PERFORMANCE-ORIENTED
SYNTAX-DIRECTED SYNTHESIS OF
ASYNCHRONOUS CIRCUITS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2010

By
Luis Tarazona Duarte
School of Computer Science
Contents

Abstract 14
Declaration 15
Copyright 16
Acknowledgements 17

1 Introduction 19
  1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
  1.2 Syntax-directed synthesis . . . . . . . . . . . . . . . . . . . . . . 20
    1.2.1 Tangram and TiDE . . . . . . . . . . . . . . . . . . . . . . 21
    1.2.2 Balsa . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
    1.2.3 Handshake circuits and handshake components . . . . . . 21
  1.3 Optimising handshake circuits . . . . . . . . . . . . . . . . . . . 24
    1.3.1 Push data-driven handshake circuits . . . . . . . . . . . . 25
    1.3.2 Automated source-to-source transformations . . . . . . . 26
    1.3.3 Behavioural synthesis of asynchronous circuits . . . . . . 26
  1.4 Teak . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  1.5 Aims of this research . . . . . . . . . . . . . . . . . . . . . . . 28
  1.6 Contribution of this research . . . . . . . . . . . . . . . . . . . 29
  1.7 Thesis organisation . . . . . . . . . . . . . . . . . . . . . . . . 29
  1.8 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

2 Background 32
  2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
  2.2 Asynchronous Circuits . . . . . . . . . . . . . . . . . . . . . . . 32
  2.3 Handshake protocols and data encoding . . . . . . . . . . . . . 34
2.3.1 Bundled-data protocols ........................................ 35
2.3.2 Dual-rail protocols ........................................... 38
2.4 Operation modes .................................................. 40
  2.4.1 Fundamental Mode Circuits .................................. 41
  2.4.2 Burst-Mode circuits .......................................... 42
  2.4.3 Input-output mode ............................................ 42
2.5 Delay models .................................................... 42
  2.5.1 Speed-independent (SI) circuits ............................. 43
  2.5.2 Delay-insensitive (DI) circuits ............................. 43
  2.5.3 Quasi-delay insensitive (QDI) circuits ......................... 44
2.6 Asynchronous synthesis ......................................... 45
  2.6.1 Synthesis of SI control circuits ............................. 45
  2.6.2 Communicating Hardware Processes (CHP) and the Cal-
        tech Asynchronous Synthesis Tool (CAST) .................. 48
  2.6.3 Macromodular based synthesis ............................. 50
  2.6.4 Desynchronisation methods .................................. 50
2.7 Summary ........................................................ 53

3 The Balsa synthesis system and language ..................... 54
  3.1 Introduction ..................................................... 54
  3.2 The Balsa synthesis system ..................................... 54
    3.2.1 Balsa design flow ......................................... 55
  3.3 The Balsa language ............................................. 55
    3.3.1 The structure of a Balsa description ....................... 56
    3.3.2 Data Types ................................................. 58
    3.3.3 Basic transfer commands .................................... 59
    3.3.4 Dataless handshakes ....................................... 60
    3.3.5 Variable assignment ....................................... 60
    3.3.6 Control operators ......................................... 60
    3.3.7 Iteration and conditional constructs ....................... 62
    3.3.8 Data processing operators ................................ 64
    3.3.9 Input enclosure ............................................ 64
    3.3.10 Arbitration ................................................. 67
    3.3.11 Permissive Concur ........................................ 68
    3.3.12 Compilation examples ..................................... 68
    3.3.13 Interconnecting Balsa modules ............................ 78
4 Optimising Balsa circuits

4.1 Introduction

4.2 Related work

4.3 The data-driven description style

4.3.1 Control driven to data driven example

4.4 Optimising data-driven descriptions

4.4.1 Separating actions into concurrent loops

4.4.2 Broadcasting values

4.4.3 Adding pipeline registers

4.5 Optimising guards

4.5.1 Encoding multiple guards

4.6 New peephole optimisations

4.6.1 Removing redundant False Variables

4.6.2 Control of active enclosures

4.6.3 Unbounded read-then-write on variables

4.6.4 Fetch component with concurrent RTZ

4.6.5 Summary

5 Optimising Token-flow circuits and descriptions

5.1 Introduction

5.2 The Teak system

5.2.1 Teak components

5.2.2 Teak synthesis

5.3 Optimising Teak circuits

5.3.1 Variables

5.3.2 Fork displacement

5.3.3 Fork-Merge-Join and Steer-Merge

5.3.4 Removing “go” cycles

5.4 Description-level optimisations

5.4.1 Commonalities with Balsa optimisations

5.4.2 Description techniques to remove Variables

5.4.3 Summary
6 Latch insertion in Teak circuits

6.1 Introduction ................................................. 144
6.2 Buffering cycles ............................................. 145
  6.2.1 Detecting cycles ........................................ 146
  6.2.2 Complexity of finding the optimum latch insertion points . 148
6.3 Buffering single-token cycles ................................. 151
6.4 Two simple latching strategies for Teak circuits .............. 153
  6.4.1 Analysis of the latching strategies ....................... 156
6.5 Specifying latching and optimisation options in Teak .......... 159
6.6 Summary ...................................................... 159

7 Design Examples and Evaluation ............................... 161

7.1 The nanoSpa processor ....................................... 161
  7.1.1 The Fetch stage ......................................... 162
  7.1.2 The Decode stage ....................................... 163
  7.1.3 The Execute stage ...................................... 164
  7.1.4 Results .................................................. 166
7.2 An asynchronous Viterbi decoder .............................. 170
  7.2.1 Introduction ............................................. 170
  7.2.2 Viterbi decoder algorithm ............................... 171
  7.2.3 Architecture of the asynchronous Viterbi decoder ........ 172
  7.2.4 Results .................................................. 174
7.3 A 32×32-bit radix-8 Booth MAC ............................... 177
  7.3.1 32-bit Multiply with 64-bit accumulation ................. 179
  7.3.2 Results .................................................. 179
7.4 The nanoSpa Forwarding Unit ................................ 182
  7.4.1 Introduction ............................................. 182
  7.4.2 Related work ............................................ 183
  7.4.3 The target processor: nanoSpa .......................... 185
  7.4.4 Architecture of the nanoForward Unit .................... 185
  7.4.5 Implementation issues .................................. 187
  7.4.6 Use of the permissive Concur ........................... 189
  7.4.7 Results .................................................. 190
7.5 A sliced-channel wormhole router ............................ 191
  7.5.1 Introduction ............................................. 192
  7.5.2 Architecture of the sliced-channel wormhole router ..... 192
7.5.3 Results .......................... 194
7.6 Summary .......................... 195
  7.6.1 Balsa .......................... 195
  7.6.2 Teak .......................... 196

8 Conclusions and future work 197
  8.1 Balsa .......................... 197
  8.2 Teak .......................... 198
  8.3 Future work ...................... 199
    8.3.1 Description-level optimisations ......... 199
    8.3.2 Peephole optimisations .................. 199
    8.3.3 Synthesis using hybrid style .......... 200
    8.3.4 Teak .......................... 200

References 202

A List of Balsa operators 215

B Balsa handshake components 216
  B.1 Control components ................. 216
    B.1.1 Loop .......................... 216
    B.1.2 Concur ........................ 217
    B.1.3 Fork .......................... 217
    B.1.4 WireFork ......................... 217
    B.1.5 Sequence ......................... 218
    B.1.6 Call .......................... 218
    B.1.7 Sync .......................... 218
    B.1.8 Arbitrate ....................... 218
    B.1.9 DecisionWait .................... 219
  B.2 Datapath components ............... 219
    B.2.1 Unary function ................... 219
    B.2.2 Binary function .................. 219
    B.2.3 CallMux ........................ 219
    B.2.4 SplitEqual ....................... 220
    B.2.5 CaseFetch ....................... 220
    B.2.6 PassivatorPush ................... 220
    B.2.7 Variable ......................... 220
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.3</td>
<td>Control to datapth interface components</td>
<td>221</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Fetch</td>
<td>221</td>
</tr>
<tr>
<td>B.3.2</td>
<td>While</td>
<td>221</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Case</td>
<td>221</td>
</tr>
<tr>
<td>B.3.4</td>
<td>FalseVariable</td>
<td>222</td>
</tr>
<tr>
<td>B.3.5</td>
<td>activeEagerFalseVariable</td>
<td>222</td>
</tr>
<tr>
<td>C</td>
<td>$FV$ and $aeFV$ implementations</td>
<td>223</td>
</tr>
<tr>
<td>D</td>
<td>Optimised Viterbi decoder Balsa description</td>
<td>226</td>
</tr>
<tr>
<td>E</td>
<td>Optimised 32x32 bit Booth multiplier Balsa description</td>
<td>238</td>
</tr>
<tr>
<td>F</td>
<td>Optimised sliced-channel wormhole router Balsa description</td>
<td>250</td>
</tr>
<tr>
<td>G</td>
<td>Optimised nanoSpa forwarding unit Balsa description</td>
<td>263</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Dual-rail encoding for 1-bit .............................................. 39

4.1 BMU Simulation results. ................................................. 89
4.2 GCD Simulation results. ................................................. 98
4.3 Influence of data widths in first-read-unfold of read-write unbounded repetitions .......................................................... 109

7.1 Performance, area and energy for three different versions of nanoSpa. 166
7.2 Balsa nanoSpa performance, area and energy results. ............... 168
7.3 Teak nanoSpa performance, area and energy results. ............... 169
7.4 Comparison of the Balsa and Teak nanoSpa implementations. .... 170
7.5 Performance, area and energy results for the Viterbi decoder in Balsa ................................................................. 175
7.6 Performance, area and energy results for the Viterbi decoder in Teak 176
7.7 Comparison of the Viterbi decoder in Balsa and Teak ............... 177
7.8 Performance, area and energy results for the MAC unit in Balsa .. 181
7.9 Performance, area and energy results for the MAC unit in Teak .. 181
7.10 Comparison of the MAC implementations using Balsa and Teak .. 182
7.11 Performance results for nanoSpa using the nFU .................. 191
7.12 Energy results for nanoSpa using the nFU ....................... 191
7.13 Balsa wormhole router simulation results. ......................... 195

A.1 Balsa binary/unary operators [30]. .................................. 215
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Handshake Circuit composed of a Transferrer (→) and a False-Variable (FV) handshake components.</td>
</tr>
<tr>
<td>1.2</td>
<td>1-place buffer: (a) Balsa description, (b) handshake circuit.</td>
</tr>
<tr>
<td>1.3</td>
<td>Teak circuit for the 1-place buffer.</td>
</tr>
<tr>
<td>2.1</td>
<td>Bundled-data channels.</td>
</tr>
<tr>
<td>2.2</td>
<td>Two-phase bundled-data protocol.</td>
</tr>
<tr>
<td>2.3</td>
<td>Four-phase bundled-data protocol.</td>
</tr>
<tr>
<td>2.4</td>
<td>Four-phase data-validity schemes for a push channel.</td>
</tr>
<tr>
<td>2.5</td>
<td>Four-phase data-validity schemes for a pull channel.</td>
</tr>
<tr>
<td>2.6</td>
<td>Four-phase dual-rail protocol. (a) push channel, (b) timing diagram.</td>
</tr>
<tr>
<td>2.7</td>
<td>n-bit four-phase dual-rail protocol in a push channel.</td>
</tr>
<tr>
<td>2.8</td>
<td>Two-phase dual-rail protocol in a 2-bit push channel.</td>
</tr>
<tr>
<td>2.9</td>
<td>The Muller C-element.</td>
</tr>
<tr>
<td>2.10</td>
<td>A T-element connected to left and right “well behaved” environments and its specification in the form of a timing diagram, a Petri net and an STG.</td>
</tr>
<tr>
<td>2.11</td>
<td>A dual-rail full adder using NCL gates.</td>
</tr>
<tr>
<td>3.1</td>
<td>Balsa design flow.</td>
</tr>
<tr>
<td>3.2</td>
<td>The structure of a Balsa description.</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of deadlocking code.</td>
</tr>
<tr>
<td>3.4</td>
<td>An uncontrolled multiplexer (merge).</td>
</tr>
<tr>
<td>3.5</td>
<td>Handshake circuit of the uncontrolled multiplexer.</td>
</tr>
<tr>
<td>3.6</td>
<td>The description of a simple two-input adder.</td>
</tr>
<tr>
<td>3.7</td>
<td>Handshake circuit of the adder code in figure 3.6.</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of conditional execution.</td>
</tr>
<tr>
<td>3.9</td>
<td>Handshake circuit of the code in figure 3.8.</td>
</tr>
</tbody>
</table>
3.10 An example of a finite loop and command composition. .... 73
3.11 Handshake circuit of the code in figure 3.10. ................. 73
3.12 Example of unsafe use of active eager enclosure. ............... 75
3.13 Using the permissive Concur with mutually exclusive writes. .. 77
3.14 Example of merging channels using the select construct. .... 77
3.15 Example of merging channels using the permissive Concur opera-
tor. ............................................................ 78
3.16 (a) Interfacing of two Balsa modules with active ports using Pas-
sivators. (b) A 1-bit dual-rail PassivatorPush. ................. 79

4.1 The simplified control-driven SPA EXECUTE stage [85]. .... 83
4.2 The simplified data-driven SPA EXECUTE stage [85]. .... 84
4.3 Branch metric computation for a Viterbi decoder [91]. .... 85
4.4 Initial BMU description. .................................................. 86
4.5 Handshake circuit of the BMU. ........................................... 87
4.6 First operations of the BMU: (a) original, (b) Data-driven. .... 87
4.7 Optimised BMU description. .................................................. 88
4.8 Handshake circuit of the optimised BMU. ......................... 89
4.9 Example of separating actions into concurrent loops (first steps). 91
4.10 Simulation results of different optimisations applied to the BMU. 92
4.11 Broadcasting: (a,c) Implicit broadcasting. (b,d) Explicit duplication. 94
4.12 Pipelining: (a,c) using variables. (b,d) using explicit pipeline buffers. 96
4.13 A pseudo-code specification of GCD [91]. ...................... 97
4.14 Two implementations of the GCD algorithm in Balsa and their compiled handshake circuits. 99
4.15 Simplified description of the South input buffer of a sliced-channel wormhole router [90]. 101
4.16 Optimised, simplified description of the South input buffer. .... 102
4.17 Handshake circuit for example in figure 3.6, (a) original, (b) optimised. 103
4.18 (a) Fork implementation. (b) Synch implementation. .......... 104
4.19 Permanent active eager input: (a) original, (b) with optimised control. 105
4.20 (a) S-element. (b) T-element. (c) S-element STG. (d) T-element STG. 106
4.21 Balsa sequencers: (a) based on the S-element, (b) based on the T-element [89].

4.22 Read-write loop: (a) code, (b) handshake circuit.

4.23 First-read-unfolded version of circuit in figure 4.22.

4.24 Optimised first-read-unfolded read-write loop.

4.25 Conventional Balsa dual-rail Fetch: (a) circuit, (b) STG.

4.26 Fetch with concurrent RTZ: (a) circuit, (b) STG.

5.1 Teak components.

5.2 (a) Teak circuit for 1-place buffer, (b) Handshake circuit for 1-place buffer.

5.3 Balsa-style channel implementation.

5.4 Multiple-output channel implementation.

5.5 Channel component optimisation.

5.6 Sequential/parallel composition.

5.7 While loop implementation.

5.8 Variable single read-after-write optimisation.

5.9 Balsa code for n-bit full adder.

5.10 Variable substitution example.

5.11 Sequential write to a channel.

5.12 Sequenced channel write example: (a) original, (b) after Fork displacement.

5.13 “Sign adjust” example.

5.14 “Sign adjust” circuit: (a) first optimisation steps, (b) final circuit.

5.15 Teak circuit of the N-bit adder: (a) Optimised, (b) With the “go” cycle removed.

5.16 Avoiding Variables associated with conditional reads.

5.17 Discarding inputs conditionally in Teak: (a, c) Balsa-optimised style; (b, d) Teak-optimised style.

5.18 Joining inputs to reduce the tagging circuitry.

5.19 Simulation results for different optimised versions of the mux example.

5.20 steerAlu example: (a) original, (b) channel duplication to avoid conditional reads.

5.21 steerAlu Teak circuit.

5.22 Optimised steerAlu Teak circuit.

5.23 Simulation results for the steerAlu example.
5.24 Circuit-level conditional reads removal. ........................................ 140
5.25 Circuit-level optimisation of the steerAlu module. ....................... 141

6.1 The Teak single-token loop Merge - Logic block - Fork structure. 145
6.2 (a) A directed graph and (b) a depth-first forest of the graph. .... 146
6.3 Mapping of a Teak circuit into a directed graph. ......................... 148
6.4 DFS forest of graph in figure 6.3(b). ........................................ 148
6.5 Optimised Teak circuit for the GCD description in figure 4.14(b) 150
6.6 Strategy for latching all cycles of the GCD circuit of figure 6.5 based in latching back edges. .............................................. 152
6.7 Latching strategy “A” for single-token M-LB-F blocks: (a) Teak circuit, (b) logic circuit, (c) dependency graph. ....................... 154
6.8 Latching strategy “B” for single-token M-LB-F blocks: (a) Teak circuit, (b) logic circuit, (c) dependency graph. ....................... 155
6.9 Comparison of both sides of inequality 6.6 for different data widths. 158
6.10 Example of passing options at procedure-level. ......................... 159

7.1 The 3-stage nanoSpa pipeline showing details of the Decode stage. 163
7.2 Simplified nanoSpa Execute stage. ............................................ 165
7.3 A convolutional encoder with $k = 3$ and code ratio $= 1/2$. .... 171
7.4 Trellis diagram for the encoder in figure 7.3. ............................... 171
7.5 Architecture of the asynchronous Viterbi decoder. ....................... 172
7.6 The Path Metric Unit. ............................................................. 173
7.7 The History Unit. ................................................................. 174
7.8 Architecture of the nanoSpa multiplier unit. ............................... 178
7.9 An “X-ray” picture of the Booth-3 Handshake Circuit revealing its control tree. ......................................................... 180
7.10 Potential performance benefits of result forwarding in a 4-stage pipeline. ................................................................. 183
7.11 AQF process model. ............................................................... 184
7.12 The 5-stage nanoSpa pipeline. .................................................. 186
7.13 The nanoForward Unit architecture .......................................... 186
7.14 Inter-process dependencies in the nFU. ..................................... 187
7.15 Composition of actions with the permissive Concur inside the nFU. 190
7.16 Wormhole NoC datapath [90]. .................................................. 193
7.17 Sliced-channel wormhole router with four sub-channels [90]. ..... 194
C.1 FalseVariable: (a) Implementation, (b) STG. . . . . . . . . . . . . 224
C.2 activeEagerFalseVariable: (a) Implementation, (b) STG. . . . . 225
Abstract

This thesis evaluates the capabilities and limitations of the syntax-directed approach to synthesise high-performance asynchronous systems and proposes a number of optimisations to improve the performance of the synthesised circuits.

The first part of this work explores new methods for improving the performance of asynchronous circuits synthesised from syntax-directed descriptions, targeting handshake circuits and using the Balsa synthesis system as the research framework. This includes investigating description styles and the use of language constructs that result in faster circuits. A number of new peephole optimisations based on the previous observations are also presented.

The second part investigates the performance of a new, token-flow based synthesiser for the Balsa language called Teak. A set of optimisations based in circuit transformations and buffering strategies are proposed in order to improve the performance of Teak circuits. These optimisations have been automated and incorporated into the Teak synthesiser.

All optimisations target dual-rail, quasi-delay-insensitive implementations as this is a robust approach that helps to reduce the impact of the timing closure problem within modern fabrication processes variability. The techniques and optimisations presented here has been tested in a set of non-trivial examples including a 32-bit RISC processor.

The use of the proposed techniques result in optimised compositions of handshake circuits and Teak components that generally synthesise into faster circuits.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of Computer Science (or the Vice-President).
Acknowledgements

I would like to express my deep gratitude to my parents Nelly and Luis for all the love and support they have always given me. My life has been blessed by having them as my parents. Thanks to all my brothers and sisters, nieces and nephews for their support and their love. I have been missing you all a lot.

Thanks to my supervisor Doug Edwards who gave me the opportunity of doing this PhD under his supervision. Thanks to my fine proof readers Andrew Bardsley and Will Toms for all the careful reading and comments done on my writing. I must thank Andrew twice for giving me the opportunity of working with him on his Teak system and having introduced me to the beauties of the Haskell language. Many thanks to Charlie Brej for all his support, nice food, friendship and for doing the final proof read of my thesis. Thanks to Wei Song for kindly allowing me to use his router design. Thanks to Mikel Luján for his advice and support.

My special gratitude to Luis Plana for all his support, advice and encouragement. Many thanks “Luis One”. Thanks to the whole Plana Oteiza family for being my family here in Manchester. Thanks to all my friends and families from Barquisimeto, Venezuela, specially to my brother José Ortiz and his family: ¡Gracias José! Thank you dear Emilia for giving me your love and support during all these years.

Thanks to the people I have met in the APT group. I have learnt many things from them. I am sure I will forget many names if I try to list them.

Finally, thanks to the School of Computer Science and EPSRC for the financial support I have received during my PhD.
To my Parents, my nieces and nephews, my brotheres and sisters, and my families from Barquisimeto.

A mis Padres, mis sobrinas y sobrinos, mis hermanas y hermanos, y a mis familias de Barquisimeto.
Chapter 1

Introduction

1.1 Motivation

Asynchronous design has regained interest in recent years due to its potential advantages over its synchronous (clocked) counterpart. Synchronous digital systems, which are the basis of most of today’s digital designs, are based on two major assumptions: all signals are binary and time is discrete, defined by the system’s clock signal which controls all communication and event sequencing. These assumptions simplify greatly the task of design but also lead to clock distribution and clock skew problems, increased power consumption, coherent electromagnetic emissions and forcing all parts of the circuit to work at the same (worst-case) rate.

Unlike synchronous systems, asynchronous systems do not rely on a global clock signal. Instead, these systems use a form of local communication that comprises *handshake signals* to request (initiate) the start of an operation and acknowledge (indicate) to determine its completion.

Asynchronous circuits have some potential advantages over their synchronous counterparts that make them attractive to use in large VLSI designs. These include: no clock distribution or clock skew problems, better modularity and composability, less coherent electromagnetic emissions, automatic power management, average-case performance and robustness towards variations in supply voltage, temperature and fabrication process parameters [91, 107, 35, 48, 37, 39].

Synchronous design has the advantage of being a mature technology supported
by many commercially available Computer-Aided Design (CAD) tools and implementation alternatives, which cannot be used or provide little support for asynchronous design. The increased interest in asynchronous design has led to the development of several methodologies and CAD tools targeted specifically at asynchronous design. Among these, Balsa [5] and TiDE\textsuperscript{TM} (formerly Tangram) [23, 83] are fully-automated systems that have successfully synthesised large-scale circuits using a process called \textit{syntax-directed synthesis}. Although these tools greatly improve the design time for a complex system, there is evidence that shows that this is done at the cost of reduced performance when compared to manual, full-custom design approaches [36, 84, 11]. If the performance penalty imposed by the syntax-directed synthesis could be reduced, then this asynchronous design methodology could be used in performance-demanding, real-world applications. These applications could then benefit from some of the potential advantages of asynchronous circuits.

This thesis explores new methods for improving the performance of asynchronous circuits synthesised using the syntax-directed approach. In this work, the Balsa synthesis system has been used as the research framework. The work includes investigating description styles and language constructs that result in faster circuits, new peephole optimisations based on these observations, and the analysis and optimisation of a novel token-flow implementation for the Balsa language, using a new system called \textit{Teak}, introduced initially in [6]. The synthesis targets dual-rail, quasi-delay-insensitive implementation (QDI – see section 2.5.3) as this is a robust approach that helps to reduce the impact of increasingly difficult timing closure within modern fabrication processes variability.

1.2 Syntax-directed synthesis

The syntax-directed approach to synthesise asynchronous circuits is based in the compilation of descriptions written in a high-level language into a communicating network of pre-designed modules. The compilation process performs a mapping of each language construct into the network of modules that implements it. This mapping gives a high degree of transparency in the design as incremental changes to the specification generates predictable changes in the resulting circuit. This transparency allows the designer to optimise the circuit, in terms of performance,
power or area, at the language level. The compiled network of components constitutes an intermediate representation that can subsequently be expanded into gate netlists.

Currently there exists two fully automated CAD systems that use this approach for the synthesis of asynchronous systems: *Timeless Design Environment* (*TiDE* [23] formerly *Tangram* [10, 8]) a proprietary system developed at Philips Research Laboratories, and *Balsa* [5, 29], an open-source system developed at the University of Manchester that closely follows the Tangram philosophy.

### 1.2.1 Tangram and TiDE

Tangram uses a CSP-like description language (the language is also called Tangram), but with a syntax more similar to traditional programming languages than CSP. Tangram has been used to successfully develop complex asynchronous chips [39, 105, 58]. The Tangram synthesis system has evolved into *TiDE*, and the new version of the Tangram language is now called *Haste* since the Tangram system began to form part of the product portfolio of *Handshake Solutions* [23].

### 1.2.2 Balsa

Balsa is an open-source package and is freely available from [4]. The Balsa system is still under development and, from version 3.5.1, incorporates a GUI user interface with facilities such as project management, editor and behavioural graphical simulator. Balsa is the name for both the framework for synthesising asynchronous circuits and the language used to describe such systems. The Balsa language has support for parameterisation and recursive procedures, records and symbolic enumerated types, has greater expressiveness than Tangram and is also more “human readable”.

In both Tangram and Balsa approach to syntax-directed synthesis, the resulting communicating network of components interact using *handshake signals*. These networks of *handshake components* are called *handshake circuits*.

### 1.2.3 Handshake circuits and handshake components

A handshake circuit is a communicating network of handshake components (handshake modules) connected point-to-point using handshake channels (see section 2.3). Each channel connects exactly one *passive port* of a handshake component to an
active port of another handshake component. An active port is a port that initiates the communication by sending a request signal to a passive port. When ready, the passive port will respond with the acknowledge signal. The handshake can involve the transfer of data or control to synchronise two processes.

Figure 1.1: A Handshake Circuit composed of a Transferrer (→) and a False-Variable (FV) handshake components.

Figure 1.1(a) shows details of a handshake circuit composed of two handshake components: a Transferrer (→) and a FalseVariable (FV) component with two read ports. In the figure, the components are represented by larger circles, passive ports by small unfilled circles, and active ports by small filled circles. Data-less control channels are composed of a request and an acknowledge pair of wires. Data are represented by thick arrows signalling the direction of data. In figure 1.1(a), data wires using binary signalling are bundled together with the req and ack pair to form a bundled-data channel.
Handshake circuits are not normally represented at the level of detail in figure 1.1(a). A simplified diagram as shown in 1.1(b) is preferred, where the channels carrying data are represented as a single arc with an arrowhead signalling the direction of data; the control (synchronisation) channels are represented by single arcs. Control direction is implied by the type of port involved.

The circuit operation is as follows:

i. The circuit starts its operation when a request is made to the Transferrer component on its upper (activate) passive port. Upon receiving this, the Transferrer issues a request to its environment connected to its active, left port. This left port is an example of a pull data channel (data flows from the passive port to the active port). The right port of the Transferrer is an example of a push data channel (data flows from the active port to the passive port).

ii. Eventually, the environment will respond with the data and the acknowledgement. The Transferrer in turn passes it as a request to the FalseVariable component at its right.

iii. The FalseVariable receives this request and issues a request on its synchronisation signal port, indicating that another process can safely read data from the read ports until the handshake in the signal port has been acknowledged.

iv. The environment connected to the read and signal ports may read the data zero or more times and when done, sends an acknowledgement on signal.

v. After receiving the acknowledgement on signal, the FalseVariable sends back an acknowledgement to the Transferrer which in turn passes it to the activate channel.

vi. The environment connected to the activate will eventually remove the request, which in turn causes the Transferrer to finish the handshake in its left channel, terminating the transfer. Note that the handshake on the activation port of the Transferrer encloses full handshakes on its input and output ports.

As an example of syntax-directed translation into handshake circuits, consider the Balsa specification for a simple 1-place buffer (register) shown in figure 1.2(a).
The specification is parameterised in the type of data the register can hold. The register has an input channel `inp` and an output channel `out`. The variable `v` stores the data and the operation consists of an infinite repetition (`loop`) of two actions: transfer of data (`<-`) from channel `inp` into `v` sequenced (`;`) with the transfer (`->`) of data stored in `v` to the channel `out`.

```plaintext
procedure buffer
{
  parameter DataType : type;
  input inp : DataType;
  output out : DataType
} is
  variable v : DataType
begin
  loop
    inp -> v;
    out <- v
  end
end
```

![Diagram](image)

Figure 1.2: 1-place buffer: (a) Balsa description, (b) handshake circuit.

Figure 1.2(b) shows the handshake circuit generated by Balsa from the code in 1.2(a). A `Transferrer` component (`→`) connects the input channel to the write port of the `Variable` component (`v`) that acts as the variable `v` of the description. The read port of `v` is connected to the output using a second `Transferrer`. A `Sequencer` (`;`) is used to sequence the writing to and the reading from `v`, and a `Loop` (`*`) component activates the `Sequencer` repeatedly. Given that many handshake components have simple implementations (for instance, a `Transferrer` can be implemented using only wires and the `Loop` using a NOR gate), the resulting synthesised circuit is not complex.

### 1.3 Optimising handshake circuits

As stated before, the syntax-directed syntax paradigm is attractive in terms of flexibility and compilation simplicity, but these come at the cost of low to moderate performance. In general, Balsa/Tangram translation generates a datapath section together with a control tree which mirrors the control flow of the language description as shown in figure 1.2(a). For this reason, the translation is
also described as control-driven. The overhead of this control-driven approach has been identified as one of the major causes of performance penalty in handshake circuits.

Previous work in handshake circuits optimisation include peephole optimisations [83], more concurrent designs for handshake components [85] and control resynthesis [19, 33]. The following sections introduce recent work on the optimisation of handshake circuits.

1.3.1 *Push* data-driven handshake circuits

In an attempt to reduce this penalty, Taylor [100] introduced a novel data-driven circuit style, together with a new description language and compiler, which produced significant performance increases in the synthesised circuits compared to those generated by conventional Balsa/Tangram. This approach is based on reducing the control overhead by using the following techniques:

- all control is activated in parallel.
- sequencing is localised to storage elements (variables). This ensures that storage elements are not concurrently read and written and allows the read and write sections of control to operate in parallel.
- data processing makes use of push-only structures and operations are speculatively executed to allow control and data sections to operate in parallel.

The techniques above are enforced by a more restrictive description language syntax. In particular, variables have a write-once, read-once behaviour, which means that they must be read every time they are written and they must be written before they can be read. Also, conditional multicasting of a channel value is replaced by speculative broadcasting to all possible destinations together with a rejection mechanism to discard unwanted data at the places where the condition states that data is not required. These and other restrictions make the generation of very small, localised control trees, possibly reducing the control overhead and improving performance. However, this is done at the expense of significantly larger area, energy use and reduced flexibility at the description level.
1.3.2 Automated source-to-source transformations

In [47] Hansen and Singh describe a series of automated “source-to-source” transformations that optimises syntax-directed descriptions using a variety of concurrency enhancing optimisations. Although considerable speed-ups are claimed, some of the examples used start with extremely naïve code sequences, so it is easy to obtain significant improvements. Also, their proposed approach is limited to slack elastic [64] systems descriptions only (a slack elastic system preserves correct operation even if extra pipeline buffer stages are introduced in any channel). This limitation reduces the usefulness of an “automated” approach as it is frequently necessary for the designer to understand the nature of the transformations to ensure they are safe, which may represent a considerable extra design effort to the user.

1.3.3 Behavioural synthesis of asynchronous circuits

In another recent work, Nielsen et al. [75, 76] presented a method for automatic behavioural synthesis of asynchronous circuits using syntax-directed translation as backend. The initial development was based on the Balsa framework but the final automated tool targets the Haste/TiDE design flow. Input to the tool is a behavioural description in the Haste language (both Haste and Balsa are behavioural languages). From this description, the tool extracts a Control Data Flow Graph, CDFG [1] (a directed graph that does not contain cycles and in which a node can be either an operation node or a control node and edges carry data and reflect dependencies between computations).

The CDFG representation of the original description is then used as the input to the behavioural synthesis which performs scheduling (time slot allocation), allocation (finding the minimum required hardware resources) and binding (mapping of operators and variables into the different resources available). The behavioural synthesis targets an architecture consisting of a datapath and a controller, similar to that used in synchronous synthesis but the architecture is constructed entirely from asynchronous handshake components. The final step is the mapping of the generated architecture into a new optimised Haste description. The overall effect is a source-to-source translation of the original description guided by either minimum area (by limiting the available resources) or minimum latency constraints. An interesting feature of this approach is the possibility of
performing constraint-driven automated design-space exploration.

Average area reductions of 30% and average speed-ups of 40% are reported when applied to a series of digital filter designs. However, it is also reported in [75] that “the origin of this large improvement lies in the fact that the source code is written for code maintainability, which is usually far from the optimal execution of operations”. In common with the previous approach, the quality of the input description will influence the results of the optimisations.

## 1.4 Teak

Teak[6] is a data-driven implementation for the Balsa language, which uses a new target component set and synthesis scheme. Teak replaces the data-less activation channel (used to enclose the behaviour of description fragments in handshake circuits synthesis) with separate go and done channels. Control/datapath interactions using components which exploit signal-level event interleaving are replaced by the forking/rendezvous of control and data channels with local handshaking to complete control interactions. This separation of go and done makes Teak much more like the Macromodules system [93] than Handshake Circuits, albeit with more flexibility in the elimination of control channels through merging with data channels. Explicit buffering is used to decouple one component from another and to introduce the desired degree of token storage to enable the circuit to function and, looking beyond this work, to allow more transforming synthesis methods to increase circuit parallelism.

![Figure 1.3: Teak circuit for the 1-place buffer.](image)

Figure 1.3 shows the buffer example in 1.2 constructed from Teak components. Notice that, rather than a composition of enclosing control components the Loop
... end construct has become a loop comprised of a Merge ($M$) to introduce the ‘go’ token, a Join ($J$) to meet incoming data, and a Fork ($F$) to return a token back around the loop after the output command. The aim of Teak is to provide a path for future performance increases in Balsa by exploiting high performance pipelined asynchronous circuit styles. More details on the Teak system will be introduced in chapter 5.

1.5 Aims of this research

The aim of this research is to explore new alternatives to increase the performance of synthesised circuits using the syntax-directed synthesis paradigm. This work targets dual-rail, quasi-delay-insensitive implementation as this is a robust approach that helps to reduce the impact of increasingly difficult timing closure within modern fabrication processes variability [14, 52].

Having a highly expressive, high-level description language like Balsa or Haste can result in naïve, poor performance descriptions for a novice or even a medium-experienced designer due to the directness of the synthesis method. Furthermore, it is always claimed that in this approach, an experienced designer could make performance/power/area trade-offs. This task would be easier if the designer could have some insight of the impact of a particular construct or coding style.

In contrast with other optimisation approaches, the approach used in the first part of this work is to help the designer select a coding style that results in more concurrent, faster implementations, while providing insight about the trade-offs made. The coding techniques presented here could also serve as a source for future optimising compilers. This work also explores further optimisations on circuits synthesised from highly optimised Balsa code and proposes some circuit transformations and new peephole optimisations that help to increase further the benefits of the performance-oriented coding style.

The second part of this work analyses the circuits generated by the new data-driven based Teak synthesis scheme as a more flexible alternative to implement data-driven circuits. A set of optimisations based on circuit transformations and buffering strategies are proposed in order to improve the performance of Teak circuits. These optimisations have been automated and incorporated into the Teak synthesiser.
The increase on performance of the above mentioned techniques and optimisations are demonstrated using substantial Balsa designs written by both novice and experienced users. The examples include a 32-bit RISC processor, a forwarding unit for this processor, a 32-bit Booth’s multiplier, a Viterbi decoder and a wormhole router. Although area and power are not considered as an optimisation target, area/power savings or penalties are shown for the evaluated examples and proposed optimisations.

1.6 Contribution of this research

The research work presented in this thesis contributes to the field of synthesis of asynchronous circuits in several aspects, including:

- An evaluation of the synthesis of high-performance asynchronous systems using the syntax-directed synthesis approach targeting handshake circuits.
- Performance-oriented language techniques that can be used to describe asynchronous systems within a syntax-directed synthesis framework and their evaluation. These techniques can also serve as the basis for developing optimising syntax-directed compilers targeting handshake circuits.
- New performance-oriented handshake circuits peephole optimisations and their evaluation.
- An evaluation of the performance of the new Teak synthesis system and a set of automated circuit optimisation rules that increase the performance of the Teak-generated circuits.
- An automated set of rules to implement latch insertion in the Teak synthesis system.
- The evaluation of a number of substantial examples using the techniques developed during the course of this research and that can be used as reference for future research.

1.7 Thesis organisation

This thesis is organised as follows:
Chapter 2 presents an overview of the asynchronous design methodologies including commonly used handshake protocols, synthesis of QDI datapaths, synthesis of control circuits and major asynchronous synthesis tools.

Chapter 3 presents an introduction to the Balsa Synthesis System and the Balsa language.

Chapter 4 introduces and analyses a set of description-level performance-oriented techniques for the Balsa language, which target handshake circuits synthesis. A number of new peephole optimisations that increase the performance of the synthesised circuits are presented.

Chapter 5 introduces the Teak synthesis system and component set, together with a number of circuit-level optimisations that have been incorporated in the Teak synthesis tools. This chapter also discusses the impact of description-level styles in the performance of the Teak-synthesised circuits.

Chapter 6 introduces a range of latching strategies currently implemented in the Teak system. An analysis on the complexity of the latching strategies and resulting performance is also presented.

Chapter 7 describes a number of design examples that have been developed to evaluate the impact on performance of the techniques proposed in this research. Simulation results of these examples for both Balsa an Teak styles are presented and discussed.

Chapter 8 present the conclusions and summary of this research work and discusses future work.

1.8 Publications

During the course of this research, the author has contributed to the following papers:


627–634, August 2009.


Chapter 2

Background

2.1 Introduction

This chapter presents an overview of basic concepts of asynchronous digital circuits and the most common handshake protocols used in asynchronous design. The chapter also introduces the various delay models used in the design of asynchronous circuits and the concepts of indication, speed-independency, delay insensitivity and quasi-delay insensitivity (QDI). Finally, the chapter presents an overview of the tools and methodologies most commonly used for the synthesis of asynchronous circuits.

2.2 Asynchronous Circuits

Asynchronous systems do not rely on a global clock signal to control communications and event sequencing. Instead, communication between two asynchronous components is implemented as a handshake protocol using handshake signals to request (initiate) the start of an operation and acknowledge (indicate) to determine its completion.

