</td>
<td>352058</td>
<td>3263750</td>
<td>31686620</td>
<td>198568816</td>
</tr>
<tr>
<td>Estimated: $S_E(m,7)$</td>
<td>717458</td>
<td>9296714</td>
<td>120852240</td>
<td>1571139584</td>
</tr>
<tr>
<td><strong>8 module implementations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple: $8^m$</td>
<td>262144</td>
<td>2097152</td>
<td>16777216</td>
<td>134217728</td>
</tr>
<tr>
<td>Actual: $S_A(m,8)$</td>
<td>728688</td>
<td>7664736</td>
<td>86586800</td>
<td>377649312</td>
</tr>
<tr>
<td>Estimated: $S_E(m,8)$</td>
<td>1478640</td>
<td>22011600</td>
<td>329891776</td>
<td>4948336128</td>
</tr>
</tbody>
</table>

Table C.2: Solutions of circuits made from functionally identical modules. The table shows a comparison between the actual module combinations, $S_A(m,n)$, and estimates, $n^m$ and $S_E(m,n)$. The number of modules is in the range $6 \leq m \leq 9$. 
References


REFERENCES