There are several properties of asynchronous circuits that make them attractive to use in large VLSI designs, including:

- *No clock distribution or clock skew problems*: It is well known that distributing a clock across the chip whilst both minimising the area, power used and the skew between clock arrival at different points of the system is one of the major problems in synchronous design. Eliminating the global
clock signal, automatically eliminates these problems.

- **Better modularity and composability:** As communication between modules depends only on the handshake interface compatibility and not on global timing constraints, modules can be reused and composed as long as their interfaces are compatible [66, 71, 8]. This can be very attractive to modern System On Chip (SoC) and reconfigurable processors.

- **Lower electromagnetic interference (EMI):** Synchronous circuits switch at fixed frequencies, generating a spectrum with relatively higher localised energy at multiples of the clock. Asynchronous circuits only switch during information exchange and the local switching frequency is less coherent, generating a broader emissions spectrum [11, 79, 9].

- **Lower power consumption:** In synchronous circuits, the clock global clock forces to switch all latching stages, unless complex clock gating circuitry is added to enable clocking only to stages where useful work is done. In asynchronous circuits, switching occurs only where the circuits are computing, there is no power wastage in unnecessary switching if the circuits are idle [9, 74].

- **Average-case performance:** In a clocked system, the clock period of the system is dictated by the slowest unit, hence the system operates at worst-case. In asynchronous circuits each unit operates at its own speed, giving the possibility of average-case operation [69, 113].

- **Robustness towards variations in supply voltage, temperature and fabrication process parameters:** In asynchronous circuits, the circuit can be insensitive to wires and gates delays, apart from some localised and easy to meet delay assumptions, reducing the effect of variations on the correct operation of the circuit [68, 74]. Variability has become a major issue in modern fabrication technologies and quasi-delay insensitive asynchronous circuits (see section 2.5.3) are being seen as an attractive solution to this problem.

However, asynchronous design has also some disadvantages, which include:

- **Increased area and circuit complexity:** In order to provide the local handshaking, it is necessary to add circuitry to each asynchronous module. The
solutions to this problem translate into area, power and performance overheads. In contrast, synchronous circuits just use the global clock as the communications control.

- **Lack of design tools:** Modern VLSI circuits cannot be designed without the support of CAD tools for the synthesis, simulation, testing and validation processes. Synchronous design is a mature methodology fully supported by industry CAD tools which have little or no support for asynchronous methodologies. Only recently has TiDE™ [23], a fully-automated commercial tool for asynchronous design with capabilities similar to those present in synchronous tools, been made available. There are also some academic tools like Balsa (freely available) and CAST (not available outside Caltech), but in general they are still far from the maturity and industrial acceptance of today’s synchronous tools.

- **Learning curve:** Designers used to thinking “synchronously” will need to learn an arguably more difficult design methodology in order to exploit the benefits of asynchronous design. The lockstep, deterministic behaviour of synchronous designs is simpler to understand than the concurrent, non-deterministic behaviour of true asynchronous circuits.

### 2.3 Handshake protocols and data encoding

In circuits that communicate using handshake channels (composed of handshake signals), the unit that initiates (requests) the communication is called the *active* party and the unit that responds is referred to as the *passive* party. If, as in figure 2.1(a), the sender of data is the active party the channel is called a *push channel*. In figure 2.1(b) it is the receiver who initiates the communication and this channel is called a *pull channel*. In abstract diagrams, it is common to identify the active end of a channel using a black dot.

There are several common asynchronous handshake protocols named according to the encoding used for handshake and data signals. This section describes the most common of them.
Section 2.3  Handshake protocols and data encoding

2.3.1 Bundled-data protocols

In bundled-data protocols there are separate wires for req and ack signals which are bundled with binary data wires to form the channel, as shown in figure 2.1. Because in this protocol data is encoded using one wire per bit, it is also called single-rail.

Two-phase bundled-data

Figure 2.2 shows a timing diagram of a two-phase bundled-data handshake protocol. The active party initiates the handshaking phase by transitioning the req signal. The other party terminates the handshake by transitioning the ack signal and takes the channel to the quiescent (idle) phase and data becomes invalid until a new request is generated by the active party. In the figure, invalid data is shown as hashed lines and implicit signal causality is shown with dashed lines. This implicit signal ordering must be enforced in bundled-data circuits by using delay matching. Two-phase protocols are also known as Non-Return-to-Zero (NRZ) protocols.
Two-phase protocols are very efficient in time because there are no redundant phases, but normally require more complex circuits to implement than the four-phase protocol described in next section. The overhead due to the circuits’ complexity often reduces the advantages of not having redundant phases.

**Four-phase bundled-data**

In four-phase protocols, the request and acknowledge signals are level-encoded. Figure 2.3(a) shows a timing diagram for a push channel using a four-phase protocol. In this example, the active party first issues the data and then initiates the handshake by setting the req signal high. The passive acknowledge the data by setting ack high. Upon receiving the acknowledge, the active party returns the req signal to zero. Finally, the receiver detects the return to zero of req and acknowledges this by taking ack low, allowing a new handshake to start.

![Figure 2.3: Four-phase bundled-data protocol.](image)

Due to presence of the return to zero phases these protocols are also known as Return-To-Zero (RTZ) protocols. Depending on the interval that valid data is available there are a number of different data-valid conventions for this protocol [83]. Figure 2.4 shows the data-valid schemes for a push channel. In the figure, the early data scheme uses req ↑ as the data validity event and ack ↑ as the data release event. The broad scheme uses req ↑ as data-valid and ack ↓ as data-release. In the late scheme, req ↓ is the data-valid signal and ack ↓ is the data-release.

For a pull channel, figure 2.5 shows that the early data scheme uses ack ↑ as the data validity signal and req ↓ as the data release signal. The broad scheme uses ack ↑ as data-valid and req ↑ (of the next handshake) as data-release. In the late scheme, ack ↓ is the data-valid signal and req ↑ of the next handshake is the data-release. Analysis and comments on the advantages and disadvantages of each of these schemes when used in real systems can be found in [83, 5, 17].
Section 2.3 Handshake protocols and data encoding

Figure 2.4: Four-phase data-validity schemes for a push channel.

Figure 2.5: Four-phase data-validity schemes for a pull channel.
Compared to two-phase bundle-data protocols, four-phase bundled-data protocols have the advantage of using simpler circuits which result in smaller and faster designs, despite requiring more transitions per handshake (which results in more energy consumption).

**Delay matching**

Bundled-data protocols rely on the timing assumption that the order of events in the sender is preserved at the receiver. For instance, in a push channel data must always be valid before $\text{req} \uparrow$. Delays in control and data wires must be matched adequately to make sure that the order of events is preserved at the sender and receiver ends. For instance, in the protocol diagram of figure 2.3, valid data must precede the $\text{req}$ signal in order to guarantee correct operation. This implicit causality is showed with dotted lines in figures 2.2 and 2.3.

A physical implementation of a circuit that uses these protocols must take this into account to avoid operational failures. Controlled placement and routing of wires, buffer insertion to adjust delays, and use of safety margins at the receiver’s end, are possible solutions to this problem. These timing closure problems are similar to those in synchronous circuits, making bundled-data protocols unattractive to use with deep sub-micron fabrication processes affected by large variability in the parameters of the transistors. An alternative to these is to use a more robust class of protocols that are insensitive to wire delays, such as the dual-rail protocols.

### 2.3.2 Dual-rail protocols

These protocols make use of the dual-rail code to transmit both data and data validity indication on the same set of wires, eliminating the timing assumptions. The dual-rail code is a member of the family of delay-insensitive codes [110]. This encoding method allows a reliable communication between two parties regardless of the delay in the wires.

**Dual-rail code**

In a dual-rail code the data is encoded using two wires per bit, $d.t$ for signalling a logic 1 (true) and $d.f$ for signalling a logic 0 (false). The pair of wires \{d.t, d.f\} form a code whose codewords are shown in table 2.1.
Table 2.1: Dual-rail encoding for 1-bit

<table>
<thead>
<tr>
<th>d.t</th>
<th>d.f</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Empty</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Valid &quot;0&quot;</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Valid &quot;1&quot;</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Not used</td>
</tr>
</tbody>
</table>

This encoding scheme can be easily extended to an \( n \)-bit channel. An \( n \)-bit channel is formed by concatenating \( n \) bits coded in dual-rail as above. For a codeword to be valid, every pair of wires must hold a valid code. Similarly, the empty codeword (also referred to as spacer or NULL) occurs when all bit pairs contain the empty code. In this way, when data changes from empty to valid (or vice versa) no intermediate value is valid. This property makes dual-rail encoding a more robust option that helps to reduce the impact of the timing closure problem caused by process variability, despite the fact of using more wires. The price to be paid for this advantage is some extra complexity, area, energy and performance penalties (see section 2.5.3).

The four-phase dual-rail protocol

In this protocol either the request signal and data (push channel) or the acknowledge signal and data (pull channel) are encoded together using the dual-rail code. Figure 2.6 shows a 1-bit push channel using the four-phase dual-rail protocol.

Figure 2.6: Four-phase dual-rail protocol. (a) push channel, (b) timing diagram.

Assuming that initially all signals are low, request is indicated by issuing a valid codeword on the data wires. Before another request can be made, the data wires must assume the empty value. The receiver identifies that data is valid when all bit pairs have become valid, then reads the data and issues \( \text{ack} \uparrow \). The sender detects the acknowledgement and changes the bits to the empty state.
The receiver then identifies when all the bits have become empty and responds with $\text{ack} \downarrow$, allowing the sender initiate a new handshake. Figure 2.7 shows a simplified timing diagram of this operation for an $n$-bit push channel.

![Timing Diagram](image)

Figure 2.7: $n$-bit four-phase dual-rail protocol in a push channel.

**The Two-phase dual-rail protocol**

This protocol also uses two wires per bit but the information is encoded as transitions instead of logic levels. On an $n$-bit channel, a new codeword is received when exactly one wire per bit has made a transition. In this case there is no empty value: a valid codeword is acknowledged and the sender can change one wire per bit again to send another codeword. Figure 2.8(b) shows a timing diagram of a 2-bit push channel using this protocol.

**2.4 Operation modes**

Operation modes specify the restrictions the circuit is subject to when communicating with the environment in order to operate correctly. The most common operation modes for asynchronous circuits are described below.
2.4.1 Fundamental Mode Circuits

Circuits having this model are also called *Huffman circuits* after D.A. Huffman, who was the developer of many theoretical concepts about these circuits. The design method of fundamental mode circuits is similar to the method used for designing synchronous circuits (finite state machine approach). However, as there is no clock to indicate when the signals are valid, the following constraints to the environment apply:

i. only one external input can change at a time.

ii. the environment must wait until the whole circuit settles into a stable state (as a result of a previous input change) before changing one of the inputs.

These strong restrictions help to make the design process easier at the expense of increasing the response time. This method is not practical for complex designs with a large number of state variables due to the exponential increase in the number of possible states.
2.4.2 Burst-Mode circuits

This model was developed by Nowick, Youn and Dill [77, 78, 117]. The model relaxes the restriction of fundamental mode by allowing a group of inputs (input burst) to change in order to move from one state into another. The following are the restrictions used in the burst-mode model:

i. the inputs in a burst are allowed to change in any order but the machine will not react until the entire group of inputs has changed.

ii. after the burst has occurred, the machine generates the specified output burst.

iii. new burst is allowed only after the machine has completely stabilised after reacting to the previous input burst.

Several tools exist to synthesise circuits using burst-mode. Minimalist [34], developed at Columbia University, is one of the more sophisticated examples. Chelcea et al. [19] developed a burst-mode oriented back-end for the Balsa Synthesis System.

2.4.3 Input-output mode

In this model, the environment cannot excite a circuit until it has responded to the previous excitation by changing the value of the output. Note that no assumption is made with respect to the settling of the internal signals. The environment is also allowed to change at any time the values of inputs that do not excite the circuit. The implied causality of the input and output transitions results in more relaxed constraints on the environment connected to input-output mode circuits, but also in more complex interfaces. Different synthesis tools are based on input-output mode, making use of Petri-nets (see section 2.6.1) techniques to facilitate the modelling of circuit interfaces.

2.5 Delay models

Together with the operation mode, in the design of asynchronous circuits some timing assumptions are used, generating a number of delay models. These assumptions allow simplifications to the modelling of the systems. Delay models fall into two main categories: bounded-delay and unbounded-delay.
In a bounded-delay model, the propagation delay of circuit components and wires is bounded. Bounded-delay models are used in the datapath of bundled-data handshaking, and in fundamental mode and burst-mode circuits. Synchronous circuits also make use of a bounded-delay assumption because the maximum delay cannot exceed the length of the clock period.

An unbounded-delay model circuit, the propagation delay of all or some of the circuit components is unbounded. The most common unbounded-delay models include *Speed-Independent*, Delay-Insensitive and Quasi-Delay-Insensitive.

### 2.5.1 Speed-independent (SI) circuits

The *speed-independent* model is based on the theory developed by David Muller [71]. A circuit that is speed-independent assumes positive, unbounded delays for the elements of the circuit (gates) and zero or negligible delay in the wires. In this model, gates are modelled as Boolean operators and, at any given time, each gate of the circuit can be in one of two states:

- stable: The output of the gate is consistent with the value implied by the values of its inputs; its “next output” is the same as its “current output”.

- excited: The inputs of the gate have changed but the corresponding output change is about to occur; its “next output” is different from its “current output”.

When an excited gate finally changes its output after some arbitrary delay and becomes stable, the gate “fires”. This in turn may excite other gates which will eventually fire and so on. The requirements for a circuit designed in this way are that once excited, a gate must fire and remain in that state until its inputs change again. This removes any hazards and guarantees monotonic transitions. Modelling SI circuits requires a state variable for each node of the circuit making the space state very large even for small circuits. Some of the synthesis techniques for SI make use of Signal Transition Graphs (STGs – see section 2.6.1) as an efficient way of representing all possible firing sequences.

### 2.5.2 Delay-insensitive (DI) circuits

In *delay-insensitive* circuits all wires and circuit elements can have positive, unbounded delay. With this assumption, an element that receives an input signal is
forced to indicate (acknowledge) to the sender when it has received the information. This is known as the principle of acknowledgement [91]. No new changes can occur at the input before receiving the acknowledge signal.

Consider for instance a two-input AND gate: If both inputs are high, the output is high and, in this case, a change in any of the inputs will generate a change in the output (the output acknowledges (or indicates) the change at the input). In the case when both inputs were low, a change in any input would not be indicated by the output. A similar situation will occur with other input combinations or by using a different basic gate. This analysis can be easily extended to any basic gate with \( n \)-inputs and a single output (\( n \geq 2 \)).

The DI model is a very robust model, however, it has limitations if applied to general circuit design due to its heavy restrictions. It is trivial to show that the basic single output gates AND, NAND, OR, NOR or XOR cannot indicate all the possible transitions that can occur at their inputs. For this reason, they cannot be used to build a DI circuit.

The only \( n \)-input, single-output gate that can be safely used in DI circuit must be one that only allows transitions on all of its inputs before generating a new transition on its output. This class of gate is called the Muller C-element [71]. Due to this restriction, the class of delay-insensitive circuits happens to be very limited. It has been demonstrated that only circuits composed of C-elements and inverters can be delay insensitive [67]. Figure 2.9 shows the symbol and the specification for a two-input C-element.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 2.9:** The Muller C-element.

### 2.5.3 Quasi-delay insensitive (QDI) circuits

This model uses the DI assumptions with the addition of isochronic forks [65]. Isochronic forks are forking wires where the difference in delays between the destinations is negligible. This allows a signal that is routed to different places to be safely acknowledged by only one of the ends, simplifying the design of the
circuits. Using this restriction, a QDI circuit is equivalent to an SI circuit if the wire delays are lumped into the unbounded-delay gates.

It is possible to extend the isochronic fork assumption to the output of the gates driven by the fork (*extended isochronic fork*) [109]. This assumption can be extended through more than one level of gates at the cost of making the circuit less robust. The type of circuits that uses the extended isochronic fork are referred to as Q^nDI.

## 2.6 Asynchronous synthesis

With the level of complexity in today’s designs, high-level modelling and synthesis is a necessary requirement. The use of of a high-level based synthesis can also reduce the design time compared to full-custom, hand-made designs. Asynchronous designers have today a set of automated and partially-automated synthesis tools available that allow the description and synthesis of complete systems including control and datapath elements (also referred to as functional blocks). This section briefly introduces some of the most popular approaches and synthesis tools available for the design of asynchronous circuits.

### 2.6.1 Synthesis of SI control circuits

Control circuits are required within an asynchronous environment in order to generate the events that guarantee the correct sequence of operation for other components. In the SI approach the designer must specify all possible sequences of input and output signal transitions that describe the restrictions on the circuit environment. This specification can be done using *Signal Transition Graphs* (STGs) [20], [21]. STGs belong to the family of models called *Petri Nets* [72]. A brief introduction to Petri Nets and STGs are given below, followed by an introduction to the STG-based synthesis tool Petrify [25].

**Petri nets**

A Petri net is a graph composed of directed arcs and two types of nodes: *transitions* and *places*. Arcs can only run between places and transitions. The places from which an arc runs to a transition are called the *input places* of the transition; the places to which arcs run from a transition are called the *output places*
of the transition. Places may contain any number of tokens. A distribution of tokens over the places of a net is called a marking. A Petri Net model can be “executed by” firing transitions. A transition is enabled to fire if there are tokens in all of its input places. When a transition fires, it consumes the tokens from its input places, performs some processing task, and places a specified number of tokens into each of its output places. This is done atomically, in one single non-pre-emptive step. During the execution, multiple transitions can be enabled and they will fire at any time, and it is also possible for an enabled transition not to fire at all. This non-deterministic behaviour makes Petri nets to be well suited for modelling the concurrent behaviour of distributed systems.

Figure 2.10: A T-element connected to left and right “well behaved” environments and its specification in the form of a timing diagram, a Petri net and an STG.
As an example of Petri-nets usage, consider the specification of an *autosweeping module* [73], (most commonly called a *T-element*) shown in figure 2.10. T-elements are used in handshake control circuits that implement *handshake enclosure* (the handshake of the right side is enclosed within the handshake of the left side), parallelisation and sequencing of operations, with concurrent RTZ phases [73, 89, 57, 82].

In the Petri net of figure 2.10, signal transitions are represented by horizontal bars and places with circles. Tokens are represented by black dots inside a place. A transition from 0 to 1 in signal $x$ is represented by $x^+$; similarly, a transition from 1 to 0 is represented by $x^-$. The graph is marked with a token in the input place of the $Ir^+$ transition. The T-element is connected to a “well behaved” dummy environments on the left and right hand sides that allow changes on the inputs only after the T-element has changed its output responding to a previous request. In this situation, the $Ir^+$, $Or^+$ $Oa^+$ transitions must fire in sequence.

After the firing of $Oa^+$, a token is placed in the input place of $Ia^+$ transition and another token is placed in the input place of the $Or^-$ transition, allowing these transitions to fire and so on. Note that the execution of $Ia^+$ followed by $Ir^-$ is allowed to occur concurrently with the execution of $Or^-$ followed by $Oa^-$. 

**Signal Transition Graphs**

An STG is a Petri net with the following characteristics [91], [48]:

i. **Input free choice**: The selection among alternatives must only be controlled by mutually exclusive inputs.

ii. **1-bounded**: There must never be more than one token on an arc.

iii. **Liveness**: the STG must be free from deadlocks. That means that from every reachable marking, every transition can eventually be fired.

iv. **Consistent state assignment**: The transitions of a signal must strictly alternate between + and −.

v. **Persistence**: For all arcs $a^* \rightarrow b^*$ in the STG (where $t^*$ means transition $t^+$ or $t^-$), there must be other arcs that insure that $b^*$ fires before the opposite transition of $a^*$ occurs.
vi. **Complete state coding (CSC):** Two or more different markings of the STG must not have the same signal values (i.e., correspond to the same state). If this is not the case, it is necessary to introduce extra variables such that different markings corresponds to different states.

STGs represent synthesisable circuit implementations. Figure 2.10 shows the STG specification of a T-element equivalent to the Petri-net at its left. Compared to a Petri net diagram, in an STG, labelled transitions are replaced with its label, and places with a single input and output are omitted. As shown in the figure, the tokens in these omitted places are placed on the corresponding arcs. Transitions corresponding to inputs are distinguished by underlines. In the example, the original Petri-net is 1-bounded with no choice, hence a circuit implementation may be synthesised.

**Petrify**

Petrify [25] is a public domain synthesis tool for manipulating Petri nets and for synthesising SI control circuits from STG specifications. STG descriptions for Petrify are written in plain text. Petrify can solve CSC violations by automatically inserting state variables. From the STG specification Petrify can produce either a complex-gate circuit, a generalised C-element circuit or map the circuit onto a gate library supplied by the user.

### 2.6.2 Communicating Hardware Processes (CHP) and the Caltech Asynchronous Synthesis Tool (CAST)

The CHP synthesis system was developed at Caltech by A.J. Martin[66]. CHP language has a syntax similar to the concurrent programming language Communicating Sequential Processes (CSP) [50], using various special symbols. Design flow in CHP starts with a specification of the system in the CHP language. The first step is to reduce complex control structures found in the specification into combinations of simple processes. In a second step, these processes are then expanded into four-phase handshake protocols (handshake expansion - HSE) to convert them into sets of transitions. In order to distinguish ambiguous states, reshuffling and variable insertion is performed and, finally, production rule sets (PRS) are generated, which can be mapped into a physical circuit realisation, targeting a specific building block called PHCB (precharge half-buffer). Many
of the above steps require user intervention and guidance and this may have a significant impact in the performance and area of the synthesised circuit.

The Caltech Asynchronous Synthesis Tool (CAST) is a suite of design tools based in CHP that provides modules to refine CHP descriptions, translate CHP descriptions into HSE, HSE into PRS and mapping PRS into PHCB circuit networks. CAST has been used to synthesise complex chips, including the MIPS R3000 [69] and the Luthonium 18 (a 8051 clone) [70], but these rely on significant manual intervention in the synthesis flow to achieve the most effective program transformations. Another issue is that the automatic program transformations used in CAST are not behaviour preserving and are only correct for designs that meet particular requirements, which may not be straightforward to an inexperienced designer. CAST tools are currently only available internally at Caltech.

**TIMA Asynchronous digital systems Synthesis Tool (TAST)**

TAST (Tool for Asynchronous circuits SynThesis)[104] is a compiler/synthesis tool that synthesises asynchronous systems from a specification written in CHP. The compiler analyses the given specification and transforms it into an internal format based on Petri Nets and Data Flow Graphs. From this intermediate form the user can generate:

- a functional VHDL description of the model for simulation purposes.
- an RTL VHDL description, which can be used to target ASICs or FPGAs technologies by means of standard CAD tools.
- an asynchronous circuit. However, in this case, CHP descriptions must be written using the *Data Transfer Level* (DTL) style subject to certain rules to ensure a correct mapping [28]. If the mapping is possible, a gate netlist is produced. The gate netlist can either be simulated using standard CAD tools or used to implement the circuit through a technology mapping process that requires a specialised TAST cell library.
2.6.3 Macromodular based synthesis

The *Macromodular* methodologies make use of pre-designed blocks (the *macromodules*) that communicate asynchronously using handshake channels. Macromodules were first proposed by Clark in Washington University [22] during the late 1960’s. More recently, Brunvand introduced a macromodular synthesis system [18], making use of the channel-based, CSP-like programming language Occam [63] to describe circuits. Descriptions are automatically synthesised into compositions of control, variable read/write and datapath macrocells implemented with 2-phase signalling with bundled data. Plana [87] describes a system of macromodules to construct asynchronous circuits that communicates using pulse-mode [56] handshaking. The macromodules are described using petri-nets with signal pulse labelled transitions. The examples of pulse-mode circuits given by Plana were constructed by hand but they are specified using a pseudo-code that could be the basis for a synthesis system.

The handshake circuits paradigm proposed by van Berkel for use in the Tangram tool is another approach to macromodular synthesis. Tangram TiDE and Balsa synthesis tools are all based in the transparent compilation and the handshake circuits paradigm (c.f 1). They use CSP-like description language but with a syntax more similar to traditional programming languages than CHP. TiDE and Balsa are currently the major fully-automated synthesis systems for asynchronous design.

Due to its relevance to the work in this thesis, a more complete introduction to the Balsa synthesis system will be presented in chapter 3. Details of Teak[6], a novel data-flow implementation for the Balsa language, will be introduced in chapter 5.

2.6.4 Desynchronisation methods

Desynchronisation methods rely on the use of a synchronous design methodology and commercial CAD tools and then convert the resulting circuits into asynchronous designs. The flow described in [26] targets bundled data implementation whereas the one presented in [61] targets QDI circuits and uses the $NCL_X$ approach. In these approaches, synchronous CAD tools are used for datapath synthesis and asynchronous control synthesis tools are used to produce controllers that replace the global clock.
The advantages of this approach are that designers need little specialist knowledge of asynchronous techniques and the synthesis uses well-known commercial tools. However, as the design is targeted at a synchronous implementation, some potential advantages of asynchronous techniques are not exploited, such as: (a) the fine-grained concurrency that can be possible in asynchronous design, (b) the possibility for asynchronous designs to use data-dependent delays instead of the worst-case delays used in synchronous design. Two popular approaches used in the desynchronisation method are briefly described below.

**Null convention logic (NCL)**

In order to reduce the complexity in QDI functional blocks, Theseus Logic Inc. [32] proposed the Null Convention Logic approach. In NCL data is DI encoded (using dual-rail encoding or other 1 of N code) and uses a 4-phase protocol. Data changes from the empty (NULL) value to a valid codeword (Data) in the set phase and then back to NULL in the reset (RTZ) phase. To implement this operation, NCL makes use of m-of-n threshold gates with hysteresis. An hysteresis threshold gate is a logic gate which will set its output high when the sum of the weights on the inputs exceeds a fixed gate threshold (m inputs for an m-of-n threshold gate). The output of the gate will return to low when all inputs become low. Notice that, in applying this idea, a C-element is an n-of-n hysteresis threshold gate and an OR gate is a 1-of-n threshold gate.

Synthesis of NCL circuits from logical descriptions can be performed by mapping two level Boolean implementations of those functions into minterms implemented with C-elements and OR gates to implement the AND and OR levels using *Delay Insensitive Minterm Synthesis* (DIMS) [71]. The C-elements and OR gates of DIMS can then be mapped onto their threshold gate analogues. Simple hysteresis threshold gates can then be optimised into threshold gates with more complicated input weightings. This is the key part of the synthesis process, however, these optimisations are not easily automated.

Figure 2.11 shows an optimised NCL implementation of a dual-rail 1-bit adder. In this figure, the number inside the gate corresponds to the value of the threshold.
NCL with explicit completeness (NCL_X)

In order to reduce the area and energy of NCL circuits, Kondratyev and Lwin [61] proposed NCL_X, a different approach based on the idea of “separate implementations for functionality and delay insensitivity” allowing independent optimisations of each. In NCL_X, after obtaining the optimised Boolean implementation of the functional block, the circuit is mapped into unate gates (gates that implement a positively unate function - all inputs in such functions are used without inversions). This is done by using two different variables, x.t and x.f for direct and inverse signals of x. The obtained network implements rail “1” (t) of a dual-rail circuit for the functional block. The dual-rail expansion is completed by creating a corresponding dual gate in the rail “0” (f) network for each gate in the rail “1” network. Finally, delay insensitivity is achieved by providing local completion detectors (OR gates) on each pair of dual gates and connecting them into a multi-input C-element to generate the done signal. In NCL_X for each dual-rail primary input of a block, there must be a signal go that indicates the state of the input (go = 0 → NULL, go = 1 → Valid).

The claimed benefits of both NCL and NCL_X methodology is that they can make use of existing electronic design automation (EDA) tools developed for synchronous synthesis. In [61] it was reported that, compared to NCL, NCL_X circuits reduce significantly the area overhead, are faster and have a similar power consumption. It was also noted there that compared to synchronous circuits, NCL_X are 2 to 2.5 times larger and consume more energy, with the benefits being on lower EMI and improved security and reliability.

Figure 2.11: A dual-rail full adder using NCL gates.
2.7 Summary

Asynchronous circuits have some attractive advantages over their synchronous counterparts. By eliminating the clock, some major problems associated with it could be alleviated. In particular, reduced EMI and robustness towards fabrication process variability are nowadays their most attractive characteristics.

Delay-insensitive encoding and quasi-delay insensitive asynchronous circuits have been proposed as an alternative to alleviate the complex problem of timing closure in modern sub-micron fabrication technologies. However, their robustness comes at the price of more complex, slower and expensive circuits when compared to synchronous implementation. Some approaches towards the synthesis of QDI datapath circuits have been proposed with different complexity/robustness trade-offs to reduce the inherent penalties of the QDI approach.

Research in asynchronous synthesis has resulted in the development of various synthesis techniques and tools available for the design of large scale asynchronous circuits, some based in pure asynchronous methodologies such as the various macromodular methods (including the handshake circuits approach used in Balsa) and other recent approaches based in synchronous methodologies plus a “desynchronisation” process.
3.1 Introduction

This chapter presents a brief introduction to the Balsa synthesis system and the Balsa language, which is also the input language for the Teak synthesis system described in chapter 5. Some small examples are included to highlight the directness of the compilation scheme and the most common input and control structures used in Balsa circuits.

3.2 The Balsa synthesis system

Balsa is the name for both the framework for synthesising asynchronous circuits and the language used to describe such systems. Balsa uses the syntax-directed compilation approach to generate handshake circuits from a description written in the Balsa language.

Originally introduced by van Berkel [108], a handshake circuit is a communicating network of handshake components connected point-to-point using handshake channels (see 2.3). Each channel connects exactly one passive port of a handshake component to an active port of another handshake component. As mentioned in section 2.3, an active port is a port that initiates the communication by sending a request signal to a passive port. When ready, the passive port will respond with the acknowledge signal. The handshake can involve the transfer of data or simply synchronisation (control) using a dataless sync channel.
(a channel conveying the request and acknowledge signals only).

### 3.2.1 Balsa design flow

As shown in figure 3.1, in order to synthesise a circuit from its description the Balsa system uses a compiler (balsa-c) that generates a handshake circuit described in an intermediate netlist format (Breeze). A Breeze description can be processed using balsa-netlist to produce a structural Verilog netlist of the circuit for a chosen target cell library, asynchronous protocol implementation style and data encoding described in the selected back-end library. This file can then be processed using commercial layout tools for simulation, validation, and fabrication.

The system also features a behavioural simulation tool, breeze-sim, that works at the handshake component level, and an area cost estimator: breeze-cost. From version 3.5, Balsa includes balsa-mgr, a graphical front-end that provides project management facilities. More detailed information on Balsa and the Balsa language can be found in the Balsa Manual [30].

The Balsa synthesis system has been used successfully to synthesise the 32-channel DMA controller for the DRACO chip [40], an asynchronous MIPS processor [118], and the G3Card smartcard System-on-Chip. Those designs together with more recent work [89, 85] have demonstrated the potential of Balsa and its synthesis approach to generate efficient asynchronous systems for complex, real world applications.

### 3.3 The Balsa language

This section briefly introduces the Balsa language. Details of the language and compilation scheme not relevant to this work have been omitted. Detailed information on the language features and a complete language syntax reference can be found in the Balsa Manual [30]. Extensive details on the compilation process and handshake circuits used with Balsa can be found in [108, 29, 89, 82]. The description of the language features are accompanied with example code and, where relevant, the resulting handshake circuit generated by the compiler.
3.3.1 The structure of a Balsa description

A description in Balsa is composed of one or more files that have a structure similar to the example shown in figure 3.2. In this figure the main parts of a Balsa description are indicated. Balsa descriptions are divided into a number of procedures. Each procedure has an implicit activation port (that activates the circuits within the procedure) and any number of input, output or sync (dataless) ports that must be explicitly declared. Within the procedure, channels and variables can be declared with local scope. Channels are used to communicate between procedures or between concurrent actions (commands). Each channel
must have at least one source (producer) and one sink (consumer). Variables are used as temporary storage for values. Writes and reads on the same variable must be sequenced.

--- This is a comment in Balsa.
Block comments use the (-- COMMENT --) pair
Extension '.breeze' assumed

import [Path.And.Name]

-- Global declarations (types, constants & procedures)
--- Examples of type declarations
type word is 16 bits -- unsigned type
type sword is 16 signed bits -- signed type
--
-- Other global declarations ...

-- A procedure declaration
procedure exampleProc
-- port declarations are separated by ';' (1
input a : someType;
output out : sword;
sync z
) is
-- local declarations (types, constants, procedures)
-- local declarations (variables & internal channels)
variable var : someType
channel c : otherType
begin -- exampleProc body
(--
Commands and procedures composed with "||" or ";" operators
--) end -- end of exampleProc body

Figure 3.2: The structure of a Balsa description.

Procedures consist of one or more commands composed using control operators. A command may consist of:

- a basic read or write action on a channel or variable.
- an iteration construct.
- a conditional construct.
- another instantiation of a procedure.
- a sequential or parallel composition of commands.
A command makes use of channels to communicate internally with other local commands using the declared channels or externally using the procedure’s ports. Channels and variables can be read or write, input ports are read-only channels and output ports are write-only channels.

Balsa supports modular compilation: a description can be divided into multiple files which are included using import statements. These must always be located at the beginning of the file preceding any other declaration. The files to be imported must be pre-compiled handshake circuits in Breeze format. Constants and user-defined data types can be declared afterwards inside or outside the procedures. In order to use any type/constant/procedure in Balsa, this has to be declared previously within the file or imported files, as Balsa follows the same “declare before use” rule of C and Modula [30].

3.3.2 Data Types
Balsa is a strongly typed language with data types based on bit vectors. Results of expressions must be guaranteed to fit within the range of the underlying vector representation [30]. Balsa supports global and local type and constant declarations. Balsa supports the following data types:

**Numeric types**

Bit vectors of width bits that can be signed or unsigned. Examples:

- `type word is 16 bits` (unsigned type with range \([0, 2^{16} - 1]\))
- `type sword is 16 signed bits` (unsigned type with range \([-2^{15}, 2^{15} - 1]\))

**Enumerated types**

This type consists of named numeric values. The numeric values are given incrementally starting at zero, with explicit values resetting the counter, for example:

```plaintext
  type MyEnum is enumeration
    ZERO, ONE, FIVE=5, OTHER
  end
```
In the above example the following values are assigned: ZERO=0, ONE=1, FIVE=5, OTHER=6. The values require 3 bits, hence this type is 3 bits wide. Note that values 2, 3, 4 and 7 are not bound to names.

Record types

Bit-wise composition of named elements of possibly different (pre-declared) types. for example:

```plaintext
    type SignMagnitude is record
        Magnitude : MyEnum;
        Sign      : bit;
    end
```

In this example, a value of type SignMagnitude will have a width of 4 bits with the Magnitude field occupying the first three least significant positions and Sign occupying the most significant position. Referring to a field within a record is accomplished with the usual dot notation.

Array types

Numerically (or enumerated) indexed compositions of values of the same type. For example:

```plaintext
    type RegisterBank : array 0..15 of word
```

3.3.3 Basic transfer commands

Balsa provides two basic commands to transfer information: channel read and channel write; these generate handshake data transfers within the involved channels.

a \(\rightarrow\) b reads channel a and writes channel to b. a can be either an input port or an internal channel. b can be either a variable, an internal channel or an output port.

d \(\leftarrow\) c transfers the value of variable/expression c to the output port or internal channel named d.
3.3.4 Dataless handshakes

\texttt{sync a} generate a handshake in the dataless channel \texttt{a}. Further actions can only occur after the handshake on \texttt{a} completes.

3.3.5 Variable assignment

\texttt{var := expression} transfers the result of \texttt{expression} to the variable \texttt{var}. Balsa allows variable auto-assignment, where an expression includes the target variable. However, the resulting circuit will contain an invisible auxiliary variable, whose contents will be written back to the programmer’s variable after being assigned the result of \( f(x) \). The type of a result must agree with that of the variable to be assigned. In cases when these types may differ, the user can truncate/expand the width of the result by explicit casting. For instance, if \( x \) is a variable of type byte, the following statement is invalid:

\[
\texttt{x := x + 1} \quad \text{-- invalid, result may require an extra bit}
\]

The correct statement should look like:

\[
\texttt{x := (x + 1 as byte)} \quad \text{-- the result is truncated to 1 byte.}
\]

3.3.6 Control operators

Balsa has two control operators to form composed commands: \texttt{Concur \|} and \texttt{Sequence ;}.

\texttt{<command1> \| <command2>} composes the two commands so that they operate concurrently. However, both commands must terminate before the composed command is completed. \texttt{Concur} generates a \textit{rendezvous} point when both commands complete.

\texttt{<command1> ; <command2>} sequences the execution of the two commands: the first must terminate before the second can proceed.

The \texttt{\|} operator has a higher precedence than \texttt{;}. This precedence can be overridden by creating groups of commands using either square brackets ( \texttt{[ ]} ) or the pair of keywords \texttt{begin ... end} to enclose a command. For instance:
-- 'x' is written first, sequenced by the concurrent write
-- of variable 'y' and channel 'z'
x := 10 ; y := 20 || z <- 30
-- here, 'x' is written first sequenced by the writing of 'y'.
-- These two actions are concurrent with the writing of 'z'
[ x := 10 ; y := 20 ] || z <- 30

When composing commands, care must be taken to avoid introducing dependencies that may lead to a deadlock. As a very simple example, consider the program in figure 3.3. The program consist of an infinite repetition of two compound commands (lines 9 and 11), which in turn are composed with the || operator, effectively creating two execution threads.

1 procedure deadlock
2 ( 3 output out : byte 4 ) is 5 channel a, b : byte 6 variable v1, v2 : byte 7 begin 8 loop 9 [ a -> v1 ; out <- v1 || b <- v1 ] -- command 2 10 [ ] 11 [b -> v2 ; a <- (v2 + v2 as byte) ] -- command 1 12 end 13 end

Figure 3.3: Example of deadlocking code.

Upon activation, a transfer from channel a into variable v1 is activated (first action in line 9). Concurrently, in the second group, a transfer from channel b into variable v2 is also activated. However, the circuit deadlocks because the read from channel a can complete only after the write to channel a completes but this last action can only start after the read from channel b completes (and that requires the completion of the read from channel a). In this simple example deadlock is eliminated by swapping the sequenced actions in one of the composed commands.

Invalid compositions

In order to avoid some potential deadlock situations or unsafe operations, the Balsa compiler will fail (and the user will get the relevant feedback on the error) when it encounters the following compositions within a description:

- A write sequenced with a read on the same channel or a read sequenced
with a write on the same channel. These will result in a deadlock because the first action cannot complete until the second action completes and the second action cannot start until the first completes.

- Commands composed with the *Concur* operator that: *(i)* write and read from the same variable, *(ii)* write to the same channel, *(iii)* write to the same variable. Concurrent occurrence of these actions are unsafe and causes malfunction. When any of these conditions occur in a description, the compilation will fail. However, from version 3.5.1 Balsa introduced the experimental “permissive” *Concur* (111) which leaves to the user the responsibility of making sure that those conditions will not actually occur during operation. This operator can be used (with care!) to exploit the designer’s knowledge on the operation of the circuit, as will be described below.

**Continue and halt commands**

The *continue* command is used to implement “no operation” (always acknowledges any activation request it receives). When the execution of a process thread reaches a *halt* command, this thread deadlocks (no further actions occur).

### 3.3.7 Iteration and conditional constructs

An unbounded repetition in Balsa uses the *loop* <command> end construct as shown in the example in figure 1.2(a). Bounded repetitions use the construct:

```plaintext
loop (<command0>) -- command0 is optional
while guard1 then <command1>
  | guard2 then <command2>
  |
  | guardN then <commandN>
end
```

This construct allows the specification of repetitive loops equivalent to *for*, *repeat ... until* and *do ... while* found in other languages. However, Balsa allows the specification of multiple guard conditions. If multiple guards are used, they are evaluated in order. If more than one guard is satisfied, only the command associated with the guard that appears earlier in the list will be activated.
The following are code examples of `loop` constructs:

```
variable x : byte
channel inp, out : byte
-- infinite loop
loop
    inp => x ;
    out <= x
end
--

-- for (x=0; x<10; x++) <command> equivalent
x := 0 ; -- initialisation
loop
    while x < 10 then
        print "value of x is: ", x ;
        x := (x + 1 as byte) -- autoassignment
    end
--

-- repeat <command> until x<10 equivalent
loop
    print "value of x is: ", x ;
    x := (x + 1 as byte) -- autoassignment
while x < 10
end
--

-- multiple guards: 0 to 9 counter with autowrap
-- Note 1: guards are evaluated in order.
-- Note 2: the loop is infinite unless initially x > 9
loop while
    x < 9 then x := (x + 1 as byte)
    | x = 9 then x := 0
end
```

Balsa features the `if ... then ... else` and the `case ... of ... else` constructs for conditional execution. The former can have multiple guards which makes it equivalent to nested `if ... else` statements found in other languages and, similarly, the `else` clause is optional. Multiple guard evaluation is similar to that of the `loop ... while` construct. The syntax of the `if ... then ... else` and the `case ... of ... else` constructs is as follows:
if condition1 then <command1>
| condition2 then <command2>
| ...
| conditionN then <commandN>
else <commandDefault> -- optional clause
end

case expression of
    guardList1 then <command1>
    | guardList2 then <command2>
    | ...
    | guardListN then <commandN>
else <commandDefault> -- optional clause
end

The guardList can be either a single expression of a comma-separated list of expressions whose values must be resolvable at compile time. All guard values must be disjoint from one another. As the reader could easily prove, the if and case constructs can be used to implement equivalent conditional behaviours. The if construct is more flexible as it allows the use of expressions for guards and guards do not need to be disjoint. In the case construct, guards must be disjoint and either explicitly given or written as expressions resolvable at compile time. However, in general, the case construct generates faster circuits (see section 4.5).

### 3.3.8 Data processing operators

Balsa provides basic unary and binary bit-wise logic (not, and, or, xor) and arithmetic operators (+, -), as well as Boolean and comparison operators (=, /=, >, <, >=, <=) to construct expressions. There are not shift operators but these can be implemented with the concatenation (@) and smash (#) operators. A complete table can be found in Appendix A. Other operators like multiply, divide and remainder (*, /, %), log and exponentiation (^) can only be used in constant expressions.

### 3.3.9 Input enclosure

Balsa features two constructs that allow the handshake of one or more input channels to be held open until a command or a group of composed commands complete: (i) passive-input enclosure with choice with the select command and (ii) active-input enclosure with the <channels> -> then <command> end
construct. The “enclosed” commands can read the value of the channels as many times as required (or even not read at all) without the need of variables to hold those values. Within the enclosure construct, enclosed channels act like variables for reading purposes.

Input enclosure can generate area benefits and help to produce simpler descriptions. However, there are performance implications: the tree of handshakes connected to the enclosing inputs cannot themselves complete until the enclosed actions complete. These implications will be discussed in section 4.4.2.

**Passive-input enclosure**

The syntax for this type of enclosure is as follows:

```
select
    groupOfChannels1 then command1
| groupOfChannels2 then command2
  .
  .
  .
| groupOfChannelsN then commandN
end --select
```

The `select` statement allows selection between groups of input channels by waiting for data on any of the groups to arrive. The arrival of data among each group must be guaranteed to be mutually exclusive. This also means that a channel can only be part of exactly one group. The enclosed commands are activated only after all the inputs involved arrive. This type of enclosure generates passive ports for the inputs as opposed to the active-ported circuits that Balsa normally generates.

It is recommended that the use of `select` is restricted to only cases where input choice is genuinely required and that the faster active-input enclosure is used instead in other cases. Another reason for using passive-input enclosure is if the interface of a design requires passive (push) inputs.

**Active-input enclosure**

This enclosure has the following syntax:
Similar to the `select` construct, the enclosed commands are activated only after the arrival of all input channels. In contrast, this type of enclosure does not allow choice and generates active (pull) inputs.

### Eager input enclosures

Balsa also features `eager` variations of the passive and active enclosures using their “banged” variants:

```plaintext
select! channels then command end
channels ->! then command end.
```

In its eager variant, the `select` cannot be used with input choice. As stated previously, in the standard enclosures the activation of the enclosed commands occur only after all of the involved inputs have arrived. In the eager enclosures, the enclosed commands are activated as soon as the control activates the inputs, without waiting for the data to arrive. This has performance benefits, because the control is given a head start, hence reducing the control overhead. However, any enclosed command that does not depend on the arrival of data may occur before the data arrives and, if not used carefully, this could result in incorrect operation.

The eager variants still guarantee that the command will not complete until input data has also completed. Details on the implementation of the eager enclosure construct were introduced in [89].

To illustrate the use and behaviour of both types of enclosure, let us consider the following examples:

```plaintext
a, b -> then
  out1 <- (a + b as byte)
  || out2 <- b
  || out3 <- 10
end -- a, b ->
```

In the previous code active enclosure is used. Writing to the channels `out1`, `out2` and `out3` can only occur after the arrival of inputs `a` and `b`, despite `out2`
being independent of \( a \) and \( \text{out3} \) being independent of both inputs.

\[
\text{a, b ->! then}
\text{  out1 <- (a + b as byte)}
\text{| | out2 <- b}
\text{| | out3 <- 10}
\text{end -- a,b ->}
\]

In this example, eager active enclosure is specified. Here the command that writes to channel \( \text{out3} \) starts as soon as the control reaches the enclosure command, without waiting for the arrival of \( a \) and \( b \). Furthermore, if \( b \) arrives earlier, the command that writes to \( \text{out2} \) starts without waiting for input \( a \). However, all commands will complete only after both inputs complete. An example of the use of eager active inputs that results in incorrect operation will be given in section 3.3.12, example 5.

### 3.3.10 Arbitration

Balsa features the \texttt{arbitrate} command when choice is required among two non-mutually exclusive inputs (or groups of inputs). Its syntax is similar to that of the \texttt{select} command:

\[
\text{arbitrate}
\text{  groupOfChannels1 then command1}
\text{| | groupOfChannels2 then command2}
\text{end}
\]

Upon arrival of every input in one of the groups, the associated command is activated. Similar to the \texttt{select} construct, the command will be enclosed within the handshakes of the inputs and these can be read as described previously. Both of the two groups of inputs may arrive, but the control will be passed only to the command enclosed by the group that arrives first. If the two groups arrive so close in time, in such a way that the first arriving group cannot be discerned, an arbitrary decision is made.

If more than two events require arbitration, an arbiter tree can be constructed using the \texttt{arbitrate} construct. An example of a parameterised arbiter tree can be found in the Balsa Manual.
3.3.11 Permissive Concur

The permissive *Concur (!!!)* permits the parallel composition of the potentially unsafe operations described previously. This relaxation can be used when the designer knows that the unsafe conditions will never occur, leading to either smaller or faster circuits and, in some cases, to more compact descriptions. Consider for instance a situation where two concurrent processes $P1$ and $P2$ write to a common channel $c$. If the operation is such that the writes are guaranteed to be mutually exclusive, there is more than one way to implement the access to channel $c$. The more straightforward implementation would be the use of the *select* command described earlier. Another option would be to use a (previously generated) selection data channel that signals which process is the next to write, and then use it as the guard of a conditional construct that selects the appropriate source to pass to the destination channel. However, because the potential conflict will not occur, $P1$ and $P2$ can be composed using the permissive *Concur* which will allow them to access the common channel. An example that illustrates its use will be given in section 3.3.12, example 6.

3.3.12 Compilation examples

This section presents some simple program examples and their resulting handshake circuits in order to familiarise the reader with the structures generated by the use of different Balsa constructs and their operation. Details of the compilation of a simple 1-place buffer have already been given in section 1.2. A brief description of the handshake components used in the examples can be found in Appendix B. Extensive details on the compilation process can be found in [108, 29, 89, 82].

**Example 1: passive enclosure**

The code for a two-input, uncontrolled multiplexer (merge) is shown in figure 3.4. Inputs $a$ and $b$ must be mutually exclusive. The resulting handshake circuit showing the translation of this construct is shown in figure 3.5. In this and the subsequent handshake circuit figures, regions of different colours show the boundaries of the commands. Control tree elements are embedded in a darker shade of the command area colour. Thick arrow lines connect datapath components and thin lines represent control (dataless) channels. Passive ports are represented by
small unfilled circles, and active ports by small filled circles.

```plaintext
procedure merge2
(
  parameter DataType: type;
  input a, b : DataType;
  output o : DataType
) is
begin
  select
    a then
      o <- a
    | b then
      o <- b
  end -- select
end -- merge2
```

Figure 3.4: An uncontrolled multiplexer (merge).

The DecisionWait component (DW) synchronises the activation signal with one of its inputs coming from the signal outputs of the FalseVariable components (FV), and activates the corresponding decision output. As its name suggests, an FV does not have storage: it simply provides passive read ports and a control output signal (the active port on the top) to indicate arrival/removal of data. The FV activates its signal output as soon as the least significant bit (bit 0) of the data input arrives. The reader can refer to [89] and Appendix C for details on the operation and current implementation of this component.

The outputs of the DW are used to pull data from the selected data channel, using a Transferrer(->) component, through the read port of its associated FalseVariable. Finally, a CallMux (-->) (mixer/merger) component is used to merge the source channels into the output o. In this particular example, everything but
the mixer is overhead, as will be explained later in example 6 with the use of the permissive Concur.

Example 2: active enclosure and operators

The code for a simple adder using active enclosure is shown in figure 3.6. The resulting handshake circuit showing the translation of the various constructs is shown in figure 3.7. Notice that the circuit features pull (active) channels at the I/O data interfaces.

```plaintext
procedure adder
{
  input a, b : dtype;
  output o : dtype;
} is
begin
  a, b -> then
    o <- (a + b as dtype)
  end -- a, b ->
end -- adder

Figure 3.6: The description of a simple two-input adder.

Figure 3.7: Handshake circuit of the adder code in figure 3.6.

On top of figure 3.7, a Fork component is used to fork the activate signal to the two Transferrer components. Upon activation, these transfer the inputs to the two FalseVariable components (FV). The signal outputs of the FVs are connected to a Synch (synchroniser) component, which activates its output when both input signals indicate the arrival of data. The Synch output activates the transfer on channel o through another Transferrer, which pulls the result from the addition operator (+). This pull action results in the reading of both FVs.
Eventually, the environment connected to channel \( o \) will acknowledge the transfer and this is passed to the \textit{Synch}, which in turns indicates this to the \textit{FVs} through their signal port. Upon receiving the acknowledge, the \textit{FVs} acknowledge the inputs and the \textit{Transferrer} components pass it to the forked activation. The activating control will eventually respond initiating the RTZ phase and a set of RTZ events will propagate in similar fashion to what has been described until the four-phase handshakes complete.

**Example 3: conditional execution and active \textit{eager} inputs**

Figure 3.8 shows the code for a circuit that reads input channel \( i \) and, depending on the value of the \( s \) signal, passes the constant 10 or the value of \( i \) to the output channel \( o \). The resulting circuit is shown in figure 3.9.

```plaintext
procedure condInput
(
  input i : byte;
  input s : bit;
  output o : byte
) is
begin
  s, i ->! then
  if s then
    o <- i
  else
    o <- 10
  end -- if s
end -- s, inp ->!
end
```

Figure 3.8: Example of conditional execution.

The description is made using active \textit{eager} inputs, but other descriptions are also possible. Note that on this occasion \textit{activeEagerFalseVariable (aeFV)} components are used. An \textit{aeFV} has an active input port and a \textit{trigger} port to activate it. Unlike a \textit{FV}, its \textit{signal} output activates as soon as the trigger is activated, without waiting for data arrival. For details of its operation and current implementation, please refer to Appendix C.

The \textit{Case (@)} component is essentially a decoder that activates only one of its control outputs at a time depending on the value on its input channel, allowing the transfer of either the value of channel \( i \) (in the bottom \textit{aeFV}) or the value 10 from the \textit{Constant} component. A \textit{CallMux} component is used to merge the
source channels. The *Case* component guarantees the required mutual exclusivity at the inputs of the *CallMux*.

As stated earlier, the use of active eager inputs has the benefit of allowing the control section to proceed without waiting for the data. Thus, when data arrives, control signals are already in place resulting in faster operation [89]. This early start of the control section also allows the outputs that do not depend on all inputs to be generated without waiting for all the inputs to arrive. Its use relies on the assumption that such early data generation will not cause interference further down in the pipeline. In the above example, if \( s = 0 \), the constant value will be sent to the output even if input \( i \) has not arrived. This implies that, in the pipeline, it must be safe to send a token to output \( o \) before receiving tokens from both inputs in the conditional block. Example 5 examines a case when the use of active eager enclosure leads to incorrect operation.

**Example 4: control operators, composed commands and finite iteration**

The code in this example implements a special kind of one-place buffer that stores and duplicates the data until a *tail* flag located in the MSB (Most Significant Bit) of the input data signals the last transfer. When the tail flag is zero, the loop terminates and control is returned to the activating party. Figure 3.10 shows the code and figure 3.11 the resulting handshake circuit. The \# in line 14 of the code is the *smash* operator: a piece of syntactic sugar that provides the bit-array
casting required to access the MSB bit.

```plaintext
1 type hdata is 9 bits -- hdata [8] = tail flag
2 procedure dupbuf
3 {
4   input i : hdata;
5   output o1, o2 : hdata
6 } is
7   variable buf : hdata
8 begin
9   loop
10     i -> buf -- buffer data
11   ;
12     o1 <- buf || o2 <- buf -- relay & duplicate
13     -- until tail flag signals the last transfer
14     while (#buf[8] as bit) = 1 then continue
15   end
16 end
```

Figure 3.10: An example of a finite loop and command composition.

![Handshake circuit of the code in figure 3.10.](image)

The implementation is effectively a `repeat ... until` loop. As in the previous examples, figure 3.11 shows the different constructs with different shadings. The `Sequencer (;)` component at the top is required to generate the first iteration of the repeat loop before checking the exit condition. The other `Sequencer` corresponds to the operator in line 11 in the code that sequences the writes and reads of the variable `buf`. 
Notice that in the figure the While component implements the control for the conditional loop. Upon receiving a handshake on its activation port (located at the top), this component holds the activation handshake open and performs two sequenced actions: first it reads the guard value through the active input at its left and, if it is a 1’, the next action is a handshake in the passive output port. If the guard is 0, the While component completes the activation handshake and control returns to the activating party. Notice how in this example the control tree is relatively more complex because of the composition of Sequence and Concur operators inside the loop.

Example 5: Pitfalls in the use of active eager inputs

To illustrate a case where incorrect operation may occur as a result of the use of active eager inputs, let us consider the segment of code corresponding to a simplified description of a processor’s execution unit, shown in figure 3.12. The code describes the operations involved to generate the value to be written to the channel registerWrite2. The operation is as follows:

The instruction type and the result from the ALU are read (line 22) and, depending on the instruction type, either a value is read from memory into channel memDataIn (line 25) or the ALU result is sent through channel statusIn to generate a new status word (line 27). The description of the status word generator is shown in lines 2 - 12. For simplicity, this unit simply appends four zeros to the lower 8 bits of the input and casts the result into a value of type Datapath. Notice that active eager enclosures are used to read the inputs in both the status generator (line 8) and inside the loop implementing the condition (line 22).

Because the conditional construct guarantees mutual exclusivity in the generation of the values for channels memDataIn and statusOut, the select construct can be used to merge these channels into channel registerWrite2. As illustrated in line 37 of the example code, this could be achieved by using an instantiation of the merge2 module shown in figure 3.4. However, the mutual exclusivity assumption does not hold in the given description as a consequence of using eager active inputs in the module genNewStatus: inside this module, the (constant) lower four bits of channel statusOut are eagerly generated, without waiting for the input, as soon as the control activates the module (in parallel with the conditional and merge loops).

As previously explained, the select construct uses the arrival of bit 0 of the
Section 3.3 The Balsa language

1 -- new status generation unit definition
2 procedure genNewStatus ( 
3   input statusIn : Datapath; 
4   output statusOut : Datapath 
5 ) is 
6  constant SUFFIX = 0b0000 : 4 bits 
7  begin 
8    statusIn ->! then 
9      -- new status is \{i[7:0], 0000\} 
10    statusOut <- (#SUFFIX @ #statusIn[7..0] as Datapath) 
11  end 
12 end 
13 
14 -- declaration of a merge2 module of type Datapath 
15 procedure merge2_Datapath is merge2(Datapath) 
16 . 
17 .
18 procedure Execute( 
19   -- I/O declarations 
20 ) is 
21   -- some local declarations (not shown) 
22 . 
23 .
24 -- interesting segment of Execute stage: 
25   -- select operation 
26   loop 
27     aluResult, instrType ->! then 
28       case instrType of 
29         MEMREAD then 
30           getDataFromMem(aluResult, memDataIn) 
31         | SETSTATUS then 
32           statusIn <- aluResult 
33         end 
34     end ||
35   -- generate new status word 
36   loop 
37     genNewStatus(statusIn, statusOut) 
38   end ||
39   -- Merge values to write in second register bank port 
40   loop 
41     merge2_Datapath(memDataIn, statusOut, registerWrite2) 
42   end 
43
44 end -- Execute

Figure 3.12: Example of unsafe use of active eager enclosure.

input channels to determine which channel will be selected. The early arrival of the eagerly generated lower four bits of channel statusOut will activate too early its side of the DW. If the operation to be executed is a read from memory,
both inputs of the DW will end up activated. In this situation, the DW will erroneously activate the transfers on the two inputs of the CallMux generating interference on its output (the incomplete dual-rail codeword from statusOut will be merged (ORed) with the dual-rail codeword from channel registerWrite2). This interference will result in a deadlock, either because of the generation of invalid dual-rail codewords or the impossibility of completing the RTZ phase on channel registerWrite2.

In summary, within an active eager enclosure, every data generation construct that involves concatenation (like the @ operator, record construction and casting to unsigned wider data types) can be potentially dangerous. If the use of the result data further down the pipeline relies on a mutual exclusivity assumption and some portions of the concatenated data can be generated unconditionally (as within the genNewStatus module of the example) the non-eager active enclosure must be used.

**Example 6: permissive Concur**

If, in the previous example, non-eager active enclosure is used in the genNewStatus module, the mutual exclusivity of channels memDataIn and newStatus will be guaranteed. In this situation, it is possible to use the permissive Concur operator (111) between the loop of the conditional construct and the genNewStatus instantiation to allow these operations to write to the common channel registerWrite2, eliminating the need for a merge module, as shown in figure 3.13, lines 25 and 33. The 111 operator implicitly introduces a CallMux to merge writes to the same channel within the composed commands.

Figure 3.14 shows a simplified handshake circuit for the corrected version of the code in figure 3.12 (no active eager enclosure in line 8) and figure 3.15 shows the circuit for the new version. Because all of the parallel-composed commands are unbounded loops, the Balsa compiler inserts a cheaper WireFork (W) component instead of a Concur. The WireFork simply forks the activation signal to each of the Loop components and, just like the Loop component themselves, never returns an acknowledgement. PassivatorPush (•) components are used to connect active inputs and outputs as will be explained in section 3.3.13. In both circuits, only the merge section has been detailed to highlight the benefits of using the permissive Concur.

Comparing both circuits, it is clear that the new circuit is simpler: The whole
merge2_Datapath module has been replaced with a single CallMux. The new circuit benefits from having less datapath latency (the FVs have been removed and there is no control for the merge section). The reduction in components results in smaller area and lower energy as additional benefits. Finally, the resulting description is simpler.
Figure 3.15: Example of merging channels using the permissive *Concur* operator.

The benefits on performance and expressiveness allowed by the (111) operator were exploited in the design of the Forwarding Unit for the nanoSpa processor that will be described in section 7.4.

### 3.3.13 Interconnecting Balsa modules

Balsa circuits generally have active inputs and outputs, that is, the synthesised modules have *pull-push* input-output interfaces. To connect an active output port with an active input, a component that synchronises requests from both sides before acknowledging them is used: the *Passivator* (*PassivatorPush* in the case of data channels). Figure 3.16 shows the use of this component to connect two Balsa procedures (modules) using one control and two data channels and the implementation of a 1-bit dual-rail *PassivatorPush*.

### 3.4 Summary

This chapter introduced the Balsa Synthesis System and the Balsa language. Details of the compilation scheme targeting handshake circuits were presented with the aim of highlighting the mapping of the main set of language constructs that will be used in the next chapter. This chapter also presented some previously
undocumented features of the language, namely, the use of active eager enclosure and its implications on the expected operation of the circuit, and the use of the permissive Concur operator, illustrating in both cases their potential benefits.
Chapter 4

Optimising Balsa circuits

4.1 Introduction

The syntax-directed synthesis paradigm has been shown to be a powerful synthesis approach. However, its control-driven nature results in significant performance overhead [100, 101]. In an attempt to reduce this overhead, the following circuit-level approaches have been previously reported:

- **Peephole optimisations**: this technique is based on the identification of a pattern of components that can be replaced with a faster alternative [106, 83, 19].

- **Control resynthesis**: this technique consists on clustering sections of control trees and replacing these with an optimised controller that implements the same behaviour [19, 60, 88].

- **Component optimisation**: this is based on finding alternative designs for the handshake components that result in more concurrent, faster operation [85].

An orthogonal alternative to the above is to exploit the *directness* of the synthesis method at the description level. Highly expressive, high-level description languages like Balsa and Haste can result in naïve descriptions with poor performance unless the designer has a good understanding of the underlying compilation process. Furthermore, it is often claimed that in this approach, an experienced designer could make performance/power/area trade-offs. This task would be easier if the designer could have some insight into the impact of a particular construct or coding style.
This chapter explores the effects of directness in the performance of Balsa synthesised circuits and proposes coding techniques and optimisations that result in more concurrent, faster implementations. The chapter begins with the description of a set of language-level techniques for increasing the performance of Balsa circuits. Finally, new peephole optimisation and handshake circuits that further improve the performance of the designs are described.

4.2 Related work

In [85], Plana et al. used Balsa to demonstrate the impact on performance of some description-level techniques combined with the introduction of more concurrent handshakes components when applied to the synthesis of a RISC processor. In particular, true asynchronous operation of the processor pipeline, a data-driven coding style and the use of speculation within the execution stage are presented as performance-driven description techniques. In this thesis, those techniques are revisited and further investigated together with new techniques introduced here, using various design examples.

In a recent work, Hansen and Singh [47] describe a series of automated “source-to-source” transformations that optimise syntax-directed descriptions using a variety of concurrency-enhancing optimisations including: automatic parallelisation, automatic pipelining using pipeline variables, arithmetic optimisation and reordering of channel communication. The proposed transformations target Haste descriptions. Although considerable speed-ups are claimed, some of the example designs start with extremely naïve code sequences (with all operations initially sequenced), where significant improvements can be easily obtained. The transformations proposed here target both sequential and parallel compositions, make use of explicit buffering as an alternative to pipeline variables and do not use speculation to optimise conditionals.

The approach proposed in the mentioned work is limited to slack elastic [64] systems descriptions only (a slack elastic system preserves correct operation even if extra pipeline buffer stages are introduced in any channel). This limitation reduces the usefulness of an “automated” approach as it is frequently necessary for the designer to understand the nature of the transformations to ensure they are safe, which may represent a considerable design effort for the user. Furthermore, automatic code generation frequently needs to be used in conjunction with manual
optimisation because there may be some code that needs to be hand-crafted to meet specific design constraints. In contrast to the ones presented in the mentioned work, the examples used in this thesis are more complex and non-slab elastic: they contain Merges (uncontrolled multiplexers), like the processor and the router.

The approach used here is more general and attempts to give the designer a clearer understanding of the source of performance inefficiencies, the techniques available to reduce it and the trade-offs made. As an additional and important benefit, manual optimisation techniques can be applied to exploit the designer’s knowledge about the behaviour of the system. This knowledge is something that is more complex to automate because it cannot be inferred by analysing the code.

This work is complementary to the approaches presented above and to the circuit-level optimisation techniques. The techniques presented here could also serve as a source for optimising compilers or to enhance automated source-to-source transformations.

4.3 The data-driven description style

In Balsa/Haste it is relatively easy for a user to write a working, but most likely low-performance, description of a system due to their similarities with C and Verilog language. One of the major challenges for an asynchronous designer is to learn to think in terms of concurrent processes, instead of the easier to understand sequential processing found in imperative languages. An imperative, sequential description generates a large control tree that directs the flow of data in the datapath. This large control tree results in performance penalties that tends to increase with the complexity of the description.

As an example, consider the simplified description of the EXECUTE stage of the SPA processor given in [85], which is reproduced in figure 4.1(a). Here, all actions are explicitly sequenced and in every “step” the control tree activates the Fetch components (→) to guide the data through the required unit. Due to this lockstep mode, the control tree guarantees mutual exclusivity of the results, allowing the use of a simple CallMux (1) to write the results into the register write-back. The resulting simplified handshake circuit is shown in figure 4.1(b).

However, it is possible to describe a more concurrent operation by using a data-driven description style, that is, a description in which the arrival of data
activates the units. In the data-driven style the description of a circuit is divided into simpler, concurrent actions that communicate using channels. Given the asynchronous nature of the circuits, these actions are activated immediately by the data arriving at their inputs, process the information and generate outputs to activate the next unit.

This strategy is used in the alternative description of the SPA’s EXECUTE stage shown in figure 4.2(a). Instead of providing an explicit sequencing of actions (with its associated large, slow control tree), the actions are composed concurrently, with incoming data used as the activation. The resulting control tree is generally small and local to the modules implementing the actions. To guide the data, steer (demultiplexing with optional multicasting) and multiplexer units are added. Control for these units comes directly from the decoder and does not involve any sequencing. Notice that this steering and multiplexing is a specific requirement of the example, not a general feature of data-driven descriptions. The simplified handshake circuit is shown in figure 4.2(b).

Key to implementing data-driven circuits is an adequate partitioning of the circuit into actions/groups of actions that source and consume data. Internal channels will connect these actions. The partitioning also involves determining the group of actions that will necessarily require sequencing, as unnecessary sequencing is a well-known source of overheads. Sequencing is normally associated with the use of variables but also may be required to prevent deadlocks. Every
variable that has a write-then-read access pattern inside each iteration of a group of actions can be substituted by a channel write and an enclosing read (where the value can be read as many times as required). Only variables that store a value required in the next iteration need to be left in the description.

**4.3.1 Control driven to data driven example**

Consider the description of a branch metric unit (BMU) for a soft-decision-based Viterbi decoder like the one described in [16]. This unit takes two 3-bit quantities \((a, c)\) which are soft-coded representations of the two received bits in a Viterbi decoder. For each input, 000 (0) denotes the reception of a strong zero and 111 (7) indicates a strong 1.

The task of the BMU is to calculate the distance (branch weight) between the received pair and the ideal branch pattern symbols \((0,0), (0,7), (7,0), (7,7)\), as shown in figure 4.3(a). The distance to be calculated is the Manhattan distance, as this turns out to be equivalent to the Euclidean distance squared in this application [91]. The required branch weights are: \(d_{00} = a + c\), \(d_{01} = a + d\), \(d_{10} = b + c\), \(d_{11} = b + d\), where \(b = 7 - a\) and \(d = 7 - c\).

The linear weights are further minimised (reduced) by subtracting the \(x\) and \(y\) distance to the nearest ideal point, so the smallest linear metric is always made zero. This can be done by finding the smallest linear metric and then subtracting this value from every metric. Figure 4.3(b) depicts the BMU algorithm.

An almost direct translation of the branch metrics algorithm into Balsa is

```
steerRegData ||
doRegisterRead  ||
doShift  ||
doAlu  ||
doMul  ||
doMemAccess  ||
multiplexResults  ||
doRegisterWriteBack
```

![Figure 4.2: The simplified data-driven SPA EXECUTE stage [85].](image)
Section 4.3 The data-driven description style

shown in figure 4.4. This description is very similar to the one that a novice Balsa user wrote for this unit in [42], although it is not a completely fully-sequential naïve description: values for b, d and d00 are calculated concurrently, and after that, the remaining metrics are calculated. To find the smallest metric, they are compared in pairs (concurrently) and two are discarded. The process is then repeated to get the final result. The four reduced metrics (outputs) are also calculated in parallel by subtracting this value. Figure 4.5 shows the compiled handshake circuit (the diagram was generated using the breeze-sim-ctrl tool).

In the circuit, the highlighted control tree shows the six-way sequencer used to activated each group of concurrent commands labelled (1) to (6) at the right side of the given code. The control tree reflects the use of /BN and /DG commands in the description.

An examination of the algorithm reveals that all variables have a write-then-read pattern, so instead of using variables, we could use channels to pass data directly from sources to the commands that make the processing. The processing commands will use active enclosure to read from channels.

For instance, consider the first three groups of actions in the above description (lines 15 to 23) which are reproduced in figure 4.6(a). The two input reads can be changed into an active input enclosure of the actions (2) and (3) as both actions require the value of the input channels ia and ic. In this particular case, the enclosure requirements are relaxed (the results can be generated in any order as soon as the inputs are available) and we can use eager active enclosure.
Chapter 4 Optimising Balsa circuits

-- A Branch Metric Unit for soft-decision
-- Viterbi decoder with 3-bit quantisation

type TInp is 3 bits

procedure BMU(
    input ia, ic : TInp;
    output bm00, bm01, bm10, bm11 : TOut
)

is

    variable a, c : TInp
    variable b, d : TOut
    variable d00, d01, d10, d11 : TOut

    variable tempA, tempB, smallestM : TOut

    begin

    loop

    [ ia -> a || ic -> c ]; -- read inputs (1)

    -- first batch of calculations (2)

    [ b := (7 - a as TOut) ||
    d := (7 - c as TOut) ||
    d00 := (a + c as TOut) ];

    -- compute the other metrics (3)

    [ d01 := (a + d as TOut) ||
    d10 := (b + c as TOut) ||
    d11 := (b + d as TOut) ];

    -- now find the smallest metric (4)

    if d00 < d01 then
        tempA := d00
    else
        tempA := d01
    end

    if d10 < d11 then
        tempB := d10
    else
        tempB := d11
    end;

    -- resolve which is the smallest (5)

    if tempA < tempB then
        smallestM := tempA
    else
        smallestM := tempB
    end;

    -- generate the reduced outputs (6)

    [ bm00 <- (d00 - smallestM as TOut) ||
    bm01 <- (d01 - smallestM as TOut) ||
    bm10 <- (d10 - smallestM as TOut) ||
    bm11 <- (d11 - smallestM as TOut) ]

    end

end

Figure 4.4: Initial BMU description.

Inside the enclosure, variables b and d are replaced by local channels that are written during action (2) and concurrently read during action (3) using another active enclosure. Inside this last enclosure, variables d01 to d11 are replaced in a
similar fashion. Because the values are now available in channels, the \( \lhd \) operators are replaced by \( \rhd \) operators increasing the concurrency of the actions. These modifications are shown in figure 4.6(b).

The replacements and active enclosure use described above can be applied to the remaining actions, resulting in the optimised code shown in figure 4.7. The compiled handshake circuit is shown in figure 4.8. The active\( \text{EagerFalseVariable} \) (ae\( \text{FV} \)) components associated with each enclosure are light coloured, grouped and labelled for illustrative purposes. Notice how the six-way Sequence in the initial circuit has been replaced by a two-way Concur plus separated small controllers for each group of enclosed actions.
-- A Branch Metric Unit for soft-decision
-- Viterbi decoder with 3-bit quantisation

---

**Figure 4.7: Optimised BMU description.**
A quantitative evaluation demonstrates that the smaller, concurrent control trees increase the performance at the cost of some area penalty, as shown in the simulation results presented in table 4.1. The results are from pre-layout, transistor-level simulation using a 180 nm cell library. The energy results presented throughout this thesis correspond to dynamic energy only. The experimental setup consisted on processing 1000 random pairs of soft-coded symbols \((a,c)\) provided by an eager environment. The figure of merit is the average processing time \(t_{\text{process}}\) of a symbol pair.

<table>
<thead>
<tr>
<th>Device</th>
<th>(t_{\text{process}}) (ns)</th>
<th>Relative speed</th>
<th>Area (transistors)</th>
<th>Relative area</th>
<th>Relative energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMU Original</td>
<td>9.152</td>
<td>1.00</td>
<td>9663</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>BMU Optimised</td>
<td>6.906</td>
<td>1.33</td>
<td>10898</td>
<td>1.13</td>
<td>1.22</td>
</tr>
</tbody>
</table>

Table 4.1: BMU Simulation results.

Thanks to the directness of the compilation method and the availability of different constructs, there is no a single way of writing data-driven descriptions. Pipelining and parallelising descriptions can be done in different fashions as will be demonstrated in the next sections.
4.4 Optimising data-driven descriptions

In this section different performance-optimised data-driven description techniques will be introduced. To give a clearer idea of the effects in the code, a simplified version of the BMU (without the linear weight minimisation step) will be used here for code and handshake circuit examples. That is, the body of the reference data-driven description will be the code in figure 4.6(b) embedded in a loop ... end construct.

As an initial evaluation of the performance gains and trade-offs made, this section also presents pre-layout, transistor-level simulation results for the complete BMU example using the proposed techniques. Results for more complex designs (including the complete Viterbi decoder) will be presented and discussed in chapter 7.

4.4.1 Separating actions into concurrent loops

The example code in figure 4.9(a), which is already split in two groups of actions, can be split into two concurrent enclosed groups instead of having two nested enclosures. Furthermore, the outer unbounded loop can be split into two concurrent unbounded loops, where any value of the original enclosure required in the second loop must be passed using new internal channels. In this example, the values of \( i_a \) and \( i_c \) required in the second group are transferred together with \( b \) and \( d \), as shown in figure 4.9(b). In general, this “splitting” can continue until all grouping possibilities are exhausted, according to the dependencies of the commands. Notice the use of active eager enclosures in the description.

The resulting circuits are shown in figure 4.9(c) and (d). After the splitting process the datapath will be a pipelineable description without pipeline registers. On the control side, the control tree in the middle has been split and now the control for the second round of computations runs concurrently with the control of the input section. The new description results in the addition of two extra \( \text{aeFV}s \) (for the copies of \( i_a \) and \( i_b \) passed to the bottom loop). The four \( \text{aeFV}s \) decouple the RTZ phases of the control of the two loops, without adding any latency.

The results for the BMU description that uses this technique are labelled “\( \text{Lopt non-eager} \)” “\( \text{Lopt eager} \)” in the graphs of figure 4.10. These correspond to the use of normal and eager active enclosures respectively. All results are normalised...
to those of the BMU original design presented earlier in table 4.1. Let us refer for now to the first group of bars labelled “no ch. broadcast” in figure 4.10 (the other groups of results will be introduced later). From the graphs, the performance gain using the technique just introduced is ≈1.5 for the eager version (compare to the previous 1.33 in table 4.1 which also uses eager enclosures), with a relative area and energy of ≈1.3 and ≈1.5 respectively. Notice that the non-eager version does not produce any significant performance improvement, it is included in the results to highlight the benefits of the active eager enclosure.

\[
\begin{align*}
\text{loop} & \quad \text{ia, ic ->! then} \\
& \quad b \leftarrow (7 - \text{ia as TOut}) || \\
& \quad d \leftarrow (7 - \text{ic as TOut}) || \\
& \quad \text{d00} \leftarrow (\text{ia} + \text{ic as TOut}) || \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{-- compute the other metrics} \\
& \quad b, d ->! \text{then} \\
& \quad \text{d01} \leftarrow (\text{ia} + \text{d as TOut}) || \\
& \quad \text{d10} \leftarrow (\text{b} + \text{ic as TOut}) || \\
& \quad \text{d11} \leftarrow (\text{b} + \text{d as TOut}) \\
\end{align*}
\]

\[
\begin{align*}
\text{end} & \quad \text{end} \\
\text{end} & \quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{loop} & \quad \text{ia, ic ->! then} \\
& \quad b \leftarrow (7 - \text{ia as TOut}) || \\
& \quad d \leftarrow (7 - \text{ic as TOut}) || \\
& \quad \text{d00} \leftarrow (\text{ia} + \text{ic as TOut}) || \\
\end{align*}
\]

\[
\begin{align*}
& \quad \text{ta} \leftarrow \text{ia} || \\
& \quad \text{tc} \leftarrow \text{ic} \\
\end{align*}
\]

\[
\begin{align*}
\text{end} & \quad \text{end} || \\
\text{loop} & \quad \text{-- compute the other metrics} \\
& \quad \text{ta, tc, b, d ->! then} \\
& \quad \text{d01} \leftarrow (\text{ta} + \text{d as TOut}) || \\
& \quad \text{d10} \leftarrow (\text{b} + \text{tc as TOut}) || \\
& \quad \text{d11} \leftarrow (\text{b} + \text{d as TOut}) \\
\end{align*}
\]

\[
\begin{align*}
\text{end} & \quad \text{end}
\end{align*}
\]

Figure 4.9: Example of separating actions into concurrent loops (first steps).
An important remark with respect to the level of granularity of this technique is that the throughput will depend on the slowest stage and increasing the pipeline depth will increase the latency. Indiscriminate loop splitting (either manually or automatically) by just analysing precedences and/or dependencies may end up being suboptimal. The designer must take into account the balancing of the pipeline, the nature of the data and the behaviour of the environment among other factors. Being able to express the designer’s knowledge about the circuit is an advantage but also a challenge in syntax-directed descriptions.

Figure 4.10: Simulation results of different optimisations applied to the BMU.
4.4.2 Broadcasting values

Often within a pipeline, a value from a channel is required unconditionally and concurrently by more than one stage in the pipeline, as noticed previously with \( a \) and \( c \). Enclosure provides a means for multicasting values but it may prevent finer grain concurrency and deeper pipelining. For instance, in the code of figure 4.6(a) the groups of actions (2) and (3) are within the same enclosure, hence no new token can be processed by action (2) until action (3) has finished. A solution for this, shown previously in the loop splitting example (figure 4.9(b)), relied on duplicating the values required by the next group of actions inside the active enclosure, but more concurrent solutions for broadcasting are possible. In Balsa, there are two ways of specifying multiple concurrent receivers for the same channel:

i. Using implicit *broadcasting*: In the description, the channel is read in every place that it is required. In this case, the reads are fully synchronised: the data will be available to the reading processes only after every read request has been received. Similarly, data withdrawal will begin only after all reading processes have signalled the consumption of data.

ii. Using explicit *duplication* of the channel by means of enclosure. This method provides more decoupling between processing and the RTZ phases of the reads, as every request will be granted independently of the arrival of the others.

The code in figure 4.11 show these two forms of broadcasting in the simplified BMU example. This technique further improves concurrency, which results in higher performance at the cost of some area and energy penalties. The bins labelled “ch. duplicate” and “ch. broadcast” in the graphs of figure 4.10 shows the results for the complete BMU design when these techniques are applied. Referring to the “Lopt eager” columns, the increase in performance is now \( \sim 2.1 \) (slightly larger for the broadcast method). The relative area and energy are \( \sim 1.45 \) and \( \sim 1.65 \) when using channel duplication and a bit smaller (\( \sim 1.35 \) and \( \sim 1.50 \)) when using implicit broadcasting.

In this particular example, the synchronisation penalty imposed by the implicit broadcasting is not apparent because the design has balanced threads: all four outputs are generated using similar operations and the simulation environment generate inputs and consumes outputs eagerly. In designs with this balanced
behaviour, broadcasting has the advantage of less area and energy penalties. However, in designs with more complex, unbalanced thread execution patterns, like a
processor, thread decoupling provided by explicit duplication allows a head start for some of the threads required to complete an instruction, resulting in fully asynchronous operations and better performance.

In common with the previous technique, it is difficult to predict the places or levels of granularity to apply efficiently this technique by only analysing the operation precedences or data dependencies without input from the designer’s knowledge about the system.

### 4.4.3 Adding pipeline registers

To increase its throughput, a pipelined description requires inter-stage pipeline registers to decouple them. These can be added in two ways:

i. Using *pipeline variables* within the stage instead of the active enclosure, as presented in [47].

ii. Using explicit pipeline buffer modules (like the one described in section 1.2.3) between stages, as presented in [85].

These two styles are shown in the example codes of figure 4.12. Use of pipeline variables adds a *Sequencer* to the control tree and results in lower performance than the use of explicit pipeline buffers. Results in the graphs of figure 4.10 reveal this performance penalty. However, pipelining using variables is cheaper in terms of area and energy because no extra *FalseVariable* and *Passivator* components are required.

Results for the design that uses pipeline variables are labelled “\textit{Lopt non-eager + pipeline var}”. Results for the designs that use explicit buffering are labelled “\textit{Lopt non-eager + pipeline buf}.” and “\textit{Lopt eager + pipeline buf}.” (with active eager inputs). Notice how in the latter case, the synchronisation imposed by channel broadcasting has limited the effectiveness of the decoupling.

A detailed look at the results in figure 4.10 reveals that adding pipeline registers when using broadcasting or channel duplication has not noticeably increased the performance, but has increased the area and energy penalties. There are two reasons for this: Firstly, the BMU stages are very simple and have low latency (four bit adders/comparators), the extra latency of the pipeline registers reduces their possible benefits. Secondly, as seen in the previous examples (figure 4.9), the use of active inputs requires *PassivatorPush* components to interface with active
Chapter 4 Optimising Balsa circuits

-- Pipeline variables:
-- va, vc,
-- vta, vtc, vb, vc

loop
  [ ia -> va || ic -> vc ]
  [ b <- (7 - va as TOut) ]
  [ d <- (7 - vc as TOut) ]
  [ ta <- va || tc <- vc ]
end
loop
  [ ta -> vta || tc -> vtc ]
  [ b -> vb || d -> vd ]
  [ d00 <- (vta + vtc as TOut) ]
  [ d01 <- (vta + vb as TOut) ]
  [ d10 <- (vtc + vb as TOut) ]
  [ d11 <- (vb + vd as TOut) ]
end

-- procedure buf3 is buf(Tinp)
-- procedure buf4 is buf(TOut)
buf3(a, pa) || buf3(c, pc) ||

loop
  pa, pc ->!
  b <- (7 - pa as TOut) 
  d <- (7 - pc as TOut) 
  ta <- pa || tc <- pc
end
loop
  pta, ptc, pb, pd ->!
  d00 <- (pta + ptc as TOut)
  d01 <- (pta + pd as TOut)
  d10 <- (ptc + pb as TOut)
  d11 <- (pb + pd as TOut)
end

Figure 4.12: Pipelining: (a,c) using variables. (b,d) using explicit pipeline buffers.
outputs. When using dual-rail or other DI encoding these interface components require storage in the form of C-elements as shown in figure 3.16(b) Hence, the *PassivatorPush* acts as a simple *half latch* [91, 17]. (a half latch allows the active output to withdraw the data after synchronising with the active input request while the other side is in the processing phase). Each time a channel is duplicated using active enclosure, a half latch is added to the pipeline, providing decoupling between stages. Inserting explicit pipeline registers in this case will only contribute to increase the latency and area of the circuit.

In summary, the implicit storage added to the channels when specifying active inputs serves in some cases as a pipeline register which, when combined with the optimised control of the active eager inputs, efficiently implements decoupling between pipeline stages.

### 4.5 Optimising guards

Another common source of inefficiencies when coding in Balsa is related to the implementation of the guard expressions for conditional loops and for the *case* and *if* constructs. These conditional constructs require the use of handshake circuits that generate control channels from the datapath, like the *Case* component in figure 3.9 and the *While* component in figure 3.11. In many cases, the designer can optimise these datapath-generated control by evaluating the guards before their use in the construct, as will be demonstrated here.

```plaintext
input (a, b);
while a ≠ b do
    if a > b then a ← a − b;
    else b ← b − a;
output (a);
```

Figure 4.13: A pseudo-code specification of GCD [91].

Consider the GCD algorithm example, that computes the greatest common divisor of an integer. Figure 4.13 shows a specification of the GCD algorithm. Figure 4.14(a) shows a direct implementation of the algorithm in Balsa. In the implementation, the two guards (*va /= vb* and *va > vb*) are evaluated only after the control reaches each conditional structure, resulting in an unnecessary delay. The code also exhibits the common “problem” of auto-assignment, which in most cases introduces additional performance penalties (see section 3.3.5).
The performance-optimised description of the GCD shown in figure 4.14(b) illustrates how to solve the above problems: Firstly, to avoid auto-assignment, two additional variables (\texttt{tva} and \texttt{tvb}) are used as temporary storage. Secondly, the two required guards are evaluated in parallel and stored using 1-bit variables \texttt{neq} and \texttt{gt}. The resulting handshake circuits are shown below the code.

Notice in the circuit at the left how the body of the \texttt{loop ... while} (highlighted) contains four sequenced operations:

i. Evaluate the guard expression for the \texttt{loop ... while} construct and proceed accordingly.

ii. Evaluate the guard expression for the \texttt{if} construct and make the decision.

iii. Update one of the auxiliary variables (labelled only for variable \texttt{b} in the circuit).

iv. Update one of the variables (labelled only for variable \texttt{b} in the circuit).

In the optimised circuit at the right the loop has only three sequenced operations:

i. Read the guard expression for the \texttt{loop ... while} construct and proceed accordingly.

ii. Read the guard expressions for the \texttt{if} construct \texttt{and} update one of the auxiliary variables.

iii. Evaluate and store both guards, and update both variables.

Table 4.2 shows the simulation results for the two circuits above. The table compares the average processing time required to calculate the GCD of two 8-bit numbers. Area and energy results are also given.

<table>
<thead>
<tr>
<th>Device</th>
<th>$t_{process}(ns)$</th>
<th>Relative speed</th>
<th>Area (transistors)</th>
<th>Relative area</th>
<th>Relative energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCD Original</td>
<td>181.68</td>
<td>1.00</td>
<td>6856</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>GCD Optimised</td>
<td>133.26</td>
<td>1.36</td>
<td>6991</td>
<td>1.02</td>
<td>1.14</td>
</tr>
</tbody>
</table>

Table 4.2: GCD Simulation results.
As the reader may have already noticed, in this example area and energy are being traded for speed. On each iteration, there is a redundant update operation
of the variable that does not change and two 1-bit variables are used. The design with the optimised guard is 36% faster at the cost of 14% extra energy and negligible area increase.

4.5.1 Encoding multiple guards

In situations where multiple guards are required, it is better to encode the guards into a multi-bit variable and use a `case` construct instead of the more straightforward (but slower) multi-guarded `if` construct. Consider the example code in figure 4.15 adapted from the description of the input buffer of a sliced-channel wormhole router designed in Balsa [90]. Each router has five I/O ports, namely, Local, North, South, East and West. The code shown corresponds to the South input buffer and has been simplified for clarity: only the operations over the dataless `sync` channels that generate the request to the destination ports are detailed.

The first value received at input `d_in[0]` is the `header flit`. It contains the XY destination addresses that will be compared with the addresses of the router `addrX` and `addrY`. The destination is chosen accordingly to the comparisons and the order of priority specified in the description. The optimised code is shown in figure 4.16. In this new description, instead of using the `if` construct, all guards are evaluated and stored in parallel with the buffering of the input value within an active enclosure. The four bits generated by these evaluations are then joined and used as the guard expression of a `case` construct. Also, in this new construct the encoding of the guards reflect the priority expressed in the original description.

Simulation results for the wormhole router, which are detailed in section 7.5.3, indicate that the guard encoding technique contributes to an increase of 10% in speed and a reduction of the area to 86% of the original, at the cost of 7% increase in energy consumption.

4.6 New peephole optimisations

The previous sections showed how to write optimised Balsa code targeting high performance. In general, the circuit derived from an optimised data-driven description will consist of small clusters of control components which reduces the possibility of further optimisations using control resynthesis.
Part of this research work focused on analysing these optimised circuits and looking for further optimisation opportunities using component substitutions or redesign. This section introduces some new peephole optimisations and components aimed to increase the performance of the synthesised circuits. Datapath optimisations include the removal and substitution of FalseVariable components and the use of a more concurrent Fetch component. Optimisations for the control of unbounded active input enclosures and unbounded read-then-write actions over a variable are also presented.

The optimisations introduced in this section were manually applied to some of the design examples presented in this thesis, as they are not yet incorporated into the Balsa design flow. Modifying the Balsa compiler to automate these was considered more a time-consuming exercise on compiler development than a contribution to the objectives of this research.
type Destination is enumeration
    WEST, NORTH, EAST, LOCAL
end

procedure input_buf_south
(
    array 4 of input d_in : 9 bits;
    array 4 of sync req;
    array 16 of output d_out : 9 bits
) is
    variable buf : array 4 of 9 bits
    constant addrX = (2 as 4 bits)
    constant addrY = (2 as 4 bits)
    channel n, e, w : bit -- guard variables
    channel d_in0 : 9 bits

begin
    loop
        -- NOTE: d_in[0][4..7] = X, d_in[0][0..3] = Y
        d_in[0] ->! then
            n <- (#(d_in[0])[4..7] as 4 bits) < addrX ||
            e <- (#(d_in[0])[0..3] as 4 bits) > addrY ||
            w <- (#(d_in[0])[0..3] as 4 bits) < addrY
            d_in0 <= d_in[0] -- replicate d_in required
        end
    end

    n, e, w ->! then
        case (#v @ #e @ #n as 3 bits) of
            0b1xx then sync req[NORTH]  
                -- data transfer commands ommited
            0b01x then sync req[EAST]  
                -- data transfer commands ommited
            0b001 then sync req[WEST]  
                -- data transfer commands ommited
            else sync req[LOCAL]  
                -- data transfer commands ommited
        end
    end

end

Figure 4.16: Optimised, simplified description of the South input buffer.

4.6.1 Removing redundant FalseVariables

As demonstrated previously in section 3.3.12, active input control can be used in Balsa when there is no input choice. In this case, Fetch and FalseVariable (or just activeEagerFalseVariable) components are used to implement the construct.

In cases when the input channels are unconditionally read only once and the control simply transfers the value to a consumer module in the datapath, the
FalseVariable can be removed safely. This can also be done with the activeEagerFalseVariable component.

Figure 4.17: Handshake circuit for example in figure 3.6, (a) original, (b) optimised.

As an example, consider the Balsa code for a simple adder shown previously in figure 3.6 where two input channels, a and b, are read and then added to produce the output o. For convenience, the resulting handshake circuit is reproduced again in figure 4.17(a). In this case, because the two input channels are unconditionally read just once, both FVs and the Synch are redundant: the control can initiate the read operation by directly triggering the transferrer at the output, which can then immediately start pulling the values from the input channels through the (+) operator. Figure 4.17(b) shows the optimised circuit.

The above transformation results in both latency and area reduction, yet preserving the external behaviour of the circuit. The control tree is reduced to
just the activation channel and the two $FVs$ are removed from the datapath. In general, in order to apply this optimisation, the single read of the $FalseVariables$ must not be activated through the use of a conditional component ($Case$, $While$ or $DW$). For instance, figures 3.5 and 3.9 are examples of circuits where this optimisation cannot be applied.

### 4.6.2 Control of active enclosures

When two or more channels are used as inputs in an active enclosure, Balsa introduces a $Fork$ component to broadcast the activation to the $Fetch$ components that push data into the $FalseVariables$. See for instance figure 4.17(a). With active eager inputs, this signal is passed to the trigger inputs of the $activeEagerFalseVariables$, as shown in figure 3.9.

In both cases, the signal control channels of all inputs are synchronised using a $Synch$ component, which activates the command that reads from the enclosure once all signal requests have been received. Figure 4.18 shows the circuit implementations of the $Fork$ and $Synch$ components. It can be seen that these components have mirrored circuits. The $Synch$ implementation guarantees that, for every input $I_i$, $I_{i:req}$ occurs before $O_{req}$. Mathematically, $I_{i:req} \prec O_{req}$. Equivalently in the $Fork$, for every output $O_i$, $O_{i:ack} \prec I_{ack}$. The $Fork$ synchronises all the transferrer/trigger acknowledges before acknowledging the activating party.

![Diagram](image)

Figure 4.18: (a) $Fork$ implementation. (b) $Synch$ implementation.

A $Permanent$ procedure is a procedure activated using a $Loop$ component that is either connected directly or through nothing other than $WireForks$ to the global circuit activation, like the circuit shown in figure 4.19(a). In these cases the synchronisation of acknowledges imposed by the $Fork$ is redundant: the $Loop$ component does not acknowledge its activation to the caller, hence each $aeFV$/$Fetch$ may be activated independently with separated $Loops$.
The *Synch* component in the enclosure structure guarantees that only one token from each *FV* or *aeFV* is allowed during each execution of the enclosed command. Each *Loop* can issue a new token only after the enclosing command completes. Eliminating the *Fork* reduces the latency of the control and increases the concurrency at the inputs. Figure 4.19(b) shows the optimised circuit. The *WireFork* is required to fork the activation request to all the control loops. Note that the *aeFV* and its activating *Loop* could be amalgamated into a single component.

![Figure 4.19: Permanent active eager input: (a) original, (b) with optimised control.](image)

### 4.6.3 Unbounded read-then-write on variables

This section introduces an optimisation of the handshake circuit required to perform unbounded read followed by write actions in variables, based in the unfolding of the first read operation and the use of the optimised sequencer introduced in [89].

**Performance of sequenced operations**

In synchronous circuits, the sequencing of events is straightforward: event *A* is sequenced with event *B* if they occur at different *clocking events*, that is, a full clock pulse, a clock level or a clock transition. In an asynchronous environment,
sequencing is more complicated and generally includes extra control overhead. Sequencing of handshake events must follow the protocol rules in order to avoid data or control hazards that may cause malfunction and deadlock.

Depending on the degree of handshake overlapping allowed, Balsa generates two types of sequencers based on the S-element and T-element respectively [89]. Figure 4.20 presents a block diagram of such components with their respective STGs (see section 2.6.1). Figure 4.21 shows the implementations of the Balsa sequencers and their respective STGs as introduced in [89].

![Figure 4.20: (a) S-element. (b) T-element. (c) S-element STG. (d) T-element STG.](image)

Notice that in the sequencer based on the T-element, the RTZ phase of the first command overlaps with the processing phase of the second command and the RTZ of the activation, which results in a more concurrent operation. Unfortunately, it is not always possible to use this type of overlapping due to the possibility of introducing write-after-write (WAW) and write-after-read (WAR) hazards. For a complete discussion of these issues, the interested reader can refer to [89].

A performance penalty occurs in designs where repeated read-then-write operations occur on the same variable. An unbounded repetition of this type can be described in Balsa as shown in the piece of code in figure 4.22(a), where processes `rd_proc()` and `wr_proc()` access the common variable `V`. Figure 4.22(b) shows the resulting handshake circuit. It is necessary in this case the use of a sequencer based on the *S-element* because the use of the T-element based sequencer may introduce a WAR hazard. This hazard is caused by the RTZ phase of the first command trying to close the variable read port concurrently with the second
command trying to write new data. If the new data arrives first it will appear at the output of the read port before it closes, potentially altering the result of the first command [89].

If the first read operation is taken out of the loop construct (the first read operation is *unfolded*), as the code shown in figure 4.23(a), the behaviour will remain the same, but now the operation inside the loop is a write-then-read, which does not have WAR hazards. In Balsa, a write-then-read sequence to a local variable within a procedure will generate a *Sequencer* based in the T-element. However, if the write and read processes reside in separate modules running in parallel, the Balsa compiler is conservative, as the level of allowed overlapping in communications is unknown, and inserts a safe sequencer based on the S-element. Performing this optimisation at the source code level requires the use of multiplexers in the datapath to merge the reads and duplicate blocks (larger area, energy and latency) as shown in the resulting circuit of figure 4.23(b).

Figure 4.21: Balsa sequencers: (a) based on the S-element, (b) based on the T-element [89].
In order to avoid hardware duplication, Balsa allows the use of *shared procedures* with the limitation that local channels may not be accessed [5]. The proposed solution is to substitute the loop-sequencer control structure obtained for unbounded loop descriptions like the one in figure 4.22(b) by the optimised control shown in figure 4.24(b). This new controller allows write and read RTZ overlapping and does not have local channel accesses restrictions.

In dual-rail circuits, the time required to complete the RTZ phase increases proportionally to the width of the data because the completion detection circuit must check more bits. Table 4.3 shows transistor-level simulation results of first-read-unfolded loops with different data widths. The simulated loop was a simple
Section 4.6 New peephole optimisations

--- other declarations
variable V : someType
--- other declarations
begin
  loop
    rd_proc(rd_args);
    wr_proc(wr_args)
  end
end

Figure 4.24: Optimised first-read-unfolded read-write loop.

read-then-write to a variable. These figures give an estimated upper bound for the performance gain that can be obtained and show that for datapath widths greater than 3 bits, the speed-up achieved by RTZ overlapping is greater than the overhead of the merge required in the unfolded control tree of figure 4.24. Section 7.4 presents the design of a Forwarding Unit that makes use of this optimisation technique.

<table>
<thead>
<tr>
<th>width (bits)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed-up (%)</td>
<td>-11.8</td>
<td>-2.5</td>
<td>-1.0</td>
<td>5.0</td>
<td>7.2</td>
<td>9.0</td>
<td>8.8</td>
<td>11.4</td>
</tr>
</tbody>
</table>

Table 4.3: Influence of data widths in first-read-unfold of read-write unbounded repetitions

4.6.4 Fetch component with concurrent RTZ

The Balsa dual-rail Fetch component, shown in figure 4.25(a), consists of wires only, with broad data validity in both data ports. The signal transition graph (STG) in figure 4.25(b) shows how the RTZ phases of the activation, input and output are fully sequenced.

In [81], Peeters described two single-rail transferrers with concurrency in the data channels, the par-ser and the ser-par, but its implementation in dual-rail would require completion detection inside the Fetch. The Fetch proposed here, focuses on the concurrency of RTZ phase of data and activation channels: if the handshake on the activation channel is itself enclosed within the handshake
of a wide data channel, the RTZ of that channel will delay unnecessarily the RTZ of the input data channel. In Balsa circuits, this situation occurs when the activation is generated by a Case component whose input data port has a considerable number of bits, as found in the implementation of the Decode stage of the nanoSpa processor [89, 95] or in the write index for arrayed variables with many entries. A more concurrent operation can allow the RTZ on the activation channel to occur in parallel with the RTZ on the data channels. Figure 4.26 (a) shows the new circuit for the dual-rail fetch and the resulting STG.

![Diagram](image)

Figure 4.25: Conventional Balsa dual-rail Fetch: (a) circuit, (b) STG.

![Diagram](image)

Figure 4.26: Fetch with concurrent RTZ: (a) circuit, (b) STG.
The new control interface now features a \textit{T-element} which provides the desired decoupling. In order to obtain performance benefits, substitution of the wires-only \textit{Fetch} should only be made when the \textit{Case} component that activates it has a slower RTZ than the delays introduced by the controller in the new \textit{Fetch}, but this threshold can be easily tunable in the compiler. The \textit{nanoSpa} processor and the Viterbi decoder presented in chapter 7 are used as design examples to evaluate this optimisation.

\subsection*{4.6.5 Summary}

This chapter has presented a number of description-level optimisations together with their effects in performance, resulting circuit structures and trade-offs made. First, the data-driven description style was introduced as a technique that, using the arrival of data to activate the processing units, results in faster circuits with smaller and localised control sections.

Description level techniques that result in faster data-driven descriptions were introduced and analysed. These included: separation of actions within unbounded loops to increase concurrency, broadcasting styles and stage decoupling techniques. The effects of the use of active eager enclosures with these techniques were also analysed. The reduced control tree achieved with these optimisation techniques combined with the head start of the control provided by the active eager enclosure contribute to the increase in performance of the circuit. The effects on the performance of the circuits clearly depend on the nature of the operations implemented. However, usually there will be some energy and area penalty as shown in the results for the running example.

Early evaluation of guards and encoding of multiple guards for conditional \texttt{loops} and \texttt{case} constructs were also presented as a way of increasing the performance. Because the structures that implement the \texttt{loop} and \texttt{case} constructs generate control signals from the datapath, optimising the decision-making circuit speeds up the control. Finally, some new peephole optimisations for the resulting optimised handshake circuits were proposed. These include the removal of single-write with unconditional single-read \textit{False Variables}, the optimisation of the control of the active input structures, the optimised control for unbounded write-then-read operations and a more concurrent version of the \textit{Fetch} component. The techniques and optimisations presented here will evaluated and further discussed in chapter 7 using a number of substantial examples.
Chapter 5

Optimising Token-flow circuits and descriptions

5.1 Introduction

In chapter 3 the Balsa synthesis system was introduced along with examples of handshake circuit implementations of synthesised circuits. The components produced are similar to those produced by the Tangram system, the precursor of Handshake Solutions’ TiDE system [23]. Balsa was developed to allow optimisation opportunities in handshake circuit designs to be explored. In particular, the FalseVariable component and input-enclosure language construct [5] have allowed pipelined descriptions with alternating latch and combinatorial handshake processing stages to be more naturally described.

The Teak system was proposed by Bardsley [6] as part of his research work within the APT Group at The University of Manchester. The Teak system extends the degree to which the Balsa language can sympathetically be used to describe pipelined systems by proposing a new set of components, synthesis rules and compiler. The aim of the Teak system is to provide a path for future performance increases in Balsa synthesis by exploiting high performance pipelined asynchronous circuit styles. The author of this thesis has contributed to the Teak System with:

i. the optimisations ideas presented in section 5.3 and their automation.

ii. automatic latch insertion strategies presented in section 6.4.

iii. the description-level optimisations presented in section 5.4.2.
iv. the evaluation of Teak using many of the design examples presented in chapter 7.

Some of the contents of this section are based in [6], a paper written by the author and Andrew Bardsley.

5.2 The Teak system

Teak replaces the dataless activation channel (used to enclose the behaviour of program fragments in handshake circuit synthesis) with separate go and done channels. Control/datapath interactions using components which exploit signal-level event interleaving are replaced by the rendezvous/forking of control and data channels with local handshaking to complete control interactions. This separation of “go” and “done” makes Teak much more like the Macromodules system [93] than handshake circuit systems. However, the ability to merge control and data channels gives the Teak system more flexibility.

Treating control channels in this way allows all the optimisation techniques usable with pipelined asynchronous systems (i.e. those with input-enclosing-output processing stages and decoupling latching stages) to be used on Teak circuits whilst still allowing local sequenced behaviour by using control channels.

Explicit pipeline latch insertion (also referred to as buffering in this chapter) is used to decouple one component from another and to introduce the desired degree of token storage to enable the circuit to function and, looking beyond the work presented in this thesis, to allow more transforming synthesis methods to increase circuit parallelism.

5.2.1 Teak components

There are currently eight Teak components (as shown in figure 5.1):

Steer (S): conditional steer of input to exactly one output. Parameterised with disjoint match conditions for each output and bit ranges to carry to outputs. With 0 bits carried to outputs, Steer works like the Balsa Case component.

Fork (F): unconditional n-way fork. Fork can be parameterised by which (if any) bits of the input are carried to each output. A two-way Fork of n and 0 bits can be used to generate a control token from moving data.
**Merge (M):** input on one of the input ports is multiplexed towards the output. Inputs must be mutually exclusive. In some configurations, Merge may have to cope with second input arrival during first input activity.

**Arbiter (A):** merge with arbitration between inputs.

**Join (J):** unconditional n-way join. Concatenates data bits of arriving inputs.

**Variable (V):** persistent storage. Separate write and read sections allow arbitrary control ordering/conditionality of reads. Variables allow complicated control activity without incurring the cost of always moving data along with control around a circuit. ‘wg/wd’ and ‘rg/rd’ (go/done) pairs make all writes data initiated and control token completed, all reads control token initiated and data delivery terminated.

**Operator (O):** any and all data transforming operations. Inputs are formed into a single word. Internally an Operator is organised into interconnected terms allowing Operators to be amalgamated or separated to allow cheaper implementation or Latch insertion.

**Latch (L):** data storage and channel handshake decoupling.

All of the components, except Latch, can be implemented with any chosen degree of input to output channel coupling (i.e. concurrency of handshaking events). Latch must provide at least some decoupling so that it can be used to separate pipeline tokens. In this way, Teak components resemble the components of other elastic token pipeline systems.

The Variable is included in this component set in order to allow sequential, storage-centric descriptions to be mapped directly into hardware. This is in contrast to other token flow approaches to asynchronous synthesis [116] [103] which perform single assignment analysis on the input language to allow variables to be eliminated in favour of pipeline buffers. This decision was made to allow the exploration of the possible power and area implications of retaining ‘fixed’ variables. Also, pipeline buffer-only approaches find it difficult to handle descriptions of persistent register banks without messy ‘register refreshing’ loops [100].

Figure 5.2(a) shows the one-place buffer example from Section 1.2.3 constructed from Teak components using synthesis rules from Section 5.2.2.
handshake circuit for this example is reproduced here in figure 5.2(b) for comparison.

Notice that the Loop component has become a loop comprised of a Merge (to introduce the ‘go’ token), a Join (to meet incoming data), and a Fork (to return a token back around the loop, through the Merge, after the output command) rather than a composition of enclosing control components.

### 5.2.2 Teak synthesis

Teak synthesis is initially syntax directed. Optimisations can then be performed on the generated Teak component netlists (Teak circuits). Each command in a Balsa description is mapped into components with dangling ‘go’ and ‘done’ control channels (a few commands never terminate and have no ‘done’). Expressions,
Figure 5.2: (a) Teak circuit for 1-place buffer, (b) Handshake circuit for 1-place buffer.

channel accesses and assignment left-hand sides similarly have a pair of dangling channels: one bearing data and the other a control initiating/completion channel. Control can be sequenced by joining commands ‘done’ to ‘go’ in a chain. Data and control usually meet with Fork and Join components.

As with Balsa intermediate Breeze netlists, there are many possible choices of data encoding and signalling protocols on the channels between components. As Teak deals with the flow of tokens rather than enclosing handshakes, Teak component implementations also have choices of the degree of interleaving between input to output handshakes, the use of weak-condition behaviour and storage within components. The Teak synthesis system consists of a single front-end program called teak. There are a number of switches that allow technology mapping to be specified and various plotting options and optimisations.

Channels

Channels in Balsa have no capacity. Inputs and outputs on a channel form a synchronisation where either party can delay the transaction until both are ready for data to be transferred. In Balsa, the select command (which allows choice based on order of arrival of data on a number of channels) and the ‘enclosed’ channel input command can be used to exploit the non-atomic nature of asynchronous channel construction to allow latch-less implementations of data processing stages to be described. In such stages, data processing and outgoing channel outputs
are enclosed within the input handshake, as described in section 3.3.9. Alternating such stages with latch-containing pipelining stages allows push pipeline-like structures to be built.

![Figure 5.3: Balsa-style channel implementation.](image)

Figure 5.3 shows a single output, single input Balsa-style channel implemented using Teak components. The pair of data and acknowledging (“done”) channels between output and input commands form a synchronisation and limit to a single token the capacity of the loop formed from output command (as data), through input command and back to the output command (as an acknowledging “done”). Note the use of Forks and Joins between data and control.

Unfortunately, Balsa’s channel implementation does not allow the capacity of buffered Teak channels to be exploited. Instead, the semantics of Balsa channel has been changed to make writes “fire and forget”. Channel outputs and inputs are no longer synchronised and enclosure inside a sending handshake can no longer be relied upon. In practice, this reduces the utility of the select language construct but allows descriptions to be formed which exploit (or possibly rely upon) non-zero channel capacity. This introduces an incompatibility with the Balsa system’s interpretation of descriptions.

Figure 5.4 shows how channel read and write commands are combined to form a complete Teak style language-level channel. The \( i \) and \( j \) constant-valued Operator components “tag” the request channels from different input/output commands so that once those requests are merged, with the following Merge component, the source of the request is encoded on the Merge output. This common request is then Joined to a token Forked from the outgoing data Merge (or,
for inputs, the incoming data itself) and Steered to provide the local command acknowledgements.

The combination of tagging Operators, the following Merge and the Join/Steer combination (the two shaded boxes in figure 5.4) plays a similar role to the Balsa DecisionWait [3] Handshake Component. This involves steering an incoming token (in this case the acknowledgement from the data-bearing merge) to the correct output based on the arrival of a single token on one of a group of input tokens (in this case, the choice of output command site). In Teak, the component parts of the DecisionWait are separated, rather than provided as a single component, to allow for flexibility of Latch insertion.

In cases where acknowledgement tokens need not be steered (e.g. where there is only one read or write to a channel in the description) much of the control/data interaction can be optimised away (as shown in figure 5.5). This implementation is similar to that of figure 5.3, but without the sequencing of variable write to the output command’s “done”.

Figure 5.4: Multiple-output channel implementation.

Figure 5.5: Channel component optimisation.
Commands

Figure 5.6 shows sequential and parallel composition of commands. Command “go” and “done” channels can be connected in series to form sequencing, so no explicit Sequence component is required. Parallel composition requires two components (Fork and Join) in contrast to Balsa’s Concur component which contains both functions in one component. Figure 5.6’s presentation of command composition is very similar to that used in non-return-to-zero (2-phase) signalled handshaking, as is illustrated by Brunvand [18]. Note, however, that here we are using handshake channels rather than individual wire signals for each of “go” and “done”. On a channel, the token recipient can stall a handshake (by denying an acknowledgement) and so the token capacity of a string of commands is not necessarily limited to one (i.e. the strict alternation of “go”, “done” events between all commands). Where resources are not shared between sequentially composed commands, this property allows pipelining to naturally arise.

Figure 5.6: Sequential/parallel composition.

Figure 5.7 shows the structure of a loop ... while command. The Steer component provides the control choice at the top of the loop. Note that the loop formed by the Merge and Steer components must have at least some buffering to prevent deadlock. Insertion of Latches will, obviously, affect circuit performance. Section 6 discusses different strategies for buffering and presents those currently available in the Teak System.

A non-terminating loop may be implemented by using the “done” of CMD1 to close the loop and removing the Steer, the COND and the CMD2 blocks. This is illustrated in figure 5.2(a).
Expressions

Expressions are compiled by adding pairs of “rg” (read go) and “rd” (read done) ports on variable components, and Operator components to process the read data. Reading from channels within expressions (when within select commands or enclosed input commands, e.g. \texttt{chan -> then var := chan + 1 end}) is achieved by inserting Variable components to capture channel read data, and then using read port pairs on those variables to use that data. These variables can often be removed if data is unconditionally used within the body command.

5.3 Optimising Teak circuits

This section introduces some optimisations for Teak circuits by exploiting the properties of components both individually and in groups. Optimisations are presented using simplified practical descriptions extracted from the design examples used in chapter 7. The plots of all Teak circuits shown in the following sections were generated automatically using the Teak System.

5.3.1 Variables

In cases where reads from a channel occur unconditionally after every write, the Variable can be removed (for single-read channels) or replaced by a cheaper and faster Fork component (for multiple-read channels), provided the Variable components are not used to enforce sequencing. Figure 5.8 shows a single-write followed by unconditional single-read channel structure before and after optimisation.

As an example, consider the Balsa description of an $n$-bit full adder whose output is separated into sum and carry-out portions shown in figure 5.9. The resulting circuit will contain Variable components implementing the inputs on $a$ and $b$ and the channel $cs$ as described in section 5.2.2.
Figure 5.8: Variable single read-after-write optimisation.

1 procedure adder (  
2 input a, b : N bits;  
3 output sum : N bits;  
4 output carry : bit  
5 ) is  
6 channel cs : N+1 bits  
7 begin  
8 loop  
9 a, b -> then  
10 cs <- (a + b as N+1 bits)  
11 end ||  
12 cs -> then  
13 sum <- (#cs[0..N-1] as N bits) ||  
14 carry <- (#cs[N] as bit)  
15 end  
16 end  
17 end

Figure 5.9: Balsa code for n-bit full adder.

For simplicity, let us consider only the part of the synthesised circuit that provides the sum and carry outputs as shown in figure 5.10(a), which corresponds to lines 12-15 in the source code.

The Variable that implements the channel cs has a single write port and two read ports (for sum and carry). Reads are initiated as soon as the cs Variable ‘wd’ (write done) port indicates that new data has been stored. The Fork component at the top provides tokens for both read ports. As a write operation is directly followed by a read, this Variable can be substituted by a Fork that provides ‘sum’, ‘carry’ and ‘done’ results as shown in figure 5.10(d). An additional benefit of this type of optimisation for dual-rail circuits is that the forked channels would only need to wait for the arrival of those input bits that will be carried to the output.

The above optimisation can be viewed as a 3-step process:

i. The Fork labelled 1 is displaced ‘downstream’ in the datapath, after the Variable cs, leaving a single write, single read Variable, as shown in figure 5.10(b).

ii. Variable cs can be removed as a write is directly followed by a read and
the three *Forks* can be merged into a four-way *Fork*, leading to the circuit in figure 5.10(c).

iii. Now, the *Join* component in figure 5.10(c) is redundant because both inputs come from the same fork. The inputs of the *Join* can be merged and the final circuit is shown in figure 5.10(d)

Similar kinds of optimisations based in component displacement will be presented in the following sections.
5.3.2 Fork displacement

In some circumstances, Fork components can also be displaced ‘upstream’ in a data or control path to allow for more concurrent operation. Consider the segment of code in figure 5.11 where the results generated by the two output commands must be written sequentially to a common channel `out`.

```plaintext
procedure tenFifteen (
  output out : 4 bits
) is
begin
  loop
    out <= 10 ; -- exprA
    out <= 15 -- exprB
  end
end
```

Figure 5.11: Sequential write to a channel.

![Diagram](a)

Figure 5.12: Sequenced channel write example: (a) original, (b) after Fork displacement.

The resulting circuit is shown in figure 5.12(a). Note how the Forks labelled `U` fork the result of `exprA` and `exprB` to generate the output and the “tag” constants `cA` and `cB`. As explained in section 5.2.2, those constants indicate which
of the expressions will be output in the next iteration via the *Steer* component. If those *Forks* are moved upwards through the expression generators, as in figure 5.12(b), the constant that steers the control for the next iteration will be generated concurrently with the output. This kind of displacement can be done through any data transforming operation or even single-input command blocks.

### 5.3.3 *Fork-Merge-Join* and *Steer-Merge*

Another target for optimisation are *Fork-Merge-Join* and *Steer-Merge* compositions. In Teak circuits, *Forks* are used in a datapath to generate a control token from a data token, to either synchronise or sequence operations. Sometimes all the *Forked* control tokens from a set of mutually exclusive data results need to be *Merged* into a single token. At some point further down in the pipeline this new control token will rendezvous in a *Join* with a second data or control token. If the second token is derived from the merging of the aforementioned set of mutually exclusive data sources, the *Fork-Merge-Join* composition for the control tokens can be simplified.

Consider the segment of code in figure 5.13(a) which is a simplified version of a “sign adjust” unit for the multiplicand input of the Booth’s multiplier in the nanoSpa processor [95]. The circuit takes an $N$ bit input word $b$ and, depending of the type of multiplication specified by the $\text{mType}$ input, either appends $M$ zeroes after the most significant bit of input $b$ or sign-extends it to $N + M$ bits to generate the adjusted output $ba$. For clarity, let $N = 8$ and $M = 3$ in this example.

The unoptimised circuit is shown in figure 5.13(b). In this figure, the dotted blocks labelled $\text{inB}$ and $\text{inM}$, contain the implementation of the two input channels reads and writes. The *Fork-Merge-Join* optimisation will be applied to the shaded block of figure 5.13(b), labelled $\text{BOut}$. In this block, the output data from *Operators* $\text{zE}$ (zeroExtend) and $\text{sE}$ (signExtend) are forked to produce control (thin lines) and data (wide lines) tokens. The data tokens are merged and then forked again to produce the output $ba$ and a new control token ($\text{Merge}$ and $\text{Fork}$ labelled $c0$).

The control tokens from the top *Forks* in block $\text{BOut}$ generate tag values (constant *Operators* $\text{2'd1}$ and $\text{2d'2}$) required to steer the control to the correct source in the next iteration (components labelled $c1$). As both data tokens are
loop
  mType, b -> then
  case mType of
    MUL, UMULL, UMLAL then
      -- unsigned, pad with 0s
      ba <- zeroExtend (8, 11, b)
    else
      -- signed, sign-extend
      ba <- signExtend (8, 11, b)
    end
  end
end

Figure 5.13: “Sign adjust” example.

derived from a common source, the outputs of the Steer are the only inputs to
the Merge that generates the control token for the next iteration (components labelled [22]). The simplification steps are:

i. The bottom Steer-Merge in figure 5.13(b) can be simplified into a single-output Fork which acts as an adaptor that generates a control token from a data token as shown in figure 5.14(a).

ii. As the generation of the control token in the new Fork is independent of the data value, the data channels that carry the constants can be simplified into control channels, making the constant blocks redundant. These are simplified in figure 5.14(a).

iii. Now the control tokens forked from the outputs of [2E] and [5E] are redundant because each one will always synchronise with its sibling data token at the
Join \( c_1 \), hence those Forks and the Merge and Join with labels \( c_1 \) can be reduced. The new single-input Fork inserted in step (i) can also be removed. The final circuit is shown in figure 5.14(b).

![Figure 5.14](image)

Figure 5.14: “Sign adjust” circuit: (a) first optimisation steps, (b) final circuit.

### 5.3.4 Removing “go” cycles

In Teak circuits, “go” cycles (loops) occur when the description specifies loop constructs (as in figures 5.2(a), 5.7, 5.12 and 5.13). A single initial “go” control token is introduced through a Merge component and the subsequent “go” tokens are locally generated when the circuit produces its outputs.

In the case of unbounded repetition loops, this control cycle can be removed if it does not contain a Steer-Merge (conditional) composition. This means that new “go” tokens for the loop are generated unconditionally.
Consider the example of the $N$-bit full adder introduced in section 5.3.1 whose code is shown in figure 5.9, and its optimised Teak circuit shown in figure 5.15(a). Clearly, the generation of the next "go" token is unconditional in this circuit. The components used to reinsert the "go" token can then be removed safely, leading to the circuit shown in figure 5.15(b). In this particular case, the circuit consists only of data channels. In fact, the optimised circuit ends up having the structure of a fully data-driven pipeline.

If a cycle contains a Steer-Merge composition, there is a possibility of inserting tokens in the wrong order through the Merge if the number of tokens in the cycle is not limited by the "go" circuitry. This is so because, as explained in section 5.2.1, channels in Teak are allowed to have any amount of storage and components can be implemented with any degree of input to output channel coupling. The circuits with conditionals inside a loop construct of the previous sections (e.g. figures 5.12 and 5.13) are examples of circuits with irremovable "go" cycles. In these cases, the number of tokens is limited to one and they are referred to as single-token cycles in the rest of this thesis.
5.4 Description-level optimisations

Teak synthesis use a different set of components and composition strategies than those used in Balsa. It is not therefore surprising that not all of the description-level strategies presented in chapter 4 will be as effective in Teak. In this section, Teak-specific optimisations will be introduced and their impact on the resulting circuit will be analysed.

5.4.1 Commonalities with Balsa optimisations

Teak descriptions also benefit from the data-driven style description introduced in section 4.3, and the following optimisations also apply to Teak: separation of actions into concurrent loops, adding pipeline registers, explicit duplication, and guard optimisation. However, the enclosure techniques (based on pull structures) used widely to speed-up Balsa descriptions may result in poor performance when compiled into Teak push-based circuits.

5.4.2 Description techniques to remove Variables

Variables in Teak are used for implementing both permanent storage and channels (see section 5.2.2). They are the most complex and expensive component in Teak, but allow sequential, storage-centric descriptions to be mapped directly into hardware, avoiding some of the restrictions of not having such a component, as explained in section 5.2.1.

Variables with unconditional reads can be removed as described in section 5.3.1. However, Variables used in the implementation of channels with conditional reads cannot be removed, although in some cases descriptions may be rewritten to avoid conditional reads and therefore allow the variables to be removed.

Avoiding Variables associated with conditional reads

Conditional channel reads occur when a channel access encloses a conditional construct within which the channel’s value is used. Figure 5.16(a) shows an example description of a two-output demultiplexer. In the example, the (write) access to channel $i$ encloses two conditional reads on this channel. In the resulting Teak circuit, in figure 5.16(c), the write and read sections of channel variable $i$ are separated by a Steer and cannot be optimised because of the conditional reads.
If the target were a Balsa handshake circuit, this description would have the advantage of triggering the control to access the pull channel \( i \) early. In Teak push channels, the arrival of valid data initiates the handshake and so no early
activation can occur.

In Teak it is more advantageous to access channel $i$ inside the conditional block, as shown in the code of figure 5.16(b). In this description, paired write and read accesses to channel $i$ are not separated by a conditional and the variable implementing $i$ can be removed. The Teak circuit is shown in figure 5.16(d). Input $i$ is tagged according to the value of $\texttt{ctrl}$ before it is Steered to the required destination.

The cost associated with this style is the extra $\text{Steer-Operator(constant)-Merge}$ structure required to generate the tags, but in general the benefits of not having $\text{Variables}$ compensates this overhead, as will be shown next and in the examples of chapter 7.

**Discarding inputs conditionally**

A similar situation can occur when inputs need to be conditionally discarded (that is, the data token is consumed but not used in any operation). Consider the description of a two-input multiplexer shown in figure 5.17(a). The specification is such that both inputs are always expected and one of them must be discarded. The description in figure 5.17(a) has been optimised to generate optimised Balsa handshake circuits. In the following, this coding style will be referred to as $\text{Balsa-optimised}$. In Balsa handshake circuits, the resulting input structure ensures that all inputs have arrived before completing the enclosing handshake, although the output is generated as soon as the selected input is present. The unused input is implicitly discarded by the input control structure. However, when compiled into Teak circuits as shown in figure 5.17(c), this Balsa-optimised style generates conditional channel reads that prevent the removal of the associated channel $\text{Variables}$. In the description optimised for Teak circuits shown in figure 5.17(b) the inputs are read (and discarded) inside the conditional block. This creates channel $\text{Variables}$ that may be removed. The resulting optimised circuit without channel variables is shown in figure 5.17(d). This style of description targeting the optimisation of Teak circuits will be referred as $\text{Teak-optimised}$. Notice in the Teak-optimised circuits that, because all read accesses to channel $i$ are now done inside the conditional construct, it has been necessary to generate steering tags for each input (to tag them with either “pass” or “discard”). Finally,
Section 5.4 Description-level optimisations

procedure mux2 ( 
  input ctl : bit; 
  input i0, i1 : N bits; 
  output z : N bits 
) is 
begin 
  loop 
    i0, i1, ctl ->! then 
      if ctl then 
        z <- i1 
      else 
        z <- i0 
      end 
    end 
  end 
end 

Figure 5.17: Discarding inputs conditionally in Teak: (a, c) Balsa-optimised style; (b, d) Teak-optimised style.
another steering tag is generated from the “passing” value to rendezvous the token from the “discarded” item.

Another possible optimisation is shown in figure 5.18(a): before using the inputs inside the conditional block, they are explicitly joined into the single channel \( i01 \) (lines 8 - 11). Inside the conditional structure and, for each condition, the relevant bits of this channel are passed to the output.

```plaintext
procedure mux2 (
  input ctl : bit;
  input i0, i1 : N bits;
  output z : N bits
) is
  channel i01 : 2*N bits
begin
  loop
    i0, i1 -> then
    i01 <- (#i0 @ #i1 as N bits)
  end loop
  ctl -> then
  if ctl then
    i01 -> then
    z <- (#i01[0..N-1] as N bits)
  end
  else
    i01 -> then
    z <- (#i01[N..2*N-1] as N bits)
  end
end
```

Figure 5.18: Joining inputs to reduce the tagging circuitry.

Figure 5.19 shows speed (processing time), area and dynamic energy comparisons for the three multiplexer designs presented above, for different data widths (the fourth, dotted bar in each series, labelled Circuit-level, will be introduced later in this section). In all the simulation results presented in this section the circuits are connected to an environment that is always ready to provide data in all the inputs simultaneously. Random data values were generated for data inputs whereas select control values were generated such that all options were equally exercised.

The results in the graphs of figure 5.19 show that, for data widths \( \geq 4 \),
the joined-inputs optimisation delivers the fastest speed with area and energy consumption smaller or comparable to that of the Teak-optimised style. The results also show that, compared to the Balsa-optimised, the Teak-optimised style is advantageous for wider datapaths, when the overhead of tagging is smaller compared to the cost of the wider Variables (their associated completion detection circuitry becomes larger and slower as the number of bits increase). For the optimised circuits, the speed-up is directly proportional to the data width whereas area and energy penalties are inversely proportional.

The joined-inputs optimisation is less effective in circuits where there is a large difference in the arrival time of the inputs because in order to generate an output, both inputs must be present. In the first optimisation an output may be produced with only one input present and so input synchronisation is only required for the RTZ phase.

**Duplicating values to avoid conditional channel reads**

In cases when multiple, non-mutually exclusive conditional reads can occur, it is necessary to explicitly duplicate some of the channels to get rid of the Variables. The SteerAlu module from the nanoSpa Execute stage is shown in figure 5.20.

This module multicasts the ALU result to a set of destinations depending on the bits of the ctrl input. The set of destinations may be empty, in which case the ALU result is discarded. The Balsa-optimised description is shown in figure 5.20(a) and the resulting Teak circuit in figure 5.21.

The Teak-optimised version is shown in figure 5.20(b) and the resulting circuit in figure 5.22. In this case, to avoid the conditional channel reads, the input has been explicitly duplicated (one copy for each condition) and, in a similar way to the demultiplexer example, each copy is either passed or discarded but it is always read.

Figure 5.23 shows speed (processing time), area and dynamic energy comparisons for the two SteerAlu descriptions, using various data widths. It can be seen in the graphs that the Teak-optimised circuits are \( \sim 30\% \) to \( 60\% \) faster, more energy efficient and with an area penalty inversely proportional to the data width. As explained earlier, as the data width increases, the overhead of a wider Variable becomes larger compared to that of the tagging circuitry.

It is clear from the examples that the overhead of the tag-and-steer mechanism will increase with the number of inputs and conditions involved. In fact, when
complex nested conditions occur, a variable-free description may result in a large, nested tag-and-steer circuitry which will result in area overhead with insignificant speed-ups. In such cases, it is not advisable to apply the above techniques.
A circuit-level approach to remove conditional channel reads

The above description-level optimisation examples have shown that in order to remove Variables in conditional structures (i) tags derived from the guard token must be added to each data token and (ii) copies of each data token must be produced for each non-mutually exclusive conditional read. The result is always a number of tagged data tokens that will be Steered accordingly.

The optimised structures suggest a new circuit-level optimisation opportunity to get rid of the Variables without having recourse to the directness of the compilation. This new optimisation is based on the data steering property of the Steer component: Steer uses a subset of the input bits as the output selector and it passes a subset of the input bits to the matching output. Instead of appending a tag generated from the guard, it is possible to append the actual guard and modify the Steer specification to use directly this value, simplifying the Teak
Figure 5.21: **steerAlu** Teak circuit.
Figure 5.22: Optimised `steerAlu` Teak circuit.
network. The cost of this approach is in the increased complexity of the Steers required and in the complexity of the rules to determine situations where the transformation may be applied.

To illustrate the proposed mechanism, let us revisit the circuit for the two-input multiplexer, reproduced again in figure 5.24(a). In this figure, channels 10,
i1 and ctrl are joined and stored into the channel Variable i0-i1-ctrl. The w0 (write done) token generated by the w0 portion activates the read portion r2, which provides the bits corresponding to the ctrl guard only.

The Steer that implements the conditional generates zero-width control tokens to activate one of the read portions r1 or r0, which provide the values of i1 or i0, respectively. If these portions are displaced upstream through the Steer, they can be combined with portion r2 into a single read portion that will provide all of the bits of the composite channel i0-i1-ctrl. The specification of the Steer must be modified accordingly to accept this wider value at its input and to steer the required portions to its outputs.

The above modifications are shown in the circuit of figure 5.24(b). In this circuit, Variable i0-i1-ctrl is unconditionally read and can be removed, as shown in figure 5.24(c).

Notice that the new optimisation does not require the tagging circuitry, but the specification of the Steer will be more complex. In this particular case, the resulting Steer-Merge combination cannot be removed because the offsets of the two Steer outputs are different (they correspond to the i0 and i1 sections in the composite channel i0-i1-ctrl).

If the write and read portions of a variable are separated by a Fork, as in the steerAlu example of figure 5.21, a further combination is required when the individual portions are displaced through the Fork, which in turn must also be modified accordingly. This is the equivalent of the duplication mechanism used before at the description level. Figure 5.25 shows the resulting optimised circuit for the steerAlu example. Again, no tagging circuitry is required but the Steers will end up being more complex.

Simulation results for these hand-applied transformations on the multiplexer and steerAlu examples are shown in figures 5.19 and 5.23 under the key Circuit-level (dotted bars). The results show that this optimisation produces circuits that are considerably faster (∼30% to 50%) and more energy efficient (∼5% to ∼50%) than the circuits produced with the description-level optimisations.

Notice in figure 5.23(a) how the more complex Steers used in the steerAlu example increase the area penalty as the data width increases.

The rules for the above transformations must check a number of conditions of the components surrounding the write and read portions, some of which have been highlighted in the examples:
Figure 5.24: Circuit-level conditional reads removal.

- the write portions of two different variables must be separated by a channel or by *Joins*.

- the write and read portions must be separated by a channel or by *Forks*. In the latter case, the *Fork* outputs must be modified accordingly to accommodate the read portions.

- the read portions that provides the selection must be separated from the selected read portions by *Steers*.

*Variables* with multiple write portions will make the transformation rules even more complex because a larger window of components will need to be checked. Furthermore, it is not always desirable to get rid of variables because they may form part of the specified behaviour. The procedure-level mechanism for passing optimisation options in section 6.5 is useful for this purpose, although a finer,
Figure 5.25: Circuit-level optimisation of the \texttt{steerAlu} module.
structure-level mechanism is envisaged. The proposed transformations demonstrates that there is still room for further optimisation of Teak networks.

5.4.3 Summary

This chapter has introduced Teak as a novel approach towards the synthesis of asynchronous circuits using a token-flow approach together with a set of optimisation techniques for the resulting networks. The basis of this system is a small set of components that provide basic datapath operations. Teak shares the push-only data style of the data-driven style proposed by Taylor [100], although Teak compiles Balsa descriptions and is more similar to the Macromodules system [93] than handshake circuits. Another difference with the data-driven handshake circuits is the availability of “real” Variables as permanent storage elements that permits flexible read and write accesses.

The properties of the Teak components and its compositions were used as the basis for optimisation techniques which are based in (i) circuit transformation, like the Fork, Operator and Join displacement, (ii) pattern-matching and replacement (like the loop removal and the Variable substitution, or (iii) a combination of transformation and substitution, like the Steer-Merge-Join optimisation.

Description-level techniques aimed specifically to this approach were also presented. In particular, it was noted that the enclosure technique used to optimise Balsa handshake components implementation may introduce performance overhead in Teak circuits. The description-level techniques target the removal of channel Variables with conditional accesses. The principle of these techniques were used as the basis to a more efficient circuit-level transformation that exploits the data-steering property of the the Steer component.

As will be seen in the chapter 7, for large and complex designs like the nanoSpa processor, implementations of Teak circuits currently have worse performances than those of Balsa circuits. This is unfortunate but not unexpected. The implementations for Teak components are at an early stage of development. However, there is a lot of headroom within the Teak approach as its small, regular component set allows the freedom to merge and split data and control much more naturally than in handshake components.

There is still much work that can be done to improve the optimisation of Teak-generated circuits. This includes: improved component implementations, the implementation of components with different data encodings (e.g. one-hot...
codes running up to *Steer* inputs) as well as extensions to the current optimisation and automation of the optimisation described in section 5.4.2. The important optimisation issue of latch insertion in Teak circuits is considered in the next chapter.
Chapter 6

Latch insertion in Teak circuits

6.1 Introduction

Teak channels can be buffered to decouple components and to introduce the desired degree of token storage. In the Teak system, Latch components are used for buffering purposes. The Latch components used in the circuits presented in this thesis are implemented as half latches [91, 17]. Although other implementations are possible, the half latch implementation was selected for its simplicity.

The Teak synthesis algorithm does not introduce any buffering initially, this allows optimisation techniques to explore different buffer placement strategies. When the network contain cycles (also referred to as rings or loops in the related literature) buffers must be inserted in order to prevent deadlock. Latches may be inserted into any circuit to decouple processes and increase throughput.

A simple insertion strategy is to add a Latch to every channel. This will add enough token storage to prevent any circuit from deadlock but has a high penalty. To optimise the circuit’s latency and throughput more elaborate buffer insertion strategies must be used. In [115, 114], Williams and Horowitz introduced some basic concepts and metrics to characterise the performance of asynchronous pipelines and rings. Based on their work, some approaches have been proposed to increase the performance of pipelined asynchronous circuits through latch insertion and slack matching [7, 45, 46]. Those approaches target iterative circuits with cycles that may contain more than one token and that can benefit from pipelining inside the cycle.

This chapter introduces a range of latching strategies currently implemented in the Teak system. The strategies target cycle structures that can hold a maximum
of one token (single-token cycles) commonly present in Teak circuits and that do not benefit from cycle pipelining. The aim of the strategies proposed here is solely to provide a more efficient alternative to the exhaustive insertion mechanism, although some analysis on the complexity and resulting performance is presented. The strategies are based on:

i. the identification and minimum latching of cycles.

ii. separation of tokens to avoid WAR hazards between portions of Variable components.

iii. the correct decoupling of control tokens in parallel and sequential compositions.

Optimal latch insertion targeting area/speed and pipeline slack matching is outside of the scope of this work.

### 6.2 Buffering cycles

Cycles in Teak circuits occur when the description specifies loop constructs, but also when modules are connected together in a ring fashion. In order to allow the circuit to progress, each cycle must have always enough buffering for a lead token to move forward and leave space for the following token. This translates into having a minimum of three half latches in a cycle [114, 91].

The most common single-token cycle structure in Teak circuits is the Merge - Logic Block - Fork circuit shown in figure 6.1, which can be clearly seen in some of the previous examples. Often, within the logic block of such structures there will be some latches required to separate the read and write tokens of Variable components.

![Figure 6.1: The Teak single-token loop Merge - Logic block - Fork structure.](image-url)
6.2.1 Detecting cycles

In the first step of the analysis the circuit is mapped into a directed graph, where the edges are ordered pairs \((s, d)\), connecting a source vertex to a destination vertex, as in figure 6.2(a). Within the graph each Teak component is mapped into a vertex and each channel into an edge. For Variable components, write and read sections are mapped into separate vertices.

![Directed graph](image)

**Figure 6.2:** (a) A directed graph and (b) a depth-first forest of the graph.

The resulting graph is then analysed using techniques based on depth-first search (DFS) [59, 98, 24] to obtain a classification of the edges. Initially all of the vertices of the graph are set to “unvisited”. The DFS algorithm begins by choosing one unvisited vertex (a root) and exploring an edge leading to a new vertex. The algorithm continues in this fashion until it reaches a vertex which has no edges leading to unvisited vertices. The algorithm will then backtrack to the previous vertex and continue from the latest vertex that does lead to new unvisited vertices.

After DFS has visited all the reachable vertices from a particular root vertex, it chooses one of the remaining unvisited vertices as a new root and continues the search. The DFS process creates a set of depth-first trees that constitute a depth-first forest. The edges of the resulting forest are classified as tree edges (edges which lead to unvisited vertices), forward edges (edges which connect ancestors with descendants in a particular tree), back edges (the ones that connect descendants with ancestors), and cross-edges (which connect vertices across the forest). Figure 6.2(b) shows one possible DFS forest and the different classes of edges of the directed graph at its left.
The interesting class of edges for cycle detection are the *back edges*, because each back edge closes one or more cycles. Typically there are many valid depth-first forests for a given graph, depending on the (arbitrary) selection of the initial root and subsequent unvisited root vertices. There are therefore many different (and equally valid) resulting classifications for the edges. In Teak networks, the best candidates for root vertices are the components connected to the “go” and input ports, in that order of priority. This selection and priority is based in the following observations derived from compiled circuits:

i. A Teak network with a “go” port will always map into a *connected graph* (a graph such that there exists at least one path between all pairs of vertices). Using the component connected to “go” as a root vertex will ensure that all vertices are visited. It also ensures that the channel that returns the control token for the next iteration will be classified as a back edge. This is the most “natural” classification for such channel and also prevents the selection of a wider data channel as the back edge of the cycle, which would be more expensive to buffer.

ii. An optimised Teak network without a “go” port may map into a *disconnected* graph, consisting of two or more connected sub-graphs. Selecting vertices connected to input ports as root vertices will ensure that all vertices will be visited.

iii. If the description contains explicit ring structures, selecting the modules connected to the input ports as root vertices will ensure that the outputs that feed back and complete the ring will be classified as back edges.

Figure 6.3 shows the Teak circuit of figure 5.12(b) and its mapping into a directed graph. Notice that input and output channels are not included in the graph, but they are used in the selection of vertices as explained above.

Figure 6.4 shows the forest that results from applying a DFS analysis to the graph in figure 6.3. The graph has three back edges, \{(L,A), (G,I), (J,K)\}. These generate four cycles, namely, \{A, B, D, H, J, K, L\}, \{A, B, E, I, K, L\}, \{C, F, H, J, K, L\} and \{C, G, I, K, L\}. Each cycle \(c\) comprising \(v_c\) vertices has \(e_c = v_c + 1\) edges. In this example, the cycles have 7, 6, 6 and 5 edges, respectively.
Figure 6.3: Mapping of a Teak circuit into a directed graph.

Figure 6.4: DFS forest of graph in figure 6.3(b).

6.2.2 Complexity of finding the optimum latch insertion points

For the simple example above it is not difficult to find by observation that the minimum set of edges that guarantees at least three latches within each cycle is:
{(L, A), (L, C), (I, K), (J, K), (K, L)}. However, in order to obtain the best possible location of the latches (targeting either minimum area or latency) each edge of the graph must be assigned a cost function depending on the type of components connected by the edge and the channel width. All possible arrangements of three latches within each cycle must then be enumerated and a cost assigned to each.

The number of possible arrangements for three latches in a cycle is given by the combinatorial number \( \binom{v}{3} \). For the previous example, the total number \( C \) of different arrangements to be examined would be:

\[
C = \binom{7}{3} \binom{6}{3} \binom{5}{3} = 140,000
\]

In practice, for medium or large circuits, the number of components and cycles in a circuit makes an exhaustive analysis to find the optimum insertion points infeasible. Added to the complexity of finding all possible arrangements for the three latches, is that of finding all the cycles. In [99], Tarjan demonstrated that the complexity of finding all the cycles (referred to as elementary circuits in his work) in a graph with \( v \) vertices, \( e \) edges and \( c \) cycles is \( O(v \cdot e(c + 1)) \). An optimised algorithm [53] reduces this complexity to \( O((v + e)(c + 1)) \). However, the optimisation excludes cases that may occur in Teak circuits, like self-loops (edges of the form \((v, v)\)) and multiple edges between the same vertices.

As it can be seen in the previous examples, Teak circuits normally comprise Fork-Join and Steer-Merge “diamonds”. Each \( n \)-branch diamond located inside a cycle multiplies the number of possible cycles by \( n \). This implies that for large circuits, the complexity of finding all cycles is too high, not to mention the combinatorial explosion of finding all possible latch placements. Consider for instance the optimised Teak circuit for the GCD shown in figure 6.5, derived from the description previously shown in figure 4.14(b). The directed graph from this circuit will have 41 vertices, 51 edges and 36 cycles, resulting in a complexity of \( O(77\,367) \) for finding cycles, which appears manageable. However, determining the best latch placements is more complex.

The GCD circuit has 8 cycles with 15 edges, 16 cycles with 16 edges and 8 cycles with 17 edges, hence, the number of possible combinations for latch placement is:

\[
C = \left( \binom{15}{3} \right)^8 \left( \binom{16}{3} \right)^{16} \left( \binom{17}{3} \right)^8 \simeq 7.85 \times 10^{87}
\]
Figure 6.5: Optimised Teak circuit for the GCD description in figure 4.14(b)
In order to efficiently determine latch placements, heuristics are required to reduce the complexity of the problem. At present, the approach implemented in Teak is to use simpler strategies to latch token-limited cycles with the minimum three latches located in places that guarantee a deadlock-free cycle and a fast decoupling of the cycle outputs.

In the current implementation of Teak, as well as the insertion of three latches in every cycle, the user can specify the insertion of an arbitrary number of latches to decouple Operators, read sections of Variables, placed in forked control tokens, or placed in every channel. These latching strategies can be specified for the whole design or in a module-by-module basis. The next section will discuss issues related to the strategy used to automatically latch token-limited cycles.

### 6.3 Buffering single-token cycles

Using the DFS analysis together with the root selection rules described in section 6.2 ensures that the channel used to return the control token in a cycle is classified as a back edge. This is the result of selecting the vertex connected to the “go” (a Merge) as the first root. Because every cycle contains at least one back edge, inserting one latch in the following places will ensure that all the cycles containing the back edge \((s_i, d_i)\) will have at least three latches:

- every back edge \((s_i, d_i)\)
- every edge ending at \(s_i\)
- every edge beginning at \(d_i\)

The insertion is optimised to avoid inserting multiple latches in the same edge. This strategy is illustrated in figure 6.6 using the circuit for the GCD. The above heuristics reduce the complexity of the latching to the complexity of the DFS used to find the back edges, which is \(O(v + e)\) [59], plus the processing of each back edge, resulting in a complexity of \(O(v + e + b)\), where \(b\) is the number of back edges in the graph.

This approach efficiently solves the problem of combinatorial explosion, but it does not guarantee the optimal placement for performance or minimum area. Depending on the topology of the circuit, some extra latches may be added and
Figure 6.6: Strategy for latching all cycles of the GCD circuit of figure 6.5 based in latching back edges.
some cycles may end up having more than three latches, as shown in figure 6.6. However, some of the additional latches can be used to fulfil other latching requirements, such as the token separation latches for read and write sections of Variables.

Two variants of this approach will be analysed in the next section. Although the strategies target single-token cycles (those generated by loop constructs), they also serve to guarantee deadlock-free operation for multi-token cycles. In any case, the most important parameter is the time elapsed between iterations (the cycle time).

6.4 Two simple latching strategies for Teak circuits

This section analyses and compares two strategies to arrange three latches in single-token cycles. The strategies use the heuristics of attempting to place the latches as close as possible to the inputs and outputs to provide them with fast decoupling. These two strategies are:

i. placing a latch in the back link and after each back link successor, and before each predecessor. This arrangement is shown in figure 6.7(a). This strategy will be called “A”.

ii. distributing the three latches so that the delay of the logic block is evenly split among smaller blocks, as shown in figure 6.8(a). This is strategy “B”.

The figure of merit to evaluate each strategy is the cycle time, which will be determined by using a dependency graph analysis [91, 114]. A dependency graph represents the dependencies between signal transitions in a circuit. The vertices of such a graph represent rising or falling transitions and the edges represent dependencies between the signal transitions. In the analysis presented here, dependencies are represented as directed arcs and transitions are represented with boxes annotated with an internal label denoting the transition name and an external label denoting the delay associated with the transition.

Figure 6.7(b) and (c) shows the logic circuit and the corresponding dependency graph for the strategy “A”. Similarly, figure 6.8(b) and (c) shows the logic circuit and the corresponding dependency graph for the strategy “B”. In the diagrams, $t_i$, $t_c$ and $t_{cd}$ represent the latencies of an inverter, a C element and an
Figure 6.7: Latching strategy “A” for single-token M-LB-F blocks: (a) Teak circuit, (b) logic circuit, (c) dependency graph.
Section 6.4  Two simple latching strategies for Teak circuits

Figure 6.8: Latching strategy “B” for single-token M-LB-F blocks: (a) Teak circuit, (b) logic circuit, (c) dependency graph.
n-bit completion detector respectively. To simplify the analysis, the following assumptions have been made:

i. all circuit components have symmetric delays for rising and falling transitions.

ii. there are \( N - 1 \) latches inside the logic block which evenly split the delay of the block by \( N \). These are the latches associated with the read and write decoupling of \( \text{Variable} \) components inside the block.

In the current dual-rail implementation, \( \text{Operator} \) components have zero backward latency (the latency for the acknowledge), whereas other Teak components have some backward latency depending on the data width (\( \text{Merge} \)) or the number of outputs (\( \text{Steer}, \text{Fork} \)). The forward and backward latency of the logic blocks are labelled \( t_{lbf} \) and \( t_{lbr} \) respectively. The latency of the environment (typically another Teak circuit connected to the outputs) has been included in the circuits and graphs and is denoted as \( t_{env} \).

### 6.4.1 Analysis of the latching strategies

The longest simple cycle for the latching strategy “A” has been highlighted in the dependency graph of figure 6.7(c). Starting from transition \( R1 \uparrow \) at the left of the graph, and following the highlighted path, the cycle time is:

\[
\begin{align*}
t_{\text{cycleA}} &= 2t_i + (N + 1)t_c + N\left(\frac{t_{lbf}}{N}\right) + t_{env} + 6t_c + 4t_i + t_{cd} \\
t_{\text{cycleA}} &= 6t_i + (N + 7)t_c + t_{cd} + t_{env} + t_{lbf}
\end{align*}
\]

Assuming that the latency \( t_c \) of a C-element is equivalent to two inversions,

\[
t_{\text{cycleA}} = (N + 10)t_c + t_{cd} + t_{env} + t_{lbf}
\]

\( (6.1) \)

Similarly, for strategy “B”, starting from transition \( \text{Req}1 \uparrow \) at the left of the graph in figure 6.8(c), the cycle time is:

\[
\begin{align*}
t_{\text{cycleB}} &= (N + 1)t_c + N\left(\frac{t_{lbf}}{N}\right) + t_{env} + 4t_c + 4t_i + 2t_{cd} + \frac{1}{N}t_{lbf} + \frac{1}{N}t_{lbr} \\
t_{\text{cycleB}} &= 4t_i + (N + 5)t_c + 2t_{cd} + t_{env} + (1 + \frac{1}{N})t_{lbf} + \frac{1}{N}t_{lbr}
\end{align*}
\]
Assuming that the latency of a C-element is equivalent to two inversions,

\[
t_{cycleB} = (N + 7)t_c + 2t_{cd} + t_{env} + (1 + \frac{1}{N})t_{lb} + \frac{1}{N}t_{lr}
\]  

(6.3)

For \( t_{cycleA} \) to be the shortest, \( t_{cycleB} - t_{cycleA} > 0 \). From Eqs. 6.2 and 6.3,

\[
(N + 7)t_c + 2t_{cd} + t_{env} + (1 + \frac{1}{N})t_{lb} + \frac{1}{N}t_{lr} - ((N + 10)t_c + t_{cd} + t_{env} + t_{lb}) > 0
\]

\[-3t_c + t_{cd} + \frac{1}{N}t_{lb} + \frac{1}{N}t_{lr} > 0
\]

\[t_{cd} + \frac{1}{N}(t_{lb} + t_{lr}) > 3t_c
\]

(6.4)

The inequality 6.4 will hold for all cases where there is at least one latch inside the logic block. The inequality is independent of the environment latency. Looking closely at the path in figure 6.7(c), it can be noticed that the latch in the back edge decouples the logic block during the RTZ phase (the path does not go through the logic block), reducing the cycle time.

In cases when the logic block contains no internal latches (that is, no variables), inequality 6.4 no longer holds. In these cases, if the strategy “B” is used, the second inserted latch (\( L2 \) in figure 6.8) will split the logic block into two halves, forcing \( N = 2 \) in equation 6.3. However, for strategy “A”, \( N \) will be equal to 1 because the inserted latches are always outside the logic block. With these conditions, there will be another longest cycle candidate for strategy “A” (the “eight” shaped dotted line in figure 6.7). Starting from transition \( Req1 \uparrow \) and following this new path,

\[
t_{cycleA1} = 5t_c + 2t_{cd} + \frac{2}{N}t_{lb} + \frac{2}{N}t_{lr}
\]

\[
t_{cycleA1} = 5t_c + 2t_{cd} + 2t_{lb} + 2t_{lr}
\]

(6.5)

Substituting \( N = 2 \) in equation 6.3 and performing the required operations (again, assuming \( t_{cycleA} \) to be the shortest):

\[
t_{env} > \frac{1}{2}t_{lb} + \frac{3}{2}t_{lr} - 5t_c
\]

(6.6)

In this case, strategy “A” will produce a faster circuit unless inequality 6.6 does not hold, that is, when the environment is faster than the delay through the
Chapter 6 Latch insertion in Teak circuits

logic block. As an example, let us consider the situation when the logic block has the equivalent latency of two adders (addition is the slowest operation in Teak). Substituting $t_{env} = t_{cd}$, $t_{bf} = 2t_{adder}$ and $t_{br} = 0$ in Eq. 6.6, the condition for strategy “A” delivering the faster circuit is:

$$t_{cd} > t_{adder} - 5t_{c}$$

(6.7)

Figure 6.9 shows a plot of both sides of inequality 6.7 as a function of the data width. The values shown are based on a worst-case longest carry chain of $width/2$. This is a conservative scenario, as in practice, the average carry chain length is less than $width/2$ [41, 62]. Figure 6.9 compares the left and right sides of inequality 6.7. The results are for a library with 2 and 3-inputs C-elements. The plot shows that, in this scenario, the inequality 6.7 will hold for $width > 16$ bits.

Figure 6.9: Comparison of both sides of inequality 6.6 for different data widths.

Experience with medium and large design examples used in this thesis have shown that complex, slow logic blocks with no variables are uncommon. In summary, the analysis presented in this section shows that, for practical cases, it is safe to assume that strategy “A” will produce a circuit with smaller cycle time. This is the strategy currently used in Teak. An additional benefit of using this strategy is that decomposition of the logic blocks into two equal parts when it contains no latches is not required.
6.5 Specifying latching and optimisation options in Teak

The Teak system provides two mechanisms to specify latching and optimisation options: (i) a command-line mechanism to specify the global, default options and (ii) a coarse-grained mechanism to specify local optimisations, at procedure-level, that overrides the global options. These mechanisms provide a flexible way of specifying and exploring the optimum set of options for a design.

The current implementation of the procedure-level mechanism is an extension to the Balsa language. If desired, local options can be passed enclosed in the (*) pair, after the port declarations, as illustrated in line 5, figure 6.10. The opts label is used to pass optimisation options and the latches label is used for latching options. Options are separated by a colon (:). In figure 6.10, trim-vars specifies Variable removal and move-fork-tos is a Fork displacement optimisation. The latching option 11 specifies the insertion of three single latches on each cycle of the circuit.

```plaintext
procedure adder ( 
  input a, b : N bits; 
  output sum : N bits; 
  output carry : bit 
) (* opts="trim-vars:move-fork-tos" latches="l1" *) 
is 
  channel cs : N+1 bits 
begin
```

Figure 6.10: Example of passing options at procedure-level.

6.6 Summary

The problem of inserting latches in Teak circuits has been introduced in this chapter. An estimation of the complexity of efficiently buffering circuits to avoid deadlock was presented. The estimation was based on the number of cycles presented in the circuit, which require at least three half-buffers to allow the circuit to progress. In order to find the cycles, the Teak circuits are mapped into directed graphs and then analysed using a depth-first search technique that makes use of some properties of the Teak networks.
Two techniques that implement a minimum latching scheme for the single-token cycles (that results from the synthesis of the loop construct) were introduced and analysed. The techniques are presented as a more efficient alternative to the exhaustive latch insertion. The techniques make use of simple heuristics (fast decoupling of input and output ports) in order to reduce the complexity of the latch placement problem. The described techniques are the basis of the automatic latching insertion available in the Teak system.

Efficient buffering of Teak circuits targeting performance remains an open issue, as more heuristics based on the structure of Teak networks are required. The aim of the work presented in this chapter was solely to provide the Teak system with a minimum latching strategy to allow the circuits to operate. Latching insertion targeting performance or area/energy efficiency are considered future work.
Chapter 7

Design Examples and Evaluation

This chapter presents the descriptions and simulation results of a series of substantial design examples that were used to evaluate the different techniques presented in this work. The examples include:

- A 32-bit processor core: Nan’s.
- A Viterbi decoder.
- A $32 \times 32$ radix-8 Booth multiply-accumulate (MAC) unit.
- A new result forwarding unit for the nanoSpa processor.
- A sliced-channel wormhole router.

All of the above designs were evaluated using the Balsa synthesis system. In Teak, the nanoSpa processor, the Viterbi decoder and the multiplier were used as evaluation examples. The forwarding unit and the router were not used as Teak examples because they rely on constructs and operations based on sequencers whose timing assumptions are not easily translatable into the Teak approach without a complete rewrite of the most complex parts of their descriptions.

All results given in this chapter were obtained using pre-layout, transistor-level simulations, using a 180 nm technology.

7.1 The nanoSpa processor

The nanoSpa processor [85] is an updated specification of the SPA processor [84], an asynchronous implementation of the 32-bit ARM v5T ISA [51] fully synthesisable using the Balsa system. The nanoSpa uses highly optimised Balsa code
targeting higher performance as opposed to SPA, whose description focused on security.

The initial version of nanoSpa [85] shares the same architecture organisation as SPA: an ARM-style 3-stage Fetch-Decode-Execute pipeline with a Harvard-style memory interface. The initial version had the following functional differences with respect to SPA:

- no support for Thumb instructions, interrupts, memory aborts or coprocessors.
- only the supervisor and user operation modes were available.
- no support for multiply operations.
- no support for half-word data transfers.

The new nanoSpa specification includes major changes in the organisation of the Decode and Execute pipeline stages, oriented to achieve its performance goal. These new features will be described in the next sections. The author has contributed to develop this new version with the following enhancements:

- implementation of the modified Decode stage.
- support for all ARM multiply instructions with the 32x32 MAC unit described in section 7.3.
- support for all ARM modes of operation.
- support for half-word data transfers.
- a branch control mechanism to reduce branch shadow penalties.
- the result forwarding unit described in section 7.4

Figure 7.1 shows a diagram of the 3-stage nanoSpa pipeline.

7.1.1 The Fetch stage

The Fetch stage fetches instructions from memory and implements the changes to instruction address flow generated by branches. This unit is very similar to the SPA fetch unit and has not had major changes. Like in SPA, the origin of the
fetch address must be arbitrated between the local generated sequential address and the branch target address. This is the only place where arbitration is required in nanoSpa.

### 7.1.2 The Decode stage

The Decode has been redesigned as a two-level modular decoder as shown in figure 7.1.

**Decode shell**

This module receives the fetched instruction and performs an initial decoding of the instruction, generating the control signals which are common to all instruction types. It also selects the appropriated fields from the instruction to be used by the next level: the decode core.
Decode core and decoding modules

The decode core classifies the instruction according to its type and activates the related decoding module. There is one decoding module for each instruction type. These modules expand the instruction into the control and register selects required by the execute stage. The modules for multi-cycle instructions (such as load and store of multiple registers) unroll the instructions and issue all the required signals multiple times to the execute unit. In Balsa, each decoding module is described as an individual procedure. This modular approach makes it easier to either modify the decoding modules or add new ones, as any additions or changes are almost transparent to the rest of the already decoded signals.

Branch control counter-flow

The decoder receives branch control information from the execute unit, allowing it to discard any fetched instructions that are not within the new instruction flow established by the execution of a branch (those instructions are said to be in the shadow of a branch). The branch control from the Execute to the Decode unit is especially important in this design because the execute unit features speculative operation. In this way, instructions already in the pipeline that would be discarded after execution because they are in the shadow of a branch are now discarded earlier, increasing the performance and saving power.

7.1.3 The Execute stage

This stage has been redesigned to implement both data-driven and speculative operation, which has improved significantly its performance. The multiplier unit has also been redesigned and it is implemented using a modified radix-8 Booth algorithm customised to support signed and unsigned operands. A complete description is given in section 7.3. Figure 7.2 shows a simplified version of the nanoSpa execute stage. Data-driven and speculative operation in nanoSpa has been presented previously [85]. A brief description of these is given below.

Data-driven operation

In nanoSpa, all units inside the execution stage are activated in parallel: they wait until data arrives, process it and sends the result out without explicit sequencing. Steering and multiplexing units are added to guide the data and are
controlled directly by control signals from the decoder, without any sequencing or synchronisation with data.

**Speculative operation**

In the ARM instruction set all instructions are conditional, i.e., they can be executed or skipped depending on the condition codes. In nanoSpa, the evaluation of the condition codes and the execution of the instruction are carried out concurrently to allow an early start of the instruction. If the condition code fails, the instruction is discarded at strategically located checkpoints without any result being written back, but ensuring that handshaking on all channels is completed. Figure 7.2 shows how kill modules (labelled "K") are used to implement these checkpoints. In nanoSpa, data-processing instructions are started speculatively whereas data memory instructions are not, because of the extremely high penalties in power and performance that could derive from them. Speculative execution will only increase performance if the percentage of executed instructions is high, but this is usually the case.
Chapter 7 Design Examples and Evaluation

7.1.4 Results

This section presents the simulation results for the nanoSpa processor using both the Balsa and Teak synthesis systems. The source description used with both synthesis tools was practically the same, apart from some minor modifications because of the “fire and forget” behaviour of Teak channels explained in section 5.2.2. All simulations results were obtained by running the Dhrystone benchmark.

**Balsa**

Table 7.1 shows the performance, area and energy results of different versions of the nanoSpa processor. The original version is called nanoSpa0 in table 7.1. The device nanoSpa1 includes the redesigned Decoder stage described in section 7.1.2 with the branch control counter-flow mechanism. The device nanoSpaRef (DD) is a description-level optimised and enhanced version of the nanoSpa1 with added support for all the ARM modes, half-word and byte memory transfers, and the MSR and MRS instructions, which read and write the current and saved status registers. A finer grained separation of actions in concurrent loops and explicit duplication optimisation techniques were applied to this description.

The results in table 7.1 serve to differentiate the sources of performance improvement. From the table, the architectural enhancements have improved the performance by 8.92%. The description level optimisations improves the performance close to 27%. Therefore, the description level optimisations have increased the performance of the nanoSpa1 design by 16%.

| nanoSpa device | DMIPS\(\dagger\) | Area elements | Energy\(\ddagger\) | \
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nanoSpa0</td>
<td>57.98</td>
<td>485108</td>
<td>460</td>
</tr>
<tr>
<td>nanoSpa1</td>
<td>63.15</td>
<td>622884</td>
<td>358</td>
</tr>
<tr>
<td>nanoSpaRef (DD)</td>
<td>73.54</td>
<td>662734</td>
<td>361</td>
</tr>
</tbody>
</table>

\(\dagger\) Dhrystone MIPS  
\(\ddagger\) per Dhrystone loop  

Table 7.1: Performance, area and energy for three different versions of nanoSpa.

An interesting side-effect of the optimised architecture of the nanoSpa1 is the reduction in energy consumption. The sources of this reduction are the more efficient design of the decoder and the branch control counterflow mechanism.
which discards instructions that fall in a branch shadow at the decoder stage. Area penalties are the result of the enhanced features.

Table 7.2 shows the performance, area and energy results of different source-code and peephole optimisations presented earlier in chapter 4. The following is a key to the devices included:

- **DD**: the reference design. Corresponds to the *nanoSpaRef (DD)* optimised data-driven description presented in table 7.1.

- **DDO**: description-level optimisation of the *DD* design, with optimised guards and the addition of explicit duplication in some modules of the Execute unit.

- **DD/DDO + CF**: the *DD/DDO* description with the use of the concurrent *Fetch* component in the register bank.

- **DD/DDO + nRFV**: the *DD/DDO* description with removal of redundant *FalseVariables* applied.

- **DDO + nRFV + CF**: the *DDO* description with the use of the concurrent *Fetch* component in the register bank and the removal of redundant *FalseVariables* applied.

- **DDP (Taylor)**: the results for the original *nanoSpa0* architecture presented in [101] using the *push-only* data-driven synthesis methodology. The description is written in the new input language proposed by Taylor in his PhD thesis [100].

The results show that the different optimisations increased the performance between 2.6% to 6% when applied individually, and more than 11% when combined. Also notice that the description-level optimisations are the largest contributor to the performance increase, at 6%. Comparing the DDO results with the nanoSpa1 in table 7.1, the description-level optimisation increased the performance by around 24%. The results also show that, apart for the negligible area increase when using the concurrent *Fetch*, the optimisations result in area and energy reductions of less than 10%.

The *DDP* device was included to compare the trade-offs of having a full data-driven synthesis against the optimisations techniques proposed here. The description used in this thesis is an enhanced implementation of the one used by Taylor. Despite the differences in architecture, the larger overheads in area and energy
of the push-only data-driven implementation is clear from the results. Also, the push-only implementation is only 4% faster than the optimised description-level implementation. It can be argued that using the improved architecture, a push-only implementation will achieve even higher performance, however, the added complexity of the new description will also increase the area and energy overheads. The performance-oriented techniques (based in a pull-push style) proposed here offer better performance trade-offs.

**Teak**

The nanoSpa processor is the largest and most complex design synthesised by Teak to date. The source description was practically the same as used to synthesise the nanoSpaRef design. The only source of incompatibility was the use of the select construct in some small modules, but the construct was relatively easy to replace, maintaining the original architecture. Table 7.3 shows the performance, area and energy results when the following optimisations were applied to the description:

- **VFJ**: removal of redundant Variables, and Fork and Join consolidation and displacement (used in this work as the basic set of optimisations).

- **VFJ+SMJ**: the above plus the optimisation of Steer-Merge-Join compositions.

### Table 7.2: Balsa nanoSpa performance, area and energy results.

<table>
<thead>
<tr>
<th>Optimisation applied</th>
<th>DMIPS†</th>
<th>Area</th>
<th>Energy‡</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>absolute</td>
<td>Δ (%)</td>
<td>elements</td>
</tr>
<tr>
<td>DD</td>
<td>73.54</td>
<td>—</td>
<td>662734</td>
</tr>
<tr>
<td>DD+nRFV</td>
<td>75.45</td>
<td>2.60</td>
<td>661651</td>
</tr>
<tr>
<td>DD+CF</td>
<td>75.99</td>
<td>3.34</td>
<td>664006</td>
</tr>
<tr>
<td>DDO</td>
<td>77.96</td>
<td>6.00</td>
<td>611793</td>
</tr>
<tr>
<td>DDO+nRFV</td>
<td>79.47</td>
<td>8.06</td>
<td>610361</td>
</tr>
<tr>
<td>DDO+CF</td>
<td>80.30</td>
<td>9.18</td>
<td>612337</td>
</tr>
<tr>
<td>DDO+nRFV+CF</td>
<td>81.74</td>
<td>11.14</td>
<td>609817</td>
</tr>
<tr>
<td>DDP (Taylor)</td>
<td>85.21</td>
<td>15.87</td>
<td>956753</td>
</tr>
</tbody>
</table>

† Dhrystone MIPS  
‡ per Dhrystone loop
- **VFJ+SMJ+DL**: the above plus the description-level optimisation techniques to remove channel variables as described in section 5.4.2.

All of the above designs used the three-latches per cycle insertion technique described in section 6.4 to allow the circuit to progress.

<table>
<thead>
<tr>
<th>Optimisation applied</th>
<th>DMIPS $^\dagger$</th>
<th>Area</th>
<th>Energy $^\ddagger$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>absolute</td>
<td>$\Delta$ (%)</td>
<td>elements</td>
</tr>
<tr>
<td>VFJ</td>
<td>24.78</td>
<td>—</td>
<td>2096953</td>
</tr>
<tr>
<td>VFJ+SMJ</td>
<td>27.87</td>
<td>12.46</td>
<td>1619302</td>
</tr>
<tr>
<td>VFJ+SMJ+DL</td>
<td>41.08</td>
<td>65.79</td>
<td>1674134</td>
</tr>
</tbody>
</table>

$^\dagger$ Dhrystone MIPS

$^\ddagger$ per Dhrystone loop

Table 7.3: Teak nanoSpa performance, area and energy results.

The results show that the optimisation of *Steer-Merge-Join* compositions has improved the speed by 12.46% and significantly reduced the area and energy overheads. Adding the description-level optimisations to remove channel variables increases the performance close to 66% with a small increase in area and decrease in energy. These optimisations target conditional constructs.

These results highlight the potential headroom for optimisation that still exist in Teak circuits as conditional structures can be identified as one of the main targets for future optimisations. The results of the circuit-level optimisation technique proposed in section 5.4.2 (shown in figure 5.23) demonstrate that the overhead of the conditional structures in speed, area and energy may be reduced even further.

Table 7.4 shows the comparative performance, area and energy results for the best Balsa and Teak implementations of nanoSpa. The Balsa nanoSpa is 87% faster than its Teak counterpart, which is also 150% larger and consumes 115% more energy.

One potential source of improvement still to be exploited is the circuit-level optimisation presented in section 5.4.2, which targets conditional structures commonly found in all three stages of the nanoSpa pipeline. This optimisation is not yet automated and its manual application was infeasible due to the complexity of the nanoSpa design. It is expected that by applying this optimisation together with a better design of the components and better latching insertion mechanism,
Teak will reach its goal of providing mechanisms to exploit high performance pipelined asynchronous circuit styles using the Balsa language.

The Balsa Synthesis System is a more mature system that has gone through a series of iterations, whereas Teak is in its initial stages of development. The contribution of this work to the development of Teak is twofold: (i) a proof of concept for the synthesis methodology through the use of a highly complex demonstrator, and (ii) a means to evaluate the performance of the resulting circuits and identify potential sources for their improvement.

<table>
<thead>
<tr>
<th>Decoder device</th>
<th>DMIPS †</th>
<th>Area elements ovh</th>
<th>Energy ‡</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balsa</td>
<td>73.54</td>
<td>662734</td>
<td>361</td>
</tr>
<tr>
<td>Teak</td>
<td>41.08</td>
<td>1674134</td>
<td>777</td>
</tr>
</tbody>
</table>

† Dhrystone MIPS
‡ per Dhrystone loop

Table 7.4: Comparison of the Balsa and Teak nanoSpa implementations.

7.2 An asynchronous Viterbi decoder

The design presented here is based in an initial description written by Gavant [42]. The description was optimised using the techniques presented in previous chapters and synthesised with the Balsa and Teak synthesis systems.

7.2.1 Introduction

Viterbi decoders [111] are used today in many digital communication applications to decode convolutional codes as part of a forward error correction (FEC) mechanism. The design of the decoder presented here is largely based in the architecture proposed by Brackenbury et. al [15] for an asynchronous Viterbi decoder aimed at a low power implementation.

Unlike the full-custom reference design, the approach in Gavant’s work was to create a synthesisable decoder using the Balsa language to facilitate the exploration of different approaches to reduce the power consumption. However, in line with the objectives of this thesis, the original description was optimised for performance and no considerations were made to the resulting power consumption.
7.2.2 Viterbi decoder algorithm

Convolutional encoding

A convolutional encoder takes the last $k$ bits of data arriving from a $v$-bit input stream and generates a $n$-bit output codeword ($n \geq v$) for each new $v$-bit input data word. The codeword is generated by combining the $k$ bits using modulo-2 (XOR) operations. The number $k$ is called the constraint length of the code. The ratio of the code is the fraction $v/n$. Figure 7.3 shows a convolutional encoder with $k = 3$, $v = 1$ and $n = 2$ (ratio = 1/2).

Figure 7.3: A convolutional encoder with $k = 3$ and code ratio = 1/2.

A convolutional encoder is a finite state machine with $2^{k-1}$ states. All possible transitions of the encoder can be represented using a trellis diagram. Figure 7.4 shows the trellis diagram for the encoder in figure 7.3. In this figure the circles represent states, a solid arrow indicates a transition when the input is 1 and a dashed arrow a transition when the input is 0. Each arrow is labelled with the output of the encoder. The highlighted path represents the state transitions for the input stream 1101, starting at state “00”.

Figure 7.4: Trellis diagram for the encoder in figure 7.3.
Decoding process

The Viterbi decoding algorithm is based on finding the most likely sequence of states (path) in the decoder that would have generated the received data. This is done by calculating, for each possible state the decoder can be in, a *branch metric* (BM) that reflects how close is the received data from the error-free data that the decoder would generate. These measures are combined with a *state metric* (SM) (based on previous observations) that represents the likelihood the encoder was in each state. The combinations of state and branch metrics are called *path metrics* (PM).

The higher of the PMs for each state represents the most likely starting point for the next decoding cycle. This PM is saved and becomes the new SM for that state in the next cycle. The identity of the path that has the highest PM (called the *local winner*) is also saved for use later in the reconstruction of the path the encoder took through the trellis. This process is called *backtracing*.

When the received voltage for each bit is quantised using more than 2 levels, the decoder is said to use *soft-decision* decoding.

### 7.2.3 Architecture of the asynchronous Viterbi decoder

The decoder consist of three units as shown in the block diagram of figure 7.5: the Branch Metric Unit (BMU), the Path Metric Unit (PMU) and the History Unit (HU). The parameters of the decoder are the following: code rate $= \frac{1}{2}$, constraint length $k = 3$ (four states), 3-bit soft-decision decoding and 16 slots of backtracing memory. A brief description of the units in the decoder follows. Extensive details about the principle of operation and the architecture can be found in [111, 15, 16].

![Figure 7.5: Architecture of the asynchronous Viterbi decoder.](image-url)
The Branch Metric Unit

The Branch Metric Unit (BMU) receives the error-containing data from the receiver and computes the distances between the ideal branch pattern symbols and the received data (branch weights). The distance to be calculated is the Manhattan distance, as this is equivalent to the Euclidean distance squared in this application [15]. The branch weights are then passed to the Path Metric Unit. Details of the BMU implementation can be found in section 4.3.1.

The Path Metric Unit

The Path Metric Unit (PMU) is the core of the Viterbi decoder. Here accumulative weight information relating to each possible encoder state (or node) is maintained.

The PMU, shown in figure 7.6, is composed of 3 main parts:

- the Add-Compare-Select (ACS) units, which compute the weight additions and determine the lowest weight between two previous states. This gives the direction (local winner) for the next branch, upper or lower.
- the PMU Memory, where the weights are stored.
- the Global Winner Generator, which determines the lowest weight of all the states already selected. The global winner is valid when the lowest weight is unique.

![Figure 7.6: The Path Metric Unit.](image-url)
The History Unit

The History Unit (HU) performs the backtracing. This is done only when a valid global winner is transmitted. With the local winner information, the previous state is computed and updated in the Global Winner Memory (GWM). This operation is repeated until the global winner computed is the same as in the GWM, which indicates that the backtracing has already been at that point. There are two memories, one the local winner (upper or lower) and one to store the global winner. The oldest state in the GWM is the current output of the decoder. Figure 7.7 shows a block diagram of the HU.

![Block diagram of the History Unit](image)

Figure 7.7: The History Unit.

Each unit in the architecture of the Viterbi decoder presents particular pipeline features. The BMU is a linear pipeline, the PMU is composed of a set of single-token rings and the HU is a single-token ring that performs multiple iterations over a token (a repeat-until loop). This unit is heavily control-dominated. As opposed to the nanoSpa pipeline, The Viterbi decoder pipeline has a fixed input-to-output token ratio.

7.2.4 Results

Balsa

The BMU was used as one of the running examples in chapter 4 of this thesis to demonstrate many of the description-level optimisations proposed here and some performance results were given. In this section performance results for the whole decoder will be given. The experimental set-up for the decoder consisted of decoding a stream of 1000 symbols with additive Gaussian white noise (AGWN)
and a signal-to-noise ratio $Eb/No = 2dB$. The parameter used to measure the performance is the average output data rate. Table 7.5 shows the performance, area and energy results for the following versions of the decoder:

- $VD$: the original unoptimised description.
- $VDO$: the description-level optimised version of $VD$.
- $VDO+CF+nRFN$: the $VDO$ description with the use of the concurrent Fetch component and the removal of redundant FalseVariables optimisations.

<table>
<thead>
<tr>
<th>Decoder device</th>
<th>data rate</th>
<th>$\Delta$ (%)</th>
<th>Area elements</th>
<th>Energy $nJ$</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>VD</td>
<td>31.59</td>
<td>—</td>
<td>58 815</td>
<td>145</td>
<td>1.00</td>
</tr>
<tr>
<td>VDO</td>
<td>64.75</td>
<td>200.5</td>
<td>80 640</td>
<td>218</td>
<td>1.50</td>
</tr>
<tr>
<td>VDO+CF+nRFN</td>
<td>66.98</td>
<td>212.0</td>
<td>68 595</td>
<td>159</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Table 7.5: Performance, area and energy results for the Viterbi decoder in Balsa.

The results indicate that fully description-level optimised version ($VDO$) achieves more than twice the performance of the original description. It is worth comparing this result with the 16% obtained by the more complex nanoSpa description in section 7.1.4. There are two reasons for this difference: firstly, the base design was written by a less experienced Balsa user and secondly the difference in complexity between the designs makes it easier to improve the critical path with the applied optimisations.

After applying the peephole optimisations to the optimised design ($VDO+CF+nRFN$), there is an extra increase in performance of 12%, (which translates into 3% when compared to the $VDO$ version). In this case, the results are similar to those obtained with nanoSpa. As in the case of nanoSpa, the peephole optimisations target only small parts of the whole design, possibly not all belonging to the critical path, hence the smaller increments in speed.

**Teak**

The Viterbi decoder was directly compiled in Teak from the optimised source code used for the Balsa synthesis. Table 7.6 shows the performance, area and energy results when the following optimisations were applied to the description:
- **VFJ+SMJ (1L)**: removal of redundant Variables, Fork and Join consolidation and displacement and Steer-Merge-Join optimisation. A simple, one-latch per link latching strategy was used here.

- **VFJ+SMJ**: The above optimisations but with the used of three-latches per cycle insertion technique.

- **VFJ+SMJ+JI**: the above design plus the input-join description-level optimisation described in section 5.4.2 applied inside the BMU and PMU.

- **VFJ+SMJ+JI+DL**: the above plus the description-level optimisation techniques to remove channel variables described in section 5.4.2.

<table>
<thead>
<tr>
<th>Decoder device</th>
<th>data rate</th>
<th>Area elements</th>
<th>Energy nJ</th>
<th>Area ratio</th>
<th>Energy ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFJ+SMJ (1L)</td>
<td>52.30</td>
<td>124656</td>
<td>269</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>VFJ+SMJ</td>
<td>60.76</td>
<td>106462</td>
<td>207</td>
<td>0.85</td>
<td>0.77</td>
</tr>
<tr>
<td>VFJ+SMJ+JI</td>
<td>62.22</td>
<td>104880</td>
<td>197</td>
<td>0.84</td>
<td>0.73</td>
</tr>
<tr>
<td>VFJ+SMJ+JI+DL</td>
<td>54.02</td>
<td>108744</td>
<td>215</td>
<td>0.87</td>
<td>0.80</td>
</tr>
</tbody>
</table>

Table 7.6: Performance, area and energy results for the Viterbi decoder in Teak.

The results in table 7.6 show that the more elaborate three-latches per cycle strategy improves the performance by 16% and reduces area and energy consumption by 15% and 23%, respectively, for this example.

The technique of joining inputs takes the performance improvement to 19% with further reductions in area and energy. Finally, notice that including the removal of channel Variables results in performance, area and energy penalties in this case. These results further support the observations of section 5.4.2. In the Viterbi decoder, the datapaths are narrow (3 to 6 bits) and the overhead of the tagging circuitry generated by the coding style overshadows the potential benefit of removing the Variables. In contrast, this technique was very effective in the nanoSpa design because of the widths of the datapaths within the design.

Table 7.7 compares the best Teak and Balsa implementations of the Viterbi decoder. The results shows a small performance overhead for the Teak implementation. However, the Teak implementation could be further optimised using the circuit-level optimisation of section 5.4.2, whereas the Balsa counterpart has already been fully optimised at both the description and circuit levels.
### 7.3 A 32×32-bit radix-8 Booth MAC

The Booth algorithm [13] is an efficient multiplication algorithm that is commonly used to implement the multiplication of two signed binary numbers in hardware. A number of bundled-data asynchronous multiplier have been described that implement the modified Booth algorithm, in which the number of iterations is fixed [54, 55, 94]. A bundled-data implementation of the original Booth algorithm, which skips consecutive chains of zeroes and ones leading to a number of iterations that depends on the operands, is described in [31].

The multiply-accumulate unit described here was designed to support all the variations of the ARM multiply instructions in the nanoSpa core, replacing the shift-and-add multiplier used in the SPA processor. Figure 7.8 shows the architecture of the nanoSpa multiplier.

The unit is a 32×32 multiplier with 32-bit accumulation. It is implemented as a radix-8 (Booth-3) modified Booth’s algorithm [13, 27]. This implementation was selected after comparing it with a radix-4 (Booth-2) implementation as a good performance-area trade-off: for an increase of 2.5% in the total nanoSpa processor area, the multiplier performance increases by 25% [95].

In figure 7.8, A and B are the multiplicands, and C is the optional 32-bit accumulate. The result is delivered as one or two 32-bits words (depending on the type of multiplication), H being the most-significant 32 bits and L the least significant 32 bits. The unit also calculates the zero (Z) and negative (N) flags. The multiplier consists of the following units:

**Bypass and Merge:** To support the speculative operation of the nanoSpa Execute stage, the multiplier is wrapped within the Bypass and Merge units. These units facilitate the early termination of the multiplication when the condition code of the instruction fails. If this is the case, the Bypass section generates a constant result of zero and discards the operands. In this way, the handshake is completed in the operand channels, the Booth loop is not

<table>
<thead>
<tr>
<th>Decoder device</th>
<th>data rate (Mspo)</th>
<th>Area elements (ovh %)</th>
<th>Energy nJ (ovh %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balsa</td>
<td>66.98</td>
<td>68 595</td>
<td>159</td>
</tr>
<tr>
<td>Teak</td>
<td>62.22</td>
<td>7.65</td>
<td>104 880</td>
</tr>
</tbody>
</table>

Table 7.7: Comparison of the Viterbi decoder in Balsa and Teak.
executed, and a result data is sent down the datapath to be discarded and to quickly finish the instruction. The Merge section passes either the zero results generated by Bypass or the actual multiplication and flags results to the output channel.

**Sign Adjust:** In order to accommodate the algorithm requirements, and the signed and unsigned operations specified in the ARM instruction set, this unit either sign-extend or zero-fill the operands $A$, $C$. For operand $B$, a zero is added at the least significant position (to complete the bit encoder bit grouping) and two bits are added at the most-significant positions (to save the carry out and to set the unsigned operands as positive numbers). This unit also passes the accumulate operand or zero if no accumulation is required.

**Booth-3:** This unit carries out the actual multiply-accumulate (MAC)
operation. This block consists of a Booth-3 decoder that selects the partial product to be added to the multiplicand, two arithmetic 70-bit shifters (one for the Carry-Save bits and one for the sum bits), a 32-bit Carry-Save Adder (CSA) to speed up the addition in the loop, a controller unit, and a 32-bit Carry-Propagate Adder (CPA). Together, the decoder, the shifters, the controller, and the CSA implement the Booth iteration. The shifter is initialised with the sign-extended multiplicand \(s_B\) in its lower 35 bits and with the value of the accumulate, \(s_C\) in its upper 35 bits (to add it on the first iteration). The sign-extended multiplier operand \(s_A\) is passed to the Booth encoder to generate the required partial products to be added. The controller initiates and stops the iterations and, after the last iteration, steers the CSA output and the 32 bits of the shifter containing the lower 32 bits of the result of the loop to the CPA. The CPA recodes the lower 32 bits by adding the lower halves of the two shift registers. It also recodes the upper 32 bits by adding the outputs of the CSA. In order to save hardware, these two recoding operations use the same CPA sequentially.

### 7.3.1 32-bit Multiply with 64-bit accumulation

Given that MAC operations with 64-bit accumulation are not very common, in order to perform a full 64-bit accumulation, nanoSpa executes any long multiply-and-accumulate in two cycles: the first cycle executes a MAC with 32 bit accumulation and then executes an ADD operation with the upper 32 bits of the MAC result and the upper 32 bits of the accumulate register. This architectural decision contributes to reducing the area overhead of the multiplier.

### 7.3.2 Results

The Booth-3 unit (the core of the multiplier) is a control-dominated circuit as can be seen in the “X-ray” picture of its Handshake Circuit, shown in figure 7.9. The parameter used to measure the performance was the average cycle time of signed multiply-and-accumulate operations. The design was synthesised in Balsa and Teak from an already optimised source, however some of the new optimisations were also applied. This design was used to further compare Balsa and Teak synthesis styles using a medium-complexity control-dominated example.
Figure 7.9: An “X-ray” picture of the Booth-3 Handshake Circuit revealing its control tree.

Balsa

Table 7.8 shows the performance (average delay time of 32-bit multiply and accumulate operations), area and energy results for the following versions of the multiplier:

- **MAC**: the original description.
- **MAC+DL**: the MAC, with description-level optimised case guards and explicit duplication in the Booth encoder block.
- **MAC+DL+AEC**: the above plus the optimisation of the control of active enclosures described in section 4.6.2.

The results show that, despite the small room available in this very optimised control-dominated design, some performance increase can be achieved with the new optimisations: The description-level optimisations delivered 2.35% performance increase, which together with the optimised control results in an extra 3.57% with a 5% of area overhead.
Section 7.3  A 32×32-bit radix-8 Booth MAC

Table 7.8: Performance, area and energy results for the MAC unit in Balsa.

<table>
<thead>
<tr>
<th>Multiplier device</th>
<th>delay ns</th>
<th>Δ (%)</th>
<th>Area elements</th>
<th>Energy nJ</th>
<th>Energy ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC</td>
<td>90.60</td>
<td>—</td>
<td>115680</td>
<td>80</td>
<td>1.00</td>
</tr>
<tr>
<td>MAC+DL</td>
<td>88.60</td>
<td>2.36</td>
<td>120900</td>
<td>78</td>
<td>0.98</td>
</tr>
<tr>
<td>MAC+DL+AEC</td>
<td>87.48</td>
<td>3.57</td>
<td>120961</td>
<td>80</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 7.8 shows the performance, area and energy results when the following optimisations were applied to the MAC description:

- **VFJ+SMJ (1L)**: removal of redundant Variables, Fork and Join consolidation and displacement, and Steer-Merge-Join optimisation. A simple, one latch per link latching strategy was used here.
- **VFJ+SMJ**: The above optimisations but with the use of three latches per cycle insertion technique.
- **VFJ+SMJ+JI**: the above design plus the input-join description-level optimisation described in section 5.4.2.

Table 7.9: Performance, area and energy results for the MAC unit in Teak.

<table>
<thead>
<tr>
<th>Multiplier device</th>
<th>delay ns</th>
<th>Δ (%)</th>
<th>Area elements</th>
<th>Energy nJ</th>
<th>Energy ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFJ+SMJ (1L)</td>
<td>147.88</td>
<td>—</td>
<td>249070</td>
<td>153</td>
<td>1.00</td>
</tr>
<tr>
<td>VFJ+SMJ</td>
<td>133.66</td>
<td>10.64</td>
<td>179242</td>
<td>98</td>
<td>0.64</td>
</tr>
<tr>
<td>VFJ+SMJ+JI</td>
<td>129.34</td>
<td>14.33</td>
<td>180972</td>
<td>93</td>
<td>0.61</td>
</tr>
</tbody>
</table>

The results in table 7.9 show that the three-latches per cycle strategy has improved the performance by 10% and has reduced the area by 30% and the energy consumption by 36%. These results are similar to those obtained for the Viterbi decoder.

The technique of joining inputs improves the MAC performance by 14% with a small increase in area, but smaller energy consumption. In this design it was impractical to use the description-level technique to remove the channel Variables due to its heavily sequenced and iterative architecture.
Finally, table 7.10 compares the best Teak and Balsa implementations of the MAC unit. In this control-dominated case, the Teak overhead in performance is larger, mainly because the design heavily relies on the use of variables.

<table>
<thead>
<tr>
<th>Multiplier device</th>
<th>delay ns</th>
<th>ovh %</th>
<th>Area elements</th>
<th>ovh %</th>
<th>Energy nJ</th>
<th>ovh %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balsa</td>
<td>87.48</td>
<td>—</td>
<td>120961</td>
<td>—</td>
<td>80</td>
<td>—</td>
</tr>
<tr>
<td>Teak</td>
<td>129.34</td>
<td>47.85</td>
<td>180972</td>
<td>49.61</td>
<td>93</td>
<td>16.15</td>
</tr>
</tbody>
</table>

Table 7.10: Comparison of the MAC implementations using Balsa and Teak.

### 7.4 The nanoSpa Forwarding Unit

This section presents the description of a synthesisable result forwarding unit for the nanoSpa asynchronous microprocessor, using the syntax-directed synthesis approach and targeting a robust QDI implementation. The author has published a paper based on this work [96].

#### 7.4.1 Introduction

Result forwarding [49] is a method used in pipelined microprocessors to reduce the penalty caused by inter-instruction data dependencies. The forwarding mechanism can also be used to allow partial overtaking of (normally slow) memory operations by faster instructions, whilst making sure that the instructions complete in the same order as they appear in the instruction stream. Figure 7.10 depicts some potential performance benefits of the result forwarding mechanism.

In synchronous systems, the problem of result forwarding can be easily solved because the clock signal serves as a reference that allows synchronisation between result producing and consuming units. In an asynchronous environment, the problem of implementing a result forwarding mechanism is more complicated due to the lack of synchronisation between producers and consumers. In this case, one cannot rely on a control signal that indicates which cycle an instruction is in as this requires a lockstep operation of the pipeline that would heavily penalise the performance.

An efficient, full-custom solution to the problem of result forwarding within an asynchronous environment was proposed and implemented in the Amulet3
Section 7.4 The nanoSpa Forwarding Unit

Without forwarding

```
ADD R1, R1, R2
CMP R1, R0
```

With forwarding

```
ADD R1, R1, R2
CMP R1, R0
```

Figure 7.10: Potential performance benefits of result forwarding in a 4-stage pipeline.

asynchronous processor [43, 44, 36], targeting a bundled-data implementation, with the consequent limitations on design-space exploration, technology portability due to its full custom design, and with similar timing closure problem as synchronous designs. In order to overcome such limitations and reduce the impact of increasingly difficult timing closure within modern fabrication process variability, it is desirable to have a synthesisable asynchronous description which can be mapped into a quasi-delay-insensitive implementation.

The following sections introduce relevant related work and discusses the implementation of a forwarding mechanism designed to be used in the nanoSpa processor described earlier in section 7.1.

### 7.4.2 Related work

Earlier asynchronous techniques for resolving dependencies include: the register locking mechanism for the Amulet1 processor [80, 35], register locking plus “last result” register used in the Amulet2 processor [37, 112], the last result bypass mechanism of the Caltech asynchronous MIPS [69], the scoreboard-like Data Hazard Detection Table (DHDT) of the SAMIPS processor [118], the CounterFlow Pipeline Processor architecture (CFPP) proposed in [92] and the asynchronous “queue” FIFO [43] for the Amulet3 processor [38]. The ARM996HS processor by Handshake Solutions is a commercially-available synthesisable asynchronous 32-bit CPU that was implemented using the TiDE tools [23]. The processor ARM996HS core is a five-stage asynchronous pipeline and so may benefit from result forwarding but no information has been published about the dependency
avoidance technique used. As with Amulet3, its implementation uses bundled-data encoding.

The Amulet3 asynchronous “queue” FIFO (AQF from herein) was used as the reference model for the nanoSpa forwarding unit (nFU). The AQF is a circular buffer that acts both as a forwarding unit and a reorder buffer. The AQF stores the results and their register destinations from previous instructions. Figure 7.11 shows a diagram of the AQF process model. The queue operation consists of 5 processes: Lookup, Allocation, Forward, Arrival and Writeout [44, 43].

![Diagram of AQF process model]

Figure 7.11: AQF process model.

**Lookup:** This process receives the source register names for instruction operands from the decoder, examines the queue to see if they are present, and returns a bit mask indicating the possible data source positions in the queue. This is performed using a CAM (Content Addressable Memory) that holds the previously allocated destination registers.

**Allocation:** After obtaining the lookup source mask, the instruction’s own destination address can be written into the CAM. The writing position is allocated cyclically within the circular buffer structure.

**Forward:** Concurrently with Allocation, this process receives the mask generated during Lookup, examines each of the possible sources (starting at the most recent), waits until the data is present and then checks for
validity. Valid data is forwarded to the required places, otherwise the process examines the next most recent possibility. If all the possibilities are exhausted (or if there were no data sources) the forwarding process gives up and the default value, read from the register bank, is used.

**Arrival:** Results arriving at the queue carry their allocated queue address. The allocation process guarantees non-conflicting allocations even in the event of multiple writes. When the data allocated to a particular slot arrives, the previous data in the slot will have been written back to the register and so can be overwritten without conflict. If the instruction was abandoned due to conditional execution then the result will be marked as invalid.

**Writeout:** This process copies valid results back to the register bank. It examines the queue locations cyclically and waits until the valid result arrives then copies the data to the register bank and marks the location as “empty” so it can be reallocated.

In order to improve the speed of the Lookup process, the Amulet3 AQF uses a small CAM to hold the information about the registers written in the buffer. Speculative read of the default value from the register bank is also performed in case the source operand is not present in the buffer. The AQF has a centralised, token-passing asynchronous control and features three read ports for forwarding and two write ports for arrival.

### 7.4.3 The target processor: nanoSpa

In a new experimental description of nanoSpa, the pipeline depth has been increased to enhance the performance. Figure 7.12 shows a simplified version of the new 5-stage nanoSpa pipeline.

### 7.4.4 Architecture of the nanoForward Unit

The nFU has the same number of read ports (3) and write ports (2) as the AQF, but as the current nanoSpa architecture does not execute instructions out of order, the nFU is not used as a reorder buffer.
Figure 7.12: The 5-stage nanoSpa pipeline.

Figure 7.13: The nanoForward Unit architecture

Figure 7.13 shows the architecture of the nFU and its location within the new nanoSpa pipeline. The figure shows details of the communication interface between the various processes, the queue and the processor units.

The decode stage generates sequenced values of allocation pointers (allocPtr)
that steer the allocation and arrival data and guarantee mutual exclusivity in
the allocation of queue cells. There are two allocation pointers, because some
instructions can generate up to two results. The queue cells communicate using a
token-passing mechanism to avoid cell reallocation when successive two-result in-
structions appear in the pipeline. Also, each queue cell handles its communication
with the other processes independently.

7.4.5 Implementation issues

In ARM processors any instruction can be executed conditionally, which adds
extra complexity to the result forwarding mechanism. In order to improve the
efficiency of the pipeline in both the AQF and the nFU, allocation is done regard-
less of whether the instruction is conditional. If a conditional instruction fails its
condition code tests, a token is sent through the pipeline to indicate that the
instruction has been processed and the allocated queue slots are marked as in-
valid. This introduces some wasted slots in the queue, however, figures reported
in [43] give 90% of queue utilisation for typical ARM programs by using this
unconditional allocation strategy.

Synchronisation between processes

To guarantee correct operation, on each instruction the nFU must perform several
operations sequentially as shown in figure 7.14. An initial nFU description was
based on the use of sync channels as a token-passing mechanism to synchronise
the processes but this caused a large performance penalty due to its reliance on
the use of Sequencers so alternatives were looked for.

Figure 7.14: Inter-process dependencies in the nFU.
A solution that dramatically reduced this penalty was to perform synchronisation using data instead of sync tokens: to decouple Forward from Arrival, the queue contents are read speculatively and sent through data channels to the Forward process. Lookup and Allocation were decoupled using an “allocation mask” that blocks the reading of the queue locations that are about to be modified by the allocation/arrival process during the current instruction. This masking mechanism has two drawbacks: (i) the effective length of the queue is reduced in one or two locations, depending on the number of results to be written (one or two), and (ii) it dissipates more power and requires larger area.

Another alternative is to implement a less concurrent operation by grouping the processes according to the information that they read or write: Lookup; Allocate are sequenced as they read; write the register names and the valid flag. Similarly, Forward; Arrival are sequenced because they read; write results. In this way, Lookup; Allocation can now run concurrently with Forward; Arrival. Synchronisation between Lookup and Forward is done with data tokens carrying the lookup result. Allocation and Arrival completion must be synchronised and this information triggers the Writeout process.

**Optimising sequenced operations**

One performance problem that arises with the grouping scheme presented earlier is that, as explained in section 4.6.3, read-then-write operations require the use of non-RTZ-overlapped sequencers based on the S-element in order to avoid the risk of WAR hazards. To allow a more concurrent operation with decoupled RTZ phases, the processes can be rearranged as Allocation; Lookup and Arrival; Forward. This write-then-read operation permits the safe use of a sequencer based on the T-element but requires an initial empty token to be sent to Allocate and Arrival before the nFU begins to process instructions.

In Balsa, a write-then-read sequence to a variable inside a procedure generates a sequencer based on the T-element. However, because in the nFU the write and read processes reside in separate modules (with multiplexed/demultiplexed accesses to a global variable) the Balsa compiler inserts a safe non-RTZ-overlapped sequencer. An improvement to the above solution is to take advantage of the unbounded repetition of read-then-write actions over common variables and use the read-then-write optimisation describe in section 4.6.3.
Lookup CAM and forward process implementation

In the Amulet3 AQF, the Lookup process uses a small, very fast custom CAM to determine if the source registers of the decoded instruction are written or have been allocated in the buffer. Balsa does not provide a way to describe a CAM and generate an efficient circuit structure. The Balsa synthesised circuit used to replace the CAM consists of a number of logic comparators that, despite being relatively simple, do not perform as well as an optimised CAM, resulting in some performance penalty for the Lookup process.

In the Amulet3 AQF the Forward process iteratively examines the possible data sources until valid data is found or, if all possibilities are exhausted, the default value read from the register bank is used. This operation was efficiently implemented at the signal-level. As Balsa is a behavioural language, no signal-level operations can be described and attempting to replicate this behaviour in the nFU would require extensive use of sequenced operations that penalise performance.

The implemented solution is to wait for the data validity flag during the allocation process and to attach this information to the register number before writing it to the nFU CAM substitute. In this way the CAM substitute will report nothing or the single most recent valid source to the forwarding process, avoiding the need for iteration.

7.4.6 Use of the permissive Concur

The composition of concurrent actions in the nFU allows the use of the permissive concur to enhance the performance. The operations in the nFU were grouped into two concurrent groups of actions: (Lookup; Allocate) and (Forward; Arrival). These actions read and write from the same set of variables (the Queue buffer). The allocation pointer and token passing mechanism guarantees mutual exclusivity of these read and write actions allowing the use of the permissive concur. Outputs of these processes can be merged without the use of the select construct acting as a data driven merge.

For the case of the queue, the guaranteed mutual exclusivity allows the Allocation and Arrival processes of each cell to be composed with permissive concurs leading to similar benefits. Figure 7.15 shows the composition of the different
processes using the permissive concur inside the description of the nFU. For clarity, the I/O signals have been removed in the code. The complete source code can be found in Appendix G.

```
-- Lookup/Allocate group
loop
   -- Lookup (one for each read port)
   for || i in 0..READPORTS-1 then
      lookup(i, ...)
   end;
   -- steer Allocate information to allocation subcells
   steerAlloc_1(...) ||!
   steerAlloc_2(...) )
end ||!

-- Forward/Arrival group

loop
   -- forward (one for each read port)
   for || i in 0..READPORTS-1 then
      forward (i, ...)
   end;
   -- Steer arrival requests
   steerArrival_1(...) ||!
   steerArrival(...) )
end ||!

-- Cell allocation subprocesses
for || i in 0..ROBSIZE-1 then
   allocCell( i, ...)
end ||!

-- Cell arrival & writeout subprocesses
for ||! i in 0..ROBSIZE-1 then
   arrCell( i, ...)
end
```

Figure 7.15: Composition of actions with the permissive Concur inside the nFU.

7.4.7 Results

After a series of pre-layout, transistor level simulations it was found that the optimum queue size is 4. Different architectures of the nFU were tested and compared running the Dhrystone benchmark program. Tables 7.11 and 7.12 show that performance increases were 10%, with area and energy overheads of 13%. These results also show that the techniques used for desynchronising the
processes achieve close to 40% increase in performance relative to the use of sync channels. Results show that the first-read-unfold technique described in section 4.6.3 is a key factor for the performance gain in the nFU, contributing more than 50% of the speed-up.

<table>
<thead>
<tr>
<th>nanoSpa device</th>
<th>DMIPS</th>
<th>speed-up (%)</th>
<th>area overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no nFU</td>
<td>78.37</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>nFU (sync signals)</td>
<td>61.22</td>
<td>-28.80</td>
<td>5.20</td>
</tr>
<tr>
<td>nFU (allocation mask)</td>
<td>82.03</td>
<td>4.67</td>
<td>15.71</td>
</tr>
<tr>
<td>nFU (grouping)</td>
<td>81.86</td>
<td>5.86</td>
<td>11.20</td>
</tr>
<tr>
<td>nFU (grouping + unfolding)</td>
<td>86.27</td>
<td>10.08</td>
<td>11.21</td>
</tr>
</tbody>
</table>

Table 7.11: Performance results for nanoSpa using the nFU

<table>
<thead>
<tr>
<th>nanoSpa device</th>
<th>Energy for a Dhrystone loop($\mu$J)</th>
<th>overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no nFU</td>
<td>0.360</td>
<td>0.00</td>
</tr>
<tr>
<td>nFU (allocation mask)</td>
<td>0.491</td>
<td>36.23</td>
</tr>
<tr>
<td>nFU (grouping)</td>
<td>0.393</td>
<td>8.90</td>
</tr>
<tr>
<td>nFU (grouping + unfolding)</td>
<td>0.408</td>
<td>13.33</td>
</tr>
</tbody>
</table>

Table 7.12: Energy results for nanoSpa using the nFU

Unfortunately, it is not possible to make a relative comparison of the performance gain with respect to the Amulet3 AQF, because there are no published figures with and without the AQF. Pre-implementation, simulation results in [44] suggest that the AQF in Amulet3 would increase its performance by 22.5% when running the Dhrystone benchmark. Notice also that the Amulet3 pipeline has a decoupled memory stage and this feature is not currently present in nanoSpa.

### 7.5 A sliced-channel wormhole router

This section presents the architecture of a novel sliced-channel wormhole router proposed by Wei Song [90] and the results of some optimisations applied to its Balsa description. Details of the implementation and operation can be found in the reference given.
7.5.1 Introduction

Network-on-chip (NoC) is a new on-chip communication paradigm. Asynchronous NoCs are attractive because they are power efficient and robust to process variation. As opposed to the store-and-forward routing scheme used in macronetworks, in NoC the prevailing scheme is wormhole routing [12]. In store-and-forward routing the node stores the complete packet and forwards it based on the information within its header. In wormhole routing, the packet is decomposed into smaller units called flits (flow control digits). The network node looks at the header of the packet to determine its next hop and immediately forwards it. The subsequent flits are forwarded as they arrive, causing the packet to worm its way through the network possibly spanning a number of nodes. The advantages of wormhole routing are low latency and the avoidance of area costly buffering queues [12].

In wormhole routing each packet is decomposed into three types of flits: (i) the head flit, which conveys the routing information (destination address) for the subsequent flits; (ii) a variable number of data flits, which carry the payload and (iii) the tail flit, which is used to close the connection.

7.5.2 Architecture of the sliced-channel wormhole router

In order to meet bandwidth requirement, state-of-the-art asynchronous routers broaden their channels by synchronising multiple sub-channels [12, 2, 86]. The new router architecture proposed in [90] and described here uses multiple independent sub-channels to transmit data. Since some synchronization is removed, the cycle period of all sub-channels are reduced, speeding up the network.

Figure 7.16(a) shows the simplified datapath of a wormhole NoC using synchronized channels. If, for instance, the asynchronous channel between routers is formed by four sub-channels, a four input C-element tree is required to generate the ack signal on each port. All sub-channels are merged into one channel and traverse the router through the multiplexer controlled by an arbiter. To remove the C-element tree, the data path could be restructured as shown in figure 7.16(b). The four sub-channels still go through the multiplexer together but each of them has its own ack line and can run independently.
The sliced-channel wormhole router

The implemented sliced-channel wormhole router has five 32-bit ports. To avoid separating the address in the head flit, the data width of a sub-channel in the router is set to 8 bits, allowing the header to address a $16 \times 16$ mesh. Consequently, the 32-bit channel is divided into four 8 bit sub-channels. The wire count is increased to 76 because the sub-channels now have their own set of end-of-frame and $ack$ wires. In contrast, a conventional router having the same number of channels requires 67 wires: 64 data wires, two wires for the end-of-frame bit and one $ack$ wire.

The architecture of the new router is shown in figure 7.17. The router comprises five input buffers, five output buffers and five multiplexers controlled by five arbiters. The depth of all buffers is one bit.

In this design example the dominating structures in the Balsa description are the data-dependant conditional structures that implements the input buffers and crossbar (multiplexers and demultiplexers). The control consists of the arbiters that select the routes and iterative \textbf{loop} \, \textbf{while} \, structures (localised at each input and output buffer) that detect the end (tail) of the packets entering/leaving the router.
7.5.3 Results

The experimental set-up consisted of sending packets of random lengths from every port to randomly selected port/destination and measuring the average throughput of the whole router (sum of the throughput of all ports). Random lengths and destinations were pre-generated and the same set was used for all designs. Table 7.13 shows the performance, area and energy results for the following versions of the router:

- \( WR \): the original unoptimised description.
- \( WR+DL \): the description-level optimised version of \( WR \). In particular, the guard optimisation and guard grouping were used in the input buffers and crossbar.
- \( WR+DL+AEC \): the above plus the optimisation of the control of active enclosures described in section 4.6.2.
- \( WR+B+DL \): the \( WR+DL \) plus data broadcasting in the output buffers.
- \( WR+B+DL+AEC \): the above plus the optimisation of the control of active enclosures

Results show that the description-level optimisation of guards has increased the performance by 7.3% with a reduction in area of 14% of the original and a
penalty of 7% in energy. Adding the peephole optimisation of the active eager inputs increases the performance by 10% with negligible penalties. Applying a more aggressive optimisation in the output buffers increases the performance by 17.4% at the expense of larger area and energy penalties.

### 7.6 Summary

This chapter has presented the evaluation of the performance-oriented techniques introduced in this thesis on a set of medium-to-large complexity designs described in the Balsa language. The impact on performance for the different techniques varies depending on the operational complexity of the circuit, with control-dominated circuits having smaller performance increases.

#### 7.6.1 Balsa

The combined use of description-level optimisations obtained performance gains that range in percentage from 5-10% for the control-dominated MAC to 200% for the Viterbi decoder. The Viterbi example is interesting because the source description was written by an inexperienced Balsa user, highlighting the fact that the expressiveness of the language can lead to functional but poor implementations. In contrast, the source description of the nanoSpa processor (by far the most complex example investigated) was written by a highly experienced user, leaving less room for improvement.

New peephole optimisations were applied to highly optimised code where they can be more effective. However, as they target more localised sections of a circuit, the performance increase obtained is limited.
In general, the description-level optimisations result in small area and energy penalties. However, some of the large speed-ups are associated with larger overheads. The combined use of the two types of optimisations compares very favourably to a more aggressive push-only data-driven style, achieving similar levels of performance increases at relatively very low cost in overheads. This result suggests that a combination of push-only data-driven style and the optimisations introduce here might yield larger improvements at lower overhead costs.

### 7.6.2 Teak

Three designs were used as evaluation for Teak: the nanoSpa, the Viterbi decoder and the MAC unit. The circuit-level optimisations proposed for Teak corresponds to the initial set of optimisations derived for Teak circuits and, in contrast to the Balsa examples, a reference design was not available. In spite of this, some sets of the optimisations were applied separately to highlight the potential optimisation headroom available. In particular, the optimisation of conditional structures that results in compositions of Steer-Merge-Join components were evaluated, showing these structures as an excellent target for optimisation. The other optimisation highlighted in the examples was the latch insertion mechanism proposed in section 6.4. The results demonstrate its effectiveness in speeding up the circuit, saving area and energy as a side effect.

The proposed description level optimisations targeting the elimination of channel variables effectively improved the performance of the designs, resulting in speed-ups directly proportional to the width of the datapaths involved.

The evaluation examples demonstrated that the Teak methodology is capable of synthesising large complex circuits that operate correctly, but currently the performance overhead of Teak circuits for complex designs (like nanoSpa) or for control-dominated circuits (like the multiplier) is too large. The structures used to provide conditional access to channels and the sequencing of operations were identified as one the main sources of overhead. For the conditional access to channels, the circuit-level optimisation described in section 5.4.2 appear to be a promising source of improvement but is yet to be automated.
Chapter 8

Conclusions and future work

8.1 Balsa

The syntax-directed synthesis approach targeting handshake circuits used in Balsa is a flexible method that allows the synthesis of complex asynchronous VLSI circuits. The flexibility for making design trade-offs at the description-level has been claimed to be one of its major advantages. The major drawback of the method is the poor performance of the synthesised circuits and different techniques have been proposed to optimise them.

This thesis has proposed and evaluated a series of description-level and peephole optimisations to increase the performance of circuits synthesised using the syntax-directed approach. The synthesis and the optimisations presented here target dual-rail, quasi-delay-insensitive implementation as this is a robust approach that helps to reduce the impact of increasingly difficult timing closure within modern fabrication process variability.

This work has contributed to the knowledge of the asynchronous design methodologies by proposing and analysing a set of description-level techniques that result in faster compositions of the target structures used in the handshake circuits approach. The techniques are based on the data-driven style of description in which the arrival of data activates the operations of the circuits, as opposed to the more traditional and straightforward control-driven style. The overall effect of the proposed description techniques is the splitting of the tree of control elements of the synthesised circuits into smaller clusters, resulting in a reduction of the associated overhead.

Another contribution is a new set of peephole optimisations targeting Balsa
handshake circuits that further increase the performance of the synthesised circuits. In general, the description-level optimisations result in small area and energy penalties and in some cases, they even improve area and energy consumption. However, large speed-ups are associated with increased area and energy overheads. The peephole optimisations presented here have negligible overheads.

The performance gains obtained by using the different optimisations depends on the original input source code. For sources written by experienced designers, performance increases of 15-24% were achieved with area and energy penalties of less than 17% in most cases. The peephole optimisations achieved limited performance increases (of the order of 5-10% for the examples analysed) because they target smaller sections of the system.

The combined use of the two types of optimisations compares very favourably to a more aggressive push-only data-driven style, achieving similar levels of performance increases at relatively very low cost in overheads. This was demonstrated using a large and complex design example. The result suggests that it may be possible to obtain higher performances at a lower cost with an adequate mixture of both techniques. This will be discussed in the future work section.

A final contribution of this work is a varied set of highly optimised designs (and their corresponding simulation results) that can be used in further investigations.

8.2 Teak

This work has also evaluated a novel token flow-based asynchronous synthesis approach and techniques for increasing the performance of the resulting circuits were proposed and analysed. Although sharing the same input language as Balsa, the synthesis method is different to both Balsa and Haste, hence different optimisation methods had to be devised. Three designs were used as evaluation for this novel token-flow approach: the nanoSpa, the Viterbi decoder and the MAC unit.

The proposed optimisations for Teak fall into circuit-level and description-level categories. Circuit-level optimisation rely on the properties of the Teak components and its compositions and comprise circuit transformations, pattern-matching and substitution, or a combination of transformation and substitution. Description-level techniques target the removal of channel Variables with conditional accesses.
Two main targets for optimisation were identified: Steer-Merge-Join compositions and conditional read accesses to channels. The optimisation of Steer-Merge-Join compositions is achieved using circuit transformations and substitutions whilst conditional read accesses to channels were optimised using description-level techniques. Furthermore, a circuit-level optimisation was proposed for conditional read accesses that could further improve the performance of Teak circuits.

Teak circuits need latch insertion to prevent deadlock within circuits with cycles. An automatic latch insertion mechanism based on the minimum token storage required in a cycle was proposed, analysed and incorporated in the synthesis system.

The evaluation examples demonstrated that the Teak methodology is capable of synthesising large complex circuits. However, further optimisations are necessary to obtain competitive levels of performance with Balsa circuits.

8.3 Future work

The optimisations presented here can contribute in several ways to further improve existing tools used in the synthesis of asynchronous circuits and to create new ones. The work conducted on the Teak synthesis is just one of the first steps towards the implementation of a mature synthesis tool for this novel synthesis approach.

8.3.1 Description-level optimisations

The circuit structures that result from the optimised descriptions can serve as a reference to create the mappings in an optimisation step of the compiler or can be incorporated as rules for automated source-to-source transformation tools. As an example, the optimisations of the guards evaluation and the encoding of multiple guards look like excellent candidates for automation.

8.3.2 Peephole optimisations

The peephole optimisations proposed for the Balsa handshake circuits can be incorporated into the Balsa compiler and further evaluations performed on them. With some of the more complicated situations which are difficult to match with
a template, such as the read-then-write sequencing, the optimisation could be incorporated as a “pragma” in the source code.

8.3.3 Synthesis using hybrid style

Balsa normally generates modules with active input ports (pull inputs) and active output ports (push outputs) with a mixture of active and passive inputs at the handshake component level. In contrast the data-driven synthesis proposed in [101], which uses push-only handshake components (passive inputs and active outputs), provide faster performance but poorer area and energy consumption. The fact that the description-level optimisations have closed the gap between Balsa and the push-only style suggests that there may be inefficiencies with the push-only approach that could be exploited by using pull structures in key places of the handshake circuits. Clearly there will be more than one way of mixing these styles, either importing push-style modules to replace slower Balsa mixed-style modules or incorporating the more efficient push-style components in Balsa or vice versa. Investigating these inefficiencies and the best way of implementing this hybrid style is a challenging future research topic.

8.3.4 Teak

Teak is still a project under development. The evaluation carried out during this work was part of the initial proof of concept for the methodology, and this work has opened a series of paths to continue its development.

The automation of the proposed circuit-level approach to remove Variables associated with conditional channel reads is required to provide further enhancements in the performance, area and energy of the synthesised circuits.

The optimisation of the three latches per cycle is necessary to reduce the number of redundant latches due to the overlapping of cycles. This is an NP, non-trivial problem that opens a good research opportunity. The use of heuristics based on the structure of the Teak networks could serve as the basis for an optimised latch insertion mechanism.

At the component-level there is much to do on the design of optimised versions of Teak components, and some work on this has already started. The methodology allows the components to be designed with any chosen degree of channel
coupling, and there are good research opportunities in investigating better degrees of channel decoupling that can be embedded inside each component. It is even possible to have different versions for each component and select the one that provides the best performance depending on the construct, the neighbour components or datapath width.

Circuit transformations and peephole optimisations for Teak circuits can be described in a language external to the compiler to facilitate its description, composition and application to the circuits. There is already some work in progress within the APT group to develop this idea.
References


REFERENCES


[112] Amulet webpages. \url{http://intranet.cs.man.ac.uk/apt/projects/processors/amulet/}.


## Appendix A

### List of Balsa operators

The following table shows the operators available in Balsa, in order of decreasing precedence:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Valid types</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>record indexing</td>
<td>record</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>smash</td>
<td>any</td>
<td>takes value from any type and reduces it to an array of bits</td>
</tr>
<tr>
<td>[]</td>
<td>array indexing</td>
<td>array</td>
<td>non-constant index possible, can generate lots of hardware</td>
</tr>
<tr>
<td>-</td>
<td>exponentiation</td>
<td>numeric</td>
<td>only constants</td>
</tr>
<tr>
<td>not, log, - (unary)</td>
<td>unary operators</td>
<td>numeric</td>
<td>log only works on constants, returns the ceiling: e.g. log 15 returns 4. - returns a result 1 bit wider than the argument</td>
</tr>
<tr>
<td>*, /, %</td>
<td>multiply, divide, remainder</td>
<td>numeric</td>
<td>only applicable to constants</td>
</tr>
<tr>
<td>+, -</td>
<td>add, subtract</td>
<td>numeric</td>
<td>results are 1 or 2 bits longer than the largest argument</td>
</tr>
<tr>
<td>@</td>
<td>concatenation</td>
<td>arrays</td>
<td></td>
</tr>
<tr>
<td>&lt;,&gt;,&lt;,&gt;,=</td>
<td>inequalities</td>
<td>numeric, enumerations</td>
<td></td>
</tr>
<tr>
<td>=, /=</td>
<td>equals, not equals</td>
<td>all</td>
<td>comparison is by sign extended values for signed numeric types</td>
</tr>
<tr>
<td>and</td>
<td>bitwise and</td>
<td>numeric</td>
<td>Balsa uses type 1 bits for if/while guards so bitwise and logical operators are the same</td>
</tr>
<tr>
<td>or, xor</td>
<td>bitwise or, xor</td>
<td>numeric</td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Balsa binary/unary operators [30].
Appendix B

Balsa handshake components

This appendix provides a brief description of the handshake components available in the Balsa Synthesis System that appear in the example circuits of this thesis. For details on the implementation and formal description, the reader can refer to [5] and the Balsa Manual [30].

Balsa handshake components can be divided into three categories, according to its interaction with data and control signals.

Control components use only sync (dataless) ports. Their operation is triggered through the activate port. Their output sync channels are connected to the activation ports of other components.

Datapath components have only data channels. They are used for storing, processing, merging and splitting data channels.

Control to datapath interface components control the movement of data through the datapath. They have one or more sync ports used to communicate with control components as well as data channel ports. Some of them initiate handshakes on data channels in response to activation. Others generate an activation in response to the arrival of data.

B.1 Control components

B.1.1 Loop

Loop implements unbounded repetition. After receiving an activation on its passive port, it produces an infinite number of activations on its active port.
B.1.2 Concur

Produces an activation on all of its output ports following an input activation. All the output activations are begun at the same time but then operate independently.

B.1.3 Fork

Produces an activation on all of its output ports following an activation on its input. All outputs synchronise between the processing and RTZ phases.

B.1.4 WireFork

Produces an activation on all of its output ports following an activation on its input, but never returns an acknowledgement. WireFork effectively forks the activation request to all of its outputs.
B.1.5 Sequence

Upon receiving and activation, its output activations are produced one at a time in sequence.

\[ \text{activateOut} \]

B.1.6 Call

Call passes a handshake on one of its input ports to the output port. The inputs must not occur concurrently.

\[ \text{inputs} \quad \Rightarrow \quad \text{out} \]

B.1.7 Sync

Synchronises the request on all of its inputs before passing these handshakes to the output.

\[ \text{inputs} \quad \Rightarrow \quad \text{out} \]

B.1.8 Arbitrate

Passes handshake on inA to outA or a handshake on inB to outB. If both inA and inB are activated concurrently it makes a non-deterministic decision as to which to pass first.

\[ \text{inA} \quad \text{arb} \quad \text{outA} \]
\[ \text{inB} \quad \text{outB} \]
B.1.9 DecisionWait

Synchronises an activation with one of its inputs and then passes this handshake to the corresponding output. The inputs must be mutually exclusive.

B.2 Datapath components

B.2.1 Unary function

Implements single-operand operations such as invert. The handshake is simply passed through the component with the modified data.

B.2.2 Binary function

Implements two-operand operations such as addition, subtraction, comparisons and bit-wise boolean functions. The output request is forked to both inputs. The input acknowledges are synchronised and passed to the output.

B.2.3 CallMux

CallMux is used as a merge element in datapaths. Multiple push input channels can are merged onto a single output channel. The inputs must be mutually exclusive.
B.2.4 SplitEqual

Splits the data on its input port to multiple chunks of the same width, one chunk being sent on each output.

![Diagram of SplitEqual]

B.2.5 CaseFetch

When CaseFetch receives a request on its output, it pulls an index and uses this to decide which of its input ports to pull data on and then passes this data to the output port.

![Diagram of CaseFetch]

B.2.6 PassivatorPush

Used to connect an active output port from one process to the active input port of another process. See also section 3.3.13.

![Diagram of PassivatorPush]

B.2.7 Variable

The Variable component has a single write port and multiple read ports. It stores data that it receives on the write port and provides it to the read ports on request. Reads and writes must not occur concurrently.

![Diagram of Variable]
B.3 Control to datapth interface components

B.3.1 Fetch

Upon activation, the Fetch component pulls data on its input port and then pushes it on the output.

B.3.2 While

Implements the guarded loop language construct. When it is activated the While component pulls a single bit data item from its guard port. If the guard is true then While produces an output activation. When this activation has been acknowledged, While pulls another guard and repeats the process until a guard that is false is received.

B.3.3 Case

Upon activation, the Case component pulls a guard on its data port. It then activates one of its outputs based on the data that was received. Multiple values can be mapped to each output. If some values are not mapped to an output they will result in no output activation.
B.3.4 FalseVariable

Upon activation, the FalseVariable pulls data on its write port. It then holds this handshake open and activates the signal port. The FalseVariable acts as a Variable component, supplying, on request, the data from the write port to a set of read ports. When the signal handshake is completed (by the activated command), the write data is released.

![Diagram of FalseVariable]

B.3.5 activeEagerFalseVariable

The activeEagerFalseVariable has an active input port and a trigger port to activate it. As opposed to a FalseVariable, its signal output activates as soon as the trigger is activated, without waiting for data arrival.

![Diagram of activeEagerFalseVariable]
Appendix C

*FV* and *aeFV* implementations

The following pages show the implementation and STG of the *FalseVariable* (*FV*) and *activeEagerFalseVariable* (*aeFV*) components.
Figure C.1: *False Variable*: (a) Implementation, (b) STG.
Figure C.2: activeEagerFalseVariable: (a) Implementation, (b) STG.
Appendix D

Optimised Viterbi decoder Balsa description

The following pages show the Balsa source files for this design.
import [balsa.types.basic]
import [BMU] -- name of the file
import [PMU]
import [HU]
import [def_2]

procedure ViterbiDecoder_k2(
    input in_a : 3 bits;
    input in_c : 3 bits;
    output Out_state : State;
    output out_o : 1 bits
) is

array 0..3 of channel data_BMU_PMU : nibble
channel data_PMU_HU : Bundle_PMU_HU

begin
  BMU
    -- Input
    in_a,
    in_c,
    -- Output
    data_BMU_PMU ) ||
  PathMetricUnit (  
    -- Input
    data_BMU_PMU,
    -- Output
    data_PMU_HU ) ||
  HistoryUnit (  
    -- Input
    data_PMU_HU,
    -- Output
    Out_state,
    out_o )
end -- ViterbiDecoder_k2
The University of Manchester
-- School of Computer Science
-- Advanced Processors Technology (APT) group
--
-- Asynchronous Viterbi Decoder r=1/2, k=3 (4 states)
-- Author: Luis Tarazona (based on original description by Fabien Gavant)
-- 10/09/2008
-- data types

import[balsa.types.basic]

type bit3 is 3 bits -- new type for the input: 0 to 7

type State is enumeration
  S0, S1, S2, S3
end

type trellisState is enumeration
  LS0, LS1, LS2, LS3,
  US0, US1, US2, US3
end

type Bundle_BMU_PMU is record
  a_c : nibble; -- 4 bits
  a_d : nibble;
  b_c : nibble;
  b_d : nibble
end

type Bundle_PMU_HU is record
  Global_winner_found : 1 bits;
  Global_winner : State;
  dir_S0 : 1 bits;
  dir_S1 : 1 bits;
  dir_S2 : 1 bits;
  dir_S3 : 1 bits
end
import[balsa.types.basic]
import[def_2]

procedure smaller ( input x : nibble; input y : nibble; output o : nibble ) is
begin
x, y ->!
if (x < y) then
  o <- x
else
  o <- y
end -- if
end -- x,y ->!
end -- procedure smaller

procedure BMU( input a : bit3; -- 3bits
input c : bit3; array 0..3 of output Out_BMU : nibble -- Bundle_BMU_PMU
output a_c : nibble; -- 4bits
output a_d : nibble;
output b_c : nibble;
output b_d : nibble ) is
constant a_c = 0 : 2 bits
constant a_d = 1 : 2 bits
constant b_c = 2
constant b_d = 3

begin
loop
a, c ->!
  b <- (? - a as nibble) ||
  d <- (? - c as nibble) ||
  b,d ->!
end -- loop
end -- procedure BMU
BMU.balsa

```
66     ta_c <- (a + c as nibble) ||
67     ta_d <- (a + d as nibble) ||
68     tc_b <- (c + b as nibble) ||
69     tb_d <- (b + d as nibble)
70     end == b, d ->! then
71     end == a, c ->!
72     end || == loop
73     loop
74     ta_c, ta_d, tc_b, tb_d ->! then
75     ta_c1 <- ta_c ||
76     ta_d1 <- ta_d ||
77     tc_b1 <- tc_b ||
78     tb_d1 <- tb_d ||
79     smaller(ta_c1, ta_d1, c0) ||
80     smaller(tc_b1, tb_d1, c1) ||
81     smaller(c0, c1, smallest) ||
82     smallest ->! then
83     Out_BMU[a_c] <- (ta_c - smallest as nibble) ||
84     Out_BMU[a_d] <- (ta_d - smallest as nibble) ||
85     Out_BMU[b_c] <- (tc_b - smallest as nibble) ||
86     Out_BMU[b_d] <- (tb_d - smallest as nibble)
87     end == smallest ->!
88     end == ta_c ... tb_d ->!
89     end == loop
90     end == procedure BMU
91```
import(balsa.types.basic)
import(def_2)

type word32 is 32 bits

type word5 is 5 bits -- 0 to 31

type word6 is 6 bits -- 0 to 63

constant lower=0
constant upper=1

procedure smaller6 (
  input x : word6;
  input y : word6;
  output o : word6
) is
begin
  loop
    x, y ->!
    if (x < y) then
      o <- x
    else
      o <- y
    end -- if (x < y)
  end -- loop
end -- procedure smaller

procedure ACSUnit (
  input WState_A : word6;
  input BMU_A : nibble;
  input WState_B : word6;
  input BMU_B : nibble;
  output WState_O : word6;
  output direction : bit;
  output isZero : bit
) is
begin
  loop
    WState_A, BMU_A, WState_B, BMU_B ->
    WA <- (WState_A + BMU_A as word6) ||
    WB <- (WState_B + BMU_B as word6) ||
    WA, WB ->!
    if(WA <= WB) then
      WState_O <- WA ||
      direction <- lower ||
      isZero <- (WA = 0)
    else
      WState_O <- WB ||
      direction <- upper ||
      isZero <- (WB = 0)
    end -- if(WA <= WB)
  end -- WA, WB ->
end
end -- WState_A, BMU_A, WState_B, BMU_B -> then
end -- loop
end -- procedure ACSUnit

procedure reduction(
array 0..3 of input WMSa : word6;
array 0..3 of input WMSb : word6;
array 0..3 of output NWMS : word6
) is

local
channel smallest, smallest1, smallest2 : word6

begin

smaller6(WMSa[0], WMSa[1], smallest1) ||
smaller6(WMSa[2], WMSa[3], smallest2) ||
smaller6(smallest1, smallest2, smallest) ||
loop

smallest, WMSb[0], WMSb[1], WMSb[2], WMSb[3] ->! then

for [] i in 0..3 then

NWMS[i] <- (WMSb[i] - smallest as word6)
end -- for [] i

end -- smallest ... WMSb[3] ->!

end -- loop
end -- procedure reduction

procedure trellis(
<-- input WMS0 : word6 -- Weight MemState 0
input WMS1 : word6 -- Weight MemState 1
input WMS2 : word6 -- Weight MemState 2
input WMS3 : word6 -- Weight MemState 3
<--
array 0..3 of input wMS : word6;
<--
input a_c : nibble
input a_d : nibble
input b_c : nibble
input b_d : nibble
<--
array 0..3 of input bM : nibble;
array 0..3 of output wmA : word6;
array 0..3 of output wmB : word6;
array 0..3 of output bmA : nibble;
array 0..3 of output bmB : nibble
) is

begin

loop

wMS[0] ->! then

wmA[0] <- wMS[0] ||
wmA[1] <- wMS[0]
end
end ||

loop

wMS[1] ->! then

end
end ||

loop

wMS[2] ->! then

wmB[0] <- wMS[2] ||
end

procedure pmBuff(
  input i : word6;
  output oa : word6;
  output ob : word6
) is

  variable b : word6
  begin
    oa <- 0;  -- initial value
    ob <- 0;  -- initial value
    loop
      i -> b;
      oa <- b;
      ob <- b
    end
  end

procedure globalWinner(
  array 0..3 of input isZero : bit;
  output globalWinner : State;
  output globalWinner_found : bit
) is

  begin
    loop
      isZero[0], isZero[1], isZero[2], isZero[3] ->! then
        case (isZero[0] @ isZero[1] @
               isZero[2] @ isZero[3] as 4 bits) of
          0b0001 then
            globalWinner <- S0   ||
            globalWinner_found <- 1
          0b0010 then
            globalWinner <- S1   ||
            globalWinner_found <- 1
          end
    end
  end

loop
  wMS[3] ->! then
  end
  loop
  bm[0] ->! then -- a_c
    bmA[0] <= bm[0]   ||
    bmB[1] <= bm[0]
  end
  loop
  bm[1] ->! then -- a_d
  end
  loop
  bm[2] ->! then -- b_c
    bmB[0] <= bm[2]
  end
  loop
  bm[3] ->! then -- b_d
    bmA[0] <= bm[3]   ||
  end
end

end -- procedure trellis

procedure pmBuff(1)
Chapter D  Optimised Viterbi decoder Balsa description

PMU.balsa

| 196 | \texttt{0b100 then} |
| 197 | \texttt{globalWinner <- S2} |
| 198 | \texttt{globalWinner_found <- 1} |
| 199 | \texttt{0b1000 then} |
| 200 | \texttt{globalWinner <- S3} |
| 201 | \texttt{globalWinner_found <- 1} |
| 202 | \texttt{else} |
| 203 | \texttt{globalWinner <- S0} |
| 204 | \texttt{globalWinner_found <- 0} |
| 205 | end -- case |
| 206 | end -- isZero ->!
| 207 | end -- loop |
| 208 | end -- procedure GlobalWinner |
| 209 | procedure PathMetricUnit( |
| 210 | array 0..3 of input Out_BMU : nibble; |
| 211 | output Out_PMU : Bundle_PMU_HU |
| 212 | is |
| 213 | -- trellis to ACS i/f |
| 214 | array 0..3 of channel wmA : word6 |
| 215 | array 0..3 of channel wmB : word6 |
| 216 | array 0..3 of channel bmA : nibble |
| 217 | array 0..3 of channel bmB : nibble |
| 218 | -- ACS to output i/f |
| 219 | array 0..3 of channel direction : bit |
| 220 | -- ACS to buffer i/f |
| 221 | array 0..3 of channel WState_0 : word6 |
| 222 | -- ACS to globalWinner i/f -- ACS to globalWinner i/f |
| 223 | array 0..3 of channel isZero : bit --for global winner check |
| 224 | -- buffer to reduction i/f |
| 225 | array 0..3 of channel WState_Oa : word6 |
| 226 | array 0..3 of channel WState_Ob : word6 |
| 227 | -- reduction to trellis i/f |
| 228 | array 0..3 of channel WState : word6 |
| 229 | channel globalWinner : State |
| 230 | channel globalWinner_found : bit |
| 231 | begin |
| 232 | trellis(WState, Out_BMU, wmA, wmB, bmA, bmB) [] |
| 233 | for | i in 0..3 then |
| 234 | ACSUnit(wmA[i], bmA[i], wmB[i], bmB[i], |
| 235 | WState_O[i], direction[i], isZero[i]) |
| 236 | end [][] |
| 237 | globalWinner(isZero, globalWinner, globalWinner_found) [] |
| 238 | for | i in 0..3 then |
| 239 | pmBuff(WState_O[i], WState_Oa[i], WState_Ob[i]) |
| 240 | end [][] |
| 241 | reduction(WState_Oa[i], WState_Ob, WState) [] |
| 242 | loop |
| 243 | globalWinner, globalWinner_found, |
| 244 | direction[0], |
| 245 | direction[1], |
| 246 | direction[2], |
| 247 | direction[3] ->! then |
| 248 | Out_PMU <- \{globalWinner_found, |
| 249 | globalWinner, |
| 250 | direction[0], |
| 251 | direction[1], |
| 252 | direction[2], |
| 253 | direction[3]\}
| 254 | end -- globalWinner ->!
| 255 | end -- loop |
| 256 | end -- procedure PathMetricUnit |
import [balsa.types.basic]
import [def_2]

type A4_t is array 4 of bit

type testbit is bit

procedure HistoryHunit(
  input In_HU : Bundle_PMU_HU;
  output Out_state : State;
  output Data_out : bit
) is

variable Temp : Bundle_PMU_HU

variable Global_Winner_Valid : array 0..15 of bit
variable Global_Winner : array 0..15 of State
variable Global_Winner_Head : State
variable DL_Winner : array 0..15 of 4 bits -- Direction_Local_Winner
variable Head, pHead, nHead : 4 bits -- Head of the current time slot
variable Parent, pParent : 4 bits -- for the reconstruction of the path
channel GW_single : 2 bits -- State 2 bits S0=00, S1=01, S2=10, S3=11
variable Temp_state : State
variable Token : bit
channel Return_direction : array 0..3 of bit
variable Var_div1 : State
channel Var_div2 : State
-- for start with valid value
variable Start, doLoop : bit
variable Safeguard, nSafeguard : 4 bits -- for the begining
variable i : 4 bits

begin
  -- initialisation
  Head := 0
  pHead := (0 - 1 as 4 bits)
  Start := 0

  loop
    Child := pHead
    Parent := Head
    Token := 0
    i := (0 - 1 as 4 bits)
    -- generates the output state when start is ready
--- (good value to release)

if Start then

    Out_state <- Global_Winner_Head

else
    Safeguard := Head
end

if In_HU ->! then -- read data & control

    -- store data on the memory
    DL_Winner[Head] := (A4_t {In_HU.dir_S0,
        In_HU.dir_S1,
        In_HU.dir_S2,
        In_HU.dir_S3} as 4 bits)

    -- I update all the data
    Global_Winner[Head] := In_HU.Global_winner
    Global_Winner_Valid[Head] := In_HU.Global_winner_found
    doLoop := In_HU.Global_winner_found
end

if doLoop then

    -- reconstruction of the path
    -- save the GW
    GW_single <-(Global_Winner[Parent] as 2 bits)

    -- load the direction of the Local_Winner
    Return_direction <-(DL_Winner[Parent]
        as array 4 of bit)

    GW_single, Return_direction ->! then
        case (#GW_single @ (Return_direction[GW_single])
            as trellisState) of
            LS0, LS1 then
                Var_div2 := S0
            [LS2, LS3 then
                Var_div2 := S1
            [US0, US1 then
                Var_div2 := S2
            [US2, US3 then
                Var_div2 := S3
            end
        end

    Var_div2 ->! then
        if {Var_div2 = Global_Winner[Child]
            and Global_Winner_Valid[Child] = 1
            or Child = Head} then
            Token := 1
        else
            Var_div1 := Var_div2
        end

    pParent := Child

    pChild := (Child - 1 as 4 bits)
    while (Token = 0 and Safeguard /= i) then -- Condition
        i := (i + 1 as 4 bits)

    Child := pChild
Parent := pParent

||
| if not Token then
| Global_Winner[pParent] := Var_div1
end
end --loop while
end -- if doLoop

pHead := Head

||
nHead := (Head + 1 as 4 bits)
;
Head := nHead

||
if nHead = 15 then
  Start := 1
end --if Head=15

||
Global_Winner_Head := Global_Winner[nHead]
end --loop
end -- HistoryHunit
Appendix E

Optimised 32x32 bit Booth multiplier Balsa description

The following pages show the Balsa source files for this design.
import [balsa.types.basic]
import [nanoMulTypes]
import [nanoMultSupport]
import [nanoMBoothR3rolled]

procedure CSAdder_DP2 is
  CSAdder( Datapath_2 );

procedure CPadder is
  fullCPadder( Datapath );

procedure nanoMultiplier
  (input bypass : bit;
   input bypassH : bit;
   input mType : MulType;
   input a : Datapath;
   input b : Datapath;
   input c : Datapath;
   output mpH : Datapath;
   output mpL : Datapath;
   output mZ : bit;
   output mN : bit)
  is
    -- length and multiply-accumulate control words
    channel mlength : bit
    channel macc : bit
    -- sign adjust I/F input
    channel ba : Datapath
    channel bb : Datapath
    channel bc : Datapath
    channel bmType : MulType
    -- sign adjust I/F input
    channel sa : Datapath_2
    channel sb : Datapath_3
    channel sc : Datapath_2
    -- CS adder I/F
    channel opA : Datapath_2
    channel opB : Datapath_2
    channel cs : Datapath_2
    channel cin : Datapath_2
    channel res : Datapath_2
    -- CP adder I/F
    channel raA : Datapath
    channel raB : Datapath
    channel rac0 : bit
    channel raS : Datapath
    channel racN : bit
    -- multiplier iteration control
    channel load : bit
    channel done : bit
    -- bypass interface
    channel pH : Datapath
    channel pL : Datapath
    channel z : bit
    channel n : bit
    channel bpH : Datapath
Chapter E  Optimised 32x32 bit Booth multiplier Balsa description

nanoMultiplier.balsa

```balsa
66  channel bpL : Datapath
67  channel bz : bit
68  channel bn : bit
69  channel bH, bL : bit
70  begin
71     CSAdder_DP2(opA,opB,cs,cin,res)
72     |   CPadder(raA,raB,raS,raC0,raC1)
73     |   nanoMBoothR3rolled(cin,res, sa, sb, sc, mlength, macc,
74     |       load,done,opA,opB,cs,raA,raB,raC0,raS,raC1,pH,pL,z,n)
75     |   mControl(10,load,done)
76     |   signAdj(bmType, ba, bb, bc, sa, sb, sc, mlength, macc)
77     |   bypassMul(bypass, bypassH, a, b, c, mType, ba, bb, bc, bmType, bH, bL)
78     |   doByPass(bH, bL, pH, pL, z, n, mpH, mpL, mH, mN)
79  end
```
nanoMulTypes.balsa

-- The University of Manchester
-- School of Computer Science
-- Advanced Processors Technology (APT) group
--
-- Radix-3 Booth's multiplier for nanoSpa/aviSpa processor in Balsa
--
-- Author: Luis Tarazona tarazonl@cs.man.ac.uk
-- v1.0 20/04/2007 -tarazonl
--
-- reduced nanoSpaTypes file for nanoMultiplier only

1  type signedByte is 8 signed bits
2  type SignedHalfWord is 16 signed bits
3  type Address is 32 bits
4  type Datapath is 32 bits
5  type signedDatapath is 32 signed bits
6
7  type Flags is record
8     V : bit;
9     C : bit;
10    Z : bit;
11    N : bit;
12  end -- type Flags
13
14  type MulType is enumeration
15     MUL=0, -- multiply (32-bit result)
16     MLA=1, -- multiply-accumulate (32-bit result)
17     MUND2=2, -- undefined code
18     MUND3=3, -- undefined code
19     UMLAL=4, -- unsigned multiply-accumulate long
20     UMULL=5, -- unsigned multiply-accumulate long
21     SMULL=6, -- signed multiply long
22     SMLAL=7 -- signed multiply-accumulate long
23
24  constant length = sizeof Datapath
25  constant xlength = sizeof Datapath + 1
26  constant tbits = log (sizeof Datapath)
27  type cntType is tbits bits
28  type Datapath_1 is length+1 bits
29  type Datapath_2 is xlength bits
30  type Datapath_3 is xlength+1 bits
31
32  type sDatapath is length signed bits
33  type sDatapath_1 is length+1 signed bits
34  type sDatapath_2 is xlength signed bits
35  type sDatapath_3 is xlength+1 signed bits
import [balsa.types.basic]
import [nanoMulTypes]

procedure signAdj
(input mType : MulType;
 input a : Datapath;
 input b : Datapath;
 input c : Datapath;
 output aa : Datapath_2;
 output ba : Datapath_3;
 output ca : Datapath_2;
 output length : bit;
 output macc : bit
) is
begin
loop
mType,a,b,c ->! then
-- Handle signed/unsigned in a,b operands,
-- also add 0 to lsb of multiplier (b operand)
case mType of
 MUL,
 UMULL,
 UMLAL
 then
-- unsigned, always fill with zeroes
aa <- (a as Datapath_2)

]| |
ba <- (#0b[0..0] @ #b[0 .. length-1] as Datapath_3)
else -- signed, extend sign
  aa <- (((a as sDatapath) as sDatapath_2) as Datapath_2)
  |
  ba <- (((#0b[0..0] @ #b[0 .. length-1]
           as sDatapath_1) as sDatapath_3) as Datapath_3)
end -- case mCode
-- Handle accumulate.
-- 'c' operand does not need sign extension, fill with zeroes
  |
  ca <- (c as Datapath_2)
  |
  length <- (#mType[2..2] as bit) -- long = 1 / short = 0
  |
  macc <- (#mType[0..0] as bit) -- acc = 1
end -- mType ->
end -- loop
end -- procedure signAdj

procedure doByPass(
 input bh : bit;
 input bl : bit;
 input bpH : Datapath;
 input bpL : Datapath;
 input bmZ : bit;
 input bmN : bit;
 output mpH : Datapath;
 output mpL : Datapath;
 output mz : bit;
 output mn : bit)
nanoMultSupport.balsa

begin
loop
  bH, bL ->!
  if bL then
    mL <= 0
    mZ <= 0
    mN <= 0
    if bH then
      mpH <= 0
    end
  else
    bpL -> mpL
    bmZ -> mZ
    bmN -> mN
    if bH then
      bpH -> mpH
    end
  end
end
-- bypasses multiplier if kill order is sent
procedure bypassMul(
  input bypass : bit;
  input bypassH : bit;
  input mulOpA : Datapath;
  input mulOpB : Datapath;
  input mulOpC : Datapath;
  input mulType : MulType;
  output mulOpAo : Datapath;
  output mulOpBo : Datapath;
  output mulOpCo : Datapath;
  output mulTypeo : MulType;
  output bH : bit;
  output bL : bit)
  is
loop
  bypass, bypassH, mulType ->!
  mulOpA, mulOpB ->!
  if bypass then
    case mulType of MLA, UMLAL, SMLAL then -- accumulate
      mulOpC ->!
    end
  else
    continue
  end
else
  mulOpAo <= mulOpA
  mulOpBo <= mulOpB
  mulTypeo <= mulType
  case mulType of MLA, UMLAL, SMLAL then -- accumulate
    mulOpC ->!
  end

nanoMultSupport.balsa

mulOpCo <= mulOpC
else
  mulOpCo <= 0
end
end -- if bypass
end -- procedure bypassMul

-- carry save adder
procedure CSAdder
  ( parameter DataType : type;
    input a : DataType;
    input b : DataType;
    input cs : DataType;
    output cout: DataType;
    output s : DataType
  ) is
local
  begin
    loop
    a,b,cs ->! then
      s <= a xor b xor cs
      end
end -- procedure CSAdder

-- carry propagate adder
procedure fullCPadder
  ( parameter DataType : type;
    input a : DataType;
    input b : DataType;
    input c0 : bit;
    output s : DataType;
    output CN : bit
  ) is
local
  constant DTLength = sizeof DataType
  type eDataType is DTLength + 1 bits
  type eeDataType is DTLength + 2 bits
  channel ea, eb : eDataType
  channel es : eeDataType
begin
  loop
    a,b,c0 ->! then
      ea <= (#c0[0..0] @ #a[0..DTLength-1] as eDataType)
      eb <= (#c0[0..0] @ #b[0..DTLength-1] as eDataType)
      end
end
nanoMultSupport.balsa

196  ||
197   es ->! then
198   s <- (#es[1..DTLength] as DataType)
199   ||
200  cN<- (#es[DTLength+1 .. DTLength+1] as bit)
201  end -- es ->!
202  end -- loop
203  end -- procedure fullCPadder
204
205  -- shift register that controls iteration
206  procedure mControl
207  { parameter cLength : cardinal;
208    input load : bit;
209    output done : bit
210  ) is
211
212  variable t : bit
213  variable c0 : cLength bits
214  variable cl : cLength bits
215
216  begin
217    loop
218      load ->! then
219      t := load
220      end -- load ->
221      ;
222      if t then
223        c0 := (2**(cLength-1) - 1 as cLength bits)
224        ||
225        done <- 1
226      else
227        done <- (#c0[0..0] as bit)
228        ||
229        cl := (#c0[1..cLength-1] as cLength bits)
230      ;
231      c0 := cl
232    end --if t
233  end -- loop
234  end -- procedure mControl
235
236  procedure mControl10 is mControl(10)
import [balsa.types.basic]
import [nanoMulTypes]

procedure nanoMBoothR3rolled
(input cin : Datapath_2;
input res : Datapath_2;
input a : Datapath_2;
input b : Datapath_3;
input c : Datapath_2;
input mlength : bit;
input macc : bit;
output load : bit;
output opA : Datapath_2;
output opB : Datapath_2;
output cs : Datapath_2;
output raA : Datapath;
output raB : Datapath;
output rac0 : bit;
input raS : Datapath;
input racN : bit;
output pH : Datapath;
output pL : Datapath;
output z : bit;
input done : bit)

is
local
channel sout : Datapath_2
channel csout : Datapath_2
channel c0 : bit

variable ctrl : 4 bits
variable vph : Datapath
variable vpl : Datapath
variable vA : Datapath_2
variable v2a : Datapath_2
variable v3a : Datapath_2
variable v4a : Datapath_2
variable nva : Datapath_2
variable nV2a : Datapath_2
variable nV3a : Datapath_2
variable nV4a : Datapath_2
variable crh : Datapath_2
variable crl : Datapath_3
variable rh : Datapath_2
variable rl : Datapath_3
variable rhp : Datapath_2
variable crhp : Datapath_2
nanoMBoothR3rolled.balsa

66 variable crlp : Datapath_3
67 variable rlp : Datapath_3
68 variable go : 1 bits
69 variable vmlength : bit
70 variable vmacc : bit
71 variable vZ : bit
72 variable VN : bit
73
74 begin
75 loop --main
76 | a,b,c,mlength, macc ->! then
77 | vmlength := mlength
78 | vmacc := macc
79 | va := a
80 | v2a := (#0b0[0..] @ #a[0..xlength-2] as Datapath_2)
81 | v4a := (#0b00 as 2 bits)[0..1] @ #a[0..xlength-3] as Datapath_2
82 | -- calculate 3A = 2A + A
83 | raA <- (#a[1..length] as Datapath) -- a without b0
84 | raB <- (a as Datapath) -- 2a without b0
85 | rac0 <- 0
86 | rAs,racN ->! then
87 | v3a := (#a[0..] @ #raS[0..length-1] @ #racN[0..0] @ #a[length..length] as Datapath_2)
88 | end
89 | rlp := b
90 | rhp := c
91 | ctrl := (#b[0..3] as 4 bits)
92 | crhp := (0b0 as Datapath_2)
93 | crlp := (0b0 as Datapath_3)
94 | end -- a,b,c,mlength ->
95 | nva := not va
96 | nv2a := not v2a
97 | nv3a := not v3a
98 | nv4a := not v4a
99 | load <- 1
100 ;
101 loop --iterate
102 opA <- rhp
103 | cs <- crhp
104 | res -> sout
105 | cin -> csout
106 |}
nanoMBoothR3rolled.balsa

111 c0 <- (#ctrl[3..3] as bit)
112 |
113 case ctrl of
114 0b0001,0b0010 then --sout <= (rhp + va as Datapath_2)
115 | opB <= va
116 | 0b0011,0b0100 then -- sout <= (rhp + v2a as Datapath_2)
117 | opB <= v2a
118 | 0b0111 then -- sout <= (rhp + v4a as Datapath_2)
119 | opB <= v4a
120 | 0b1000 then -- sout <= (rhp + v4a + 1 as Datapath_2)
121 | opB <= v4a
122 | 0b1001,0b1010 then -- sout <= (rhp + v3a + 1 as Datapath_2)
123 | opB <= v3a
124 | 0b1011,0b1100 then -- sout <= (rhp + v2a + 1 as Datapath_2)
125 | opB <= v2a
126 | 0b1111 then -- sout <= (rhp + v4a + 1 as Datapath_2)
127 | opB <= v4a
128 | 0b1000 then -- sout <= (rhp + nv4a + 1 as Datapath_2)
129 | opB <= nv4a
130 | 0b1010,0b1001 then -- sout <= (rhp + nv3a + 1 as Datapath_2)
131 | opB <= nv3a
132 | 0b1100,0b1011 then -- sout <= (rhp + nv2a + 1 as Datapath_2)
133 | opB <= nv2a
134 | 0b1110,0b1101 then -- sout <= (rhp + nva + 1 as Datapath_2)
135 | opB <= nva
136 else
137 | opB <= (((ctrl as 4 signed bits)
138 | as sDatapath_2) as Datapath_2)
139 end -- case 2
140 141 -- shifter:
142 |
143 sout,c sout,c0 ->! then
144 145-- shift 2 times (rh arithmetic, but rl logic)
146 crh := csout
147 |
148 | crl := (#crlp[3..xlength-1] &
149 | #c0[0..0] & #csout[0..1] as Datapath_3)
150 |
151 | rh := (#sout[3..xlength-1] as xlength-3 signed bits)
152 | as sDatapath_2) as Datapath_2)
153 |
154 | rl := (#rlp[3..xlength-1] & #sout[0..2] as Datapath_3)
155 end -- sout ->
156 |
157 done ->! then
158 go := done
159 end
160 while go then-- while counter /= (length/2 + 1 as tbits bits)
162 #crh[xlength-1..xlength-1] as Datapath_2)
163 |
164 crlp := crl
165 |
166 rhp := rh
167 |
168 rlp := rl
169 |
170 ctrl := (#rl[0..3] as 4 bits)
171 |
172 load <= 0
173 end --loop iterate
174 |
175 raA <= (#rl[2..xlength+1] as Datapath)
176 |
177 raB <= (#crl[2..xlength+1] as Datapath)
178 |
179 rac0 <= 0
180 |
181 raS,racN ->! then
182 vpl:= raS
183 |
184 go := racN -- save carry for pH
nanoMBoothR3rolled.balsa

```cpp
if vmlength then -- calculate pH and produce two results + flags
    raA <- (rl[length + 2..length+2] as Datapath)
    raB <- (crh[1..length] as Datapath)
    rac0 <- go
    raS, racN ->!
    vph := raS --(raS[0..length-1] as Datapath)
end -- raS, raN ->!

pH <- vph
pL <- vpl
if vmacc then
    z <- vZ
else
    z <- (vph = 0 as bit) and vZ
end
n <- (#vph[31]as bit)
else -- only produce pL & flags
    pL <- vpl
    z <- vZ
    n <- vN
end -- if vmlength
end --loop main
end
```
Appendix F

Optimised sliced-channel wormhole router Balsa description

The following pages show the Balsa source files for this design.
import [balsa.types.basic]
import [arbiter]
import [input_buf]
import [crossbar]

procedure router (array 20 of input d_in : 9 bits;
array 20 of output d_out : 9 bits)

array 64 of channel data_m : 9 bits
array 16 of sync req
array 3 of channel cfg_lwe : 2 bits
array 2 of channel cfg_sn : 1 bits

begin
  input_buf_south(d_in[0..3], req[0..3], data_m[0..15])
  input_buf_west(d_in[4..7], req[4..5], data_m[16..23])
  input_buf_north(d_in[8..11], req[6..9], data_m[24..39])
  input_buf_east(d_in[12..15], req[10..11], data_m[40..47])
  input_buf_loc(d_in[16..19], req[12..15], data_m[48..63])
  arbiter_sn({req[1], req[14]}, cfg_sn[0])
  arbiter_lwe({req[0], req[7], req[10], req[13]}, cfg_lwe[0])
  arbiter_sn({req[2], req[4], req[8], req[15]}, cfg_lwe[1])
  arbiter_lwe({req[3], req[5], req[9], req[11]}, cfg_lwe[2])
  crossbar(data_m, cfg_lwe, cfg_sn, d_out)
end
import [balsa.types.basic]

procedure sub_arb_low (
  array 2 of sync req;
  output winner : 1 bits
) is
begin
  loop
    arbitrate req[0] then winner <= 0
    |    req[1] then winner <= 1
  end
end

procedure sub_arb (  
  array 2 of input req : 1 bits;
  output winner : 2 bits
) is
constant one = (1 as 1 bits)
begin
  loop
    arbitrate req[0] then winner <= (req[0] as 2 bits)
    |    req[1] then winner <= ((#(req[1]) @ #one) as 2 bits)
  end
end

procedure sub_arb_sync (  
  parameter Wi : byte;
  parameter Wo : byte;
  parameter Ds : byte;  -- the data when sync is selected
  input in0 : Wi bits;
  sync in1;
  output cfg : Wo bits
) is
begin
  loop
    arbitrate in0 then cfg <= (in0 as Wo bits)
    |    in1 then cfg <= (Ds as Wo bits)
  end
end

procedure arbiter_lwe (  
  array 4 of sync req;
  output cfg : 2 bits
) is
array 2 of channel arb_dir_sub : 1 bits
begin
  sub_arb_low(req[0..1], arb_dir_sub[0])
  |  sub_arb_low(req[2..3], arb_dir_sub[1])
  |  sub_arb(arb_dir_sub, cfg)
end
import [balsa.types.basic]

procedure input_buf_south (array 4 of input data_in : 9 bits; array 4 of sync req; array 16 of output data_out : 9 bits)

variable buf : array 4 of 9 bits
constant addrx = (2 as 4 bits)
constant addry = (2 as 4 bits)

procedure ibuf_demux (parameter X : byte; input data_in : 9 bits; input steer : 3 bits; array 4 of output data_out : 9 bits)

variable steerV : 3 bits

begin
  loop
    steer => steerV;
    loop
      data_in => buf[X];
      case steerV of
        0b1xx then
data_out[0] <= buf[X]
        | 0b01x then
data_out[1] <= buf[X]
        | 0b001 then
data_out[2] <= buf[X]
        else
data_out[3] <= buf[X]
      end
    end
  end
  while (#(buf[X])[8] as 1 bits) /= (1 as 1 bits)
end

array 4 of channel steer : 3 bits
channel n,e,w : bit
channel data_in0 : 9 bits
variable isTail : bit

begin
  loop
    data_in[0] =>! then
      n <= (#(data_in[0])[4..7] as 4 bits) < addrx
      e <= (#(data_in[0])[0..3] as 4 bits) > addrx
      w <= (#(data_in[0])[6..8] as 4 bits) < addrx
      data_in0 <= data_in[0]
      isTail := #(data_in[0])[8]
end

loop
  data_in[8] =>! then
data_in0 <= data_in[0]
  | isTail := #(data_in[0])[8]
while not isTail
end

loop
    n,e,w -> | then
        for | | i in 0..3 then
            steer[i] <- (n & e & w as 3 bits)
        end
        | case (n & e & w as 3 bits) of
            0b10x then sync req[1]
            0b01x then sync req[2]
            0b001 then sync req[0]
            else sync req[3]
        end
    end
    while (1 as 3 bits) /= (1 as 3 bits)
end

procedure input_buf_west (array 4 of input data_in : 9 bits;
array 2 of sync req;
array 8 of output data_out : 9 bits
) is
    constant addrx = (2 as 4 bits)
    constant addry = (2 as 4 bits)
    procedure ibuf_demux (parameter X : byte;
input data_in : 9 bits;
input steer : 1 bits;
array 2 of output data_out : 9 bits
) is
        variable steerV : 1 bits
begin
    loop
        steer -> steerV;
        loop
            data_in -> buf[X];
            case steerV of
                0b1 then
                    data_out[0] <- buf[X]
                else
                    data_out[1] <- buf[X]
                end
            end
        while (1 as 1 bits) /= (1 as 1 bits)
    end
end
array 4 of channel steer : bit
channel e : bit
input_bufv2.balsa

```
channel data_in0 : 9 bits
variable isTail : bit

begin
  loop
    data_in[0] ->! then
      e <- (data_in[0])[0..3] as 4 bits > addry
      data_in[0] <- data_in[0]
    end
    loop
      data_in[0] ->! then
        data_in0 <- data_in[0]
      end
      isTail := (data_in[0])[8]
    end
    while not isTail
    end
  end
  loop
    e ->! then
      for || i in 0..3 then
        steer[i] <- e
      end
      case e of
        0b1 then sync req[0]
        else sync req[1]
      end
    end
  end
  end

end

procedure input_buf_north (array 4 of input data_in : 9 bits;
array 4 of sync req;
array 16 of output data_out : 9 bits is
variable buf : array 4 of 9 bits
constant addrx = (2 as 4 bits)
constant addry = (2 as 4 bits)
)

procedure ibuf_demux (parameter X : byte;
input data_in : 9 bits;
input steer : 3 bits;
array 4 of output data_out : 9 bits ) is
variable steerV : 3 bits
begin
  loop
    steer -> steerV;
    loop
      data_in -> buf[X];
      case steerV of
        0b1xx then
          data_out[0] <- buf[X]
        | 0b01x then
          data_out[1] <- buf[X]
        | 0b001 then
```
begin

loop

for ||| i in 0..3 then

steer[i] <= (#w @ #e @ #s as 3 bits)
end

end

for ||| i in 1..3 then

ibuf_demux(i, data_in[i], steer[i], {data_out[i], data_out[i+8], data_out[i+4], data_out[i+12]})
end

end

procedure input_buf_east ( array 4 of input data_in : 9 bits;
array 2 of sync req;
array 8 of output data_out : 9 bits ) is

variable buf : array 4 of 9 bits
constant addrx = {2 as 4 bits}
custom addry = {2 as 4 bits}

procedure ibuf_demux (
parameter X : byte;
input data_in : 9 bits;
input steer : 1 bits;
array 2 of output data_out : 9 bits)
is
variable steerV : 1 bits
begin
loop
steer -> steerV;
loop
data_in -> buf[X];
case steerV of
  0b1 then
data_out[0] <-> buf[X]
else
data_out[1] <-> buf[X]
end
while (#(buf[X])$ as 1 bits) /= (1 as 1 bits)
end
end

array 4 of channel steer : bit
channel w : bit
channel data_in0 : 9 bits
variable isTail : bit
begin
loop
data_in[0] !-> then
  w <= #(data_in[0])0..3 as 4 bits) < addr
  ||
  data_in0 <- data_in[0]
end
loop
data_in[0] !-> then
  data_in0 <- data_in[0]
  ||
  isTail := #(data_in[0])8
end
while not isTail
end

procedure input_buf_loc {
array 4 of input data_in : 9 bits;
array 4 of output data_out : 9 bits;
array 4 of sync req;
```haskell
array 16 of output data_out : 9 bits

variable buf : array 4 of 9 bits
constant addrx = (2 as 4 bits)
constant addry = (2 as 4 bits)

procedure ibuf_demux (parameter X : byte;
input data_in : 9 bits;
input steer : 3 bits;
array 4 of output data_out : 9 bits)

) is
variable steerV : 3 bits
begin
loop steer -> steerV;
  loop
    data_in -> buf[X];
    case steerV of
    0b1xx then
    data_out[0] <- buf[X]
    | 0b0lx then
    data_out[1] <- buf[X]
    | 0b00l then
    data_out[2] <- buf[X]
    else
    data_out[3] <- buf[X]
end while (#(buf[X])[8] as 1 bits) /= (1 as 1 bits)
end
end

array 4 of channel steer : 3 bits
channel s,n,e : bit
channel data_in0 : 9 bits
variable isTail : bit

begin
loop data_in[0] ->! then
  s <= (#(data_in[0])[4..7] as 4 bits) > addrx
  | n <= (#(data_in[0])[4..7] as 4 bits) < addrx
  | e <= (#(data_in[0])[0..3] as 4 bits) > addry
  | data_in0 <= data_in[0]
end ;
loop data_in[0] ->! then
  data_in0 <= data_in[0]
  | isTail := #(data_in[0])[8]
end while not isTail
end

| loop s,n,e ->! then
  for | i in 0..3 then
    steer[i] <= (#e @ #n @ #s as 3 bits)
  end case (#e @ #n @ #s as 3 bits) of
  0b1xx then sync reg[0]
  | 0b0lx then sync reg[2]
  | 0b00l then sync reg[3]
```

else
    sync req[1]
end
end

||
for || i in 1..3 then
    ibuf_demux(i, data_in[i], steer[i],
        {data_out[i], data_out[i+8],
        data_out[i+12],data_out[i+4]})
end

||
ibuf_demux(0, data_in0, steer[0],
    {data_out[0], data_out[8], data_out[12],data_out[4]})
end
import [balsa.types.basic]

procedure obuf_mux2 (array 2 of input data_in : 9 bits;
array 2 of input tail : bit;
input steer : bit;
output data_out : 9 bits
) is
variable steerV : bit
variable buf : 9 bits
variable isTail : array 0..1 of bit
begin
  loop
  steer -> steerV;
  case steerV of
  for i in 0..1 then
    loop
    tail[i] -> isTail [i]
    || data_in[i] -> data_out
    while not isTail[i] end
  end
end
end

procedure obuf_mux4 (array 4 of input data_in : 9 bits;
array 4 of input tail : bit;
input steer : 2 bits;
output data_out : 9 bits
) is
variable steerV : 2 bits
variable buf : 9 bits
variable isTail : array 0..3 of bit
begin
  loop
  steer -> steerV;
  case steerV of
  for i in 0..3 then
    loop
    tail[i] -> isTail [i]
    || data_in[i] -> data_out
    while not isTail[i] end
  end
end
end

procedure outbuffer (input data_in : 9 bits;
output tail : bit;
output data_out : 9 bits
) is
variable buf : 9 bits
begin
  loop
  data_in -> buf;
data_out <- buf || tail <- (#buf[8] as bit)

procedure sub_crossbar_lwe (array 16 of input data_in : 9 bits;
input cfg : 2 bits;
array 4 of output data_out : 9 bits
) is
  -- variable cfg_m : 2 bits
  array 4 of channel cfg_m : 2 bits
  array 16 of channel tail : bit
  array 16 of channel bdata_in : 9 bits
begin
  loop
    for i in 0..3 then
      cfg_m[i] <- cfg
    end
  end
  [for || i in 0..3 then
    obuf_mux4{
      { bdata_in[i], bdata_in[i+4], bdata_in[i+8], bdata_in[i+12] },
      { tail[i], tail[i+4], tail[i+8], tail[i+12] },
      cfg_m[i],
      data_out[i]
    }
  ]
  [for || i in 0..15 then
    outbuffer(data_in[i], tail[i], bdata_in[i])
  end
  end

procedure sub_crossbar_sn (array 8 of input data_in : 9 bits;
input cfg : bit;
array 4 of output data_out : 9 bits
) is
  -- variable cfg_m : bit
  array 4 of channel cfg_m : bit
  array 8 of channel tail : bit
  array 8 of channel bdata_in : 9 bits
begin
  loop
    for i in 0..3 then
      cfg_m[i] <- cfg
    end
  end
  [for || i in 0..3 then
    obuf_mux2{
      { bdata_in[i], bdata_in[i+4] },
      { tail[i], tail[i+4] },
      cfg_m[i],
      data_out[i]
    }
  ]
  [for || i in 0..7 then
    outbuffer(data_in[i], tail[i], bdata_in[i])
  end
  end

procedure crossbar (}
array 64 of input data_in : 9 bits;
array 3 of input cfg_lwe : 2 bits;
array 2 of input cfg_sn : 1 bits;
array 20 of output data_out : 9 bits
)
begin
  sub_crossbar_sn(data_in[24..27])
    @ data_in[48..51], cfg_sn[0], data_out[0..3])
  sub_crossbar_lwe(data_in[0..3] @ data_in[28..31])
    @ data_in[40..43] @ data_in[52..55],
    cfg_lwe[0], data_out[4..7])
  sub_crossbar_sn(data_in[4..7])
    @ data_in[56..59], cfg_sn[1], data_out[8..11])
  sub_crossbar_lwe(data_in[8..11] @ data_in[16..19])
    @ data_in[32..35] @ data_in[60..63],
    cfg_lwe[1], data_out[12..15])
  sub_crossbar_lwe(data_in[12..15] @ data_in[20..23])
    @ data_in[36..39] @ data_in[44..47],
    cfg_lwe[2], data_out[16..19])
end
Appendix G

Optimised nanoSpa forwarding unit Balsa description

The following pages show the Balsa source files for this design.
procedure nanoForwardUnit

{ -- Allocation pointers
input allocPtr1 : allocPtrType;
input allocPtr2 : allocPtrType;
input arrPtr1 : WROBSIZE bits;
input arrPtr2 : WROBSIZE bits;
array 2 of input doAlloc : bit;
array 2 of input doArrival : bit;
array 2 of input invalidIn : bit;
-- results from Execute
input wrdata0 : Datapath;
input wrdata1 : Datapath;
-- destination
array 2 of input wraddr : RegSpec;
-- lookup interface
array READPORTS of input readIn : bit;
array READPORTS of input raddr : RegSpec;
-- register read interface
array READPORTS of output readReg : bit;
array READPORTS of output fwReg : bit;
array READPORTS of output fdata : Datapath;
-- writeback interface
output wbaddr : RegSpec;
output wbdata : Datapath
} is

-- the buffer cells
-- the buffer data structure
variable bvaddr : array ROBSIZE of RegSpecExt
variable bdata : array ROBSIZE of Datapath
variable bpos : array ROBSIZE of bit
-- extended reg addresses for lookup
array READPORTS of channel addr : RegSpecExt
-- lookup-forward i/f
array READPORTS of channel posMask : 4 bits
array READPORTS of channel foundMask : 4 bits
--
-- steerAlloc - mux allocCells i/f
array ROBSIZE of channel age1 : bit
array ROBSIZE of channel sAddr1 : RegSpec
-- steerAllocX - mux allocCells i/f
array ROBSIZE of channel age2 : bit
array ROBSIZE of channel sAddr2 : RegSpec
array ROBSIZE of channel invalid2 : bit
--
-- mux allocCells - allocCells i/f
array ROBSIZE of channel ageM : bit
array ROBSIZE of channel aAddrM : RegSpec
array ROBSIZE of channel invalidM : bit
--
-- mux arriveCells - arriveCells i/f
array ROBSIZE of channel wbdataM : Datapath
array ROBSIZE+1 of channel Tin : bit
array ROBSIZE of channel Tout : bit
-- arriveCells - mux writeback i/f
array ROBSIZE of channel wbdataC : Datapath
array ROBSIZE of channel wbaddrC : RegSpec
--
-- writeback network
array ROBSIZE+1 of channel wbToken : bit
array ROBSIZE+1 of channel allocToken : bit
array ROBSIZE of channel allocC : bit
-- internal forward & register read control
array READPORTS of channel doFwd : bit
array READPORTS of channel read : bit

procedure forward (  
    input doFwd : bit;  
    input foundMask : 4 bits;  
    input posMask : 4 bits;  
    output fwfound : bit;  
    output readReg : bit;  
    output fdata : Datapath  
) is

    -- do the comparison and generate outputs
    doFwd ->! then
        if doFwd then
            foundMask, posMask ->! then
                case (#posMask @ #foundMask as 2*ROBSIZE bits) of
                0b0000_xxxx then -- nothing found
                    continue -- fdata <= 0
                | 0bxx1_0000, 0bxx1_1110,
                0bxx01_0001, 0bxx01_110x,
                0bxx01_01xx, 0bxx01_10xx,
                0b0001_0000, 0b0001_1111 then
                    fdata <= bdata[0]
                | 0bxx1x_001x, 0bxx1x_110x,
                0b001x_0000, 0b001x_1111,
                0b0010_0001, 0b0010_1110 then
                    fdata <= bdata[1]
                | 0b1xxx_0000, 0b1xxx_1111,
                0b1xx0_0001, 0b1xx0_1100,
                0b1xx0_001x, 0b1xx0_110x,
                0b1000_01xx, 0b1000_10xx then
                    fdata <= bdata[2]
                | 0b1xxx_0000, 0b1xxx_1111,
                0b1xx0_0001, 0b1xx0_1100,
                0b1xx0_001x, 0b1xx0_110x,
                0b1000_01xx, 0b1000_10xx then
                    fdata <= bdata[3]
                end -- case (#posMask @ #foundMask as 2*ROBSIZE bits)
                |
                readReg <= (foundMask /= 0)
                |
                fwfound <= (foundMask /= 0)
            end -- foundMask, posMask ->! then
        end --if doFwd
    end -- doFwd ->!
end -- procedure
-- Lookup unit, returns position (age) mask and found mask

```
procedure lookup(
    input read : bit;
    input addr : RegSpecExt;
    output posMask : 4 bits;
    output foundMask : 4 bits
) is
begin
    read ->!
    if read then
        addr ->!
        foundMask <- (maskArray { (bvaddr[0] = addr),
                                  (bvaddr[1] = addr),
                                  (bvaddr[2] = addr),
                                  (bvaddr[3] = addr) }
                          as 4 bits
                  ) ||
        posMask <- (maskArray { bpos[0], bpos[1], bpos[2], bpos[3] })
                          as 4 bits)
    end
end
```

-- Cell's allocation interface module

```
procedure allocCellT(
    parameter N : cardinal;
    input Tin : bit;
    input age1 : bit;
    input waddr : RegSpec;
    input invalid : bit;
    output Tout : bit;
    output allocT : bit
) is
begin
    Tin ->!
    if read then
        addr ->!
        foundMask <- (maskArray { (bvaddr[0] = addr),
                                  (bvaddr[1] = addr),
                                  (bvaddr[2] = addr),
                                  (bvaddr[3] = addr) }
                          as 4 bits
                  ) ||
        posMask <- (maskArray { bpos[0], bpos[1], bpos[2], bpos[3] })
                          as 4 bits)
    end
end
```

-- steers the allocation info to destination cell

```
procedure steerAlloc(
    input doAlloc : bit;
    input allocPtr1 : allocPtrType;
    input addrIn : RegSpec;
    input invalidIn : bit;
    array ROBSIZE of output age1 : bit;
    array ROBSIZE of output addr1 : RegSpec;
    array ROBSIZE of output invalid : bit
) is
begin
    addrIn, invalidIn, allocPtr1, doAlloc ->!
    if doAlloc then
        case (allocPtr1.index as WROBSIZE bits) of
            0b00 then
            age1[0] <- allocPtr1.cy ||
            addr1[0] <- addrIn ||
            invalid[0] <- invalidIn
```

```
nanoForwadUnit.balsa

```vhdl
267 |0b01 then
196 age[1] <= allocPtr1.cycl
197 sAddr[1] <= addrIn ||
198 invalid[1] <= invalidIn
199 |0b10 then
200 age[2] <= allocPtr1.cycl
201 sAddr[2] <= addrIn ||
202 invalid[2] <= invalidIn
203 |0b11 then
204 age[3] <= allocPtr1.cycl
205 sAddr[3] <= addrIn ||
206 invalid[3] <= invalidIn
207 end
208 end -- if doAlloc
209 end -->!;
210 end --procedure steerAlloc
211
212 -- steers the arrival information to cells
213 procedure steerArrival(
214 input doAlloc : bit;
215 input arrPtr1 : WROBSIZE bits;
216 input data : Datapath;
217 array ROBSIZE of output wdata : Datapath
218 ) is
219 begin
220 loop
221 doAlloc, arrPtr1 -->! then
222 if doAlloc then
223 case arrPtr1 of
224 0b00 then
225 data --> wdata[0]
226 |0b01 then
227 data --> wdata[1]
228 |0b10 then
229 data --> wdata[2]
230 |0b11 then
231 data --> wdata[3]
232 end
233 end -- if doAlloc
234 end -->!;
235 end --loop
236 end --procedure steerArrival
237
238 -- arrival interface to cells
239 procedure arrCellNAA2(
240 parameter N : cardinal;
241 input allocT : bit;
242 input Tin : bit;
243 input data : Datapath;
244 output Tout : bit;
245 output wdata : Datapath;
246 output waddr : RegSpec
247 ) is
248 constant INVALID_BIT = sizeof RegSpec
249 begin
250 loop
251 allocT -->! then
252 continue
253 end ||
254 data -->! then
255 bdata[N] := data
256 end;
257 Tin --> then -- no active eager (-->t) allowed here !!!
258 if (not (#(bvaddr[N])#INVALID_BIT) as bit)) then
259 wdata <= bdata[N] ||
```
nanoForwardUnit.balsa

```vhd
260   waddr <= (#(bvaddr[M])|[0..INVALID_BIT-1] as RegSpec)
261   end
262   end;
263   Tout <= 1
264   end
265 end
266
267 -- allocation/arrival initiator
268 procedure arrCellH(
269   input Tin : bit;
270   output Tout : bit
271 ) is
272 begin
273   Tout <= 1 ;
274   loop
275     Tin -> Tout
276   end
277 end
278
279 begin --nanoForwardUnit
280 -- initialise buffer
281   for ||! i in 0..ROBSIZE-1 then
282     bpos[i] := 0b0
283     ||
284     bvaddr[i] := 0b1_0000_0000 -- invalidate all entries
285   end
286
287 -- Lookup i/f
288   for ||! i in 0..READPORTS-1 then
289     loop
290       raddr[i] ->! then
291       addr[i] <= (#(raddr[i])) @ #0b0 as RegSpecExt) -- invalid = 1
292     end
293     end ||
294     loop
295       readIn[i] ->! then
296       read[i] <= readIn[i] ||
297       doFwd[i] <= readIn[i]
298     end
299   end ||
300   -- (Lookup ; Allocate) group
301   loop
302     for ||! i in 0..READPORTS-1 then
303       lookup( read[i],
304         addr[i],
305         posMask[i],
306         foundMask[i]
307       )
308     end ;-- This `;' MUST be substituted by unfolded `;'
309     -- Allocate
310     steerAlloc( doAlloc[0],
311       allocPtr1,
312       wraddr[0],
313       invalidIn[0],
314       ageM,
315       sAddrM,
316       invalidM
317     )
318     ||!
319     steerAlloc( doAlloc[1],
320       allocPtr2,
321       wraddr[1],
322       invalidIn[1],
323       ageM,
324       sAddrM,
```
nanoForwardUnit.balsa

```c
invalidM
}
for[i in 0..ROBSIZE-1 then
    allocCell(i,
            allocToken[i],
            ageM[i],
            aAddrM[i],
            --wbaddr,
            invalidM[i],
            allocToken[i+1],
            allocT[i])
end
end
--+ (Forward; Arrival) Group
loop
for[i in 0..READPORTS-1 then
    forward(
            doFwd[i],
            foundMask[i],
            posMask[i],
            fwfound[i],
            readReg[i],
            fwdata[i])
end
--+ arrival
-- Steer arrival requests
steerArrival(doArrival[0],
             arrPtr1,
             wrdata0,
             wbdataM)
steerArrival(doArrival[1],
             arrPtr2,
             wrdata1,
             wbdataM)
end
--+ arrival cells
for[i in 0..ROBSIZE-1 then
    allocCellNAA2(i,
                 allocT[i],
                 wbToken[i],
                 wbdataM[i],
                 wbToken[i+1],
                 wbdata, wbaddr)
end
--+ arrive cell initiators
arrCellH(wbToken[ROBSIZE], wbToken[0])
arrCellH(allocToken[ROBSIZE], allocToken[0])
end
```