http://researchspace.auckland.ac.nz

ResearchSpace@Auckland

Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.

ACCELERATION OF ODE-BASED BIOMEDICAL SIMULATIONS WITH RECONFIGURABLE HARDWARE

TING YU

Supervised by Dr Oliver Sinnen and Dr Chris Bradley

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Electrical and Electronic Engineering, the University of Auckland, 2015.

Ting Yu: *Acceleration of ODE-Based Biomedical Simulations with Reconfigurable Hardware* © 2015

ABSTRACT

Biomedical models and simulations often require high performance computing environments. For example, simulating one minute of electrical activity of a human heart may require more than one month of computation time with today's fastest processor. Biomedical models often are based on ordinary differential equations (ODEs) which require numerical integration during the simulation. The numerical integration is regular and easy to parallelise. Parallel systems that consist of a large number of general purpose processors (GPPs) and graphics processing units (GPUs) as accelerators have been traditionally used for these types of simulations. However, such systems usually involve high financial cost and energy consumption. Given the inherent parallelism and high computational requirements, FPGAs (Field Programmable Gate Arrays) with their high parallel architecture and flexibility, are promising for accelerating these kind of computations, whilst being power efficient.

FPGAs are highly configurable devices with logic blocks and interconnects. The logic blocks are programmable and can incorporate parallelism into arbitrary digital circuits such as being arranged into pipelines or replicated for task and data parallelism. However, FPGAs are not widely adopted by biomedical scientists due to their lack of hardware expertise. Furthermore, FPGAs have a limited usable area and so design tool chains can create problems when implementing large sized biomedical models.

To overcome these obstacles and to exploit the potential of FPGAs, this thesis investigates the automatic generation of digital hardware for the domain of

biomedical models that can be described as ODEs. The hardware accelerator is based on a pipelined architecture with a hardware/software co-design system. ODoST, an ODE-based domain-specific sythesis tool, is proposed. The tool is capable of automatically generating a FPGA-based hardware accelerator module (HAM) from a high-level description of a mathematical model. This tool will be of benefit to biomedical scientists and engineers without hardware design expertise. In addition, a list of optimisation strategies are investigated and implemented in order to maximise the use of a target FPGA device with limited resources.

The experimental evaluation on real hardware shows that FPGAs deliver a much higher power efficiency than CPU and GPU implementations. Furthermore, FPGA implementations have a significant performance advantage compared to multicore implementations and a comparable processing speed to GPU implementations.

ACKNOWLEDGMENTS

It would not have been possible to complete this thesis without the help and support from a number of people.

I would like to sincerely express my gratitude to my supervisors Dr Oliver Sinnen and Dr Chris Bradley for their guidance and inspiration over the past couple of years. Without their constant supervision, encouragement and great support throughout the years, I would not have been where I am now.

I am grateful to all members of the Parallel and Reconfigurable Computing group (PARC) for their generosity in sharing knowledge and experience in work and life.

I would like to acknowledge the financial support I have received during the work. This work has been supported by the Tertiary Education Commission (TEC) and the Auckland Bioengineering Institute (ABI) under the Bright Future Enterprise Scholarship and University of Auckland under a University of Auckland Doctoral Scholarship.

Finally, I must thank my friends for their help and support, especially Wendy for proof-reading my thesis chapters and offering grammatical assistance. Most importantly, I should thank my beloved husband, Yang, and my parents for their understanding and believing in me, for helping me get through the difficult times, and for all the emotional support and caring they provided.

CONTENTS

1 INTRODUCTION 1
1.1 Biomedical Modelling and Simulation 1
1.1.1 Biomedical Modelling with CellML 2
1.1.2 Biomedical Simulation with OpenCMISS 9
1.2 Hardware Acceleration with Reconfigurable Hardware 11
1.2.1 Hybrid Acceleration System 12
1.2.2 Field Programmable Gate Arrays 12
1.2.3 PCI Express 17
1.2.4 Floating Point Unit 19
1.3 High-level Synthesis 24
1.3.1 Benefit 24
1.3.2 Design Processes 25
1.4 Thesis Motivation and Contributions 26
1.4.1 Motivations 26
1.4.2 Contributions 28
1.5 Thesis Structure 29
2 HARDWARE ACCELERATOR MODULE 31
2.1 Introduction 32
2.2 Related Work 34
2.3 CellML Hardware Model 35
2.3.1 A Motivating Example 35
2.3.2 Model Overview 35
2.3.3 Pipelined Floating Point Operations 38

2.3.4 The Hardware Model Architecture 41
2.4 System Design and Implementation 42
2.4.1 Overall System Architecture 42
2.4.2 Host Computer Design 43
2.4.3 FPGA Design 43
2.5 Experiments 46
2.5.1 Experimental Setup 46
2.5.2 Synthesis Results 47
2.5.3 Performance comparison 49
2.5.4 Discussion 49
2.6 Conclusions 51
3 ODE-BASED DOMAIN-SPECIFIC SYNTHESIS TOOL 53
3.1 Introduction 54
3.2 Related Work 56
3.3 Biomedical Hardware Accelerator Module 58
3.3.1 A Motivating Example 58
3.3.2 Biomedical Model Overview 58
3.3.3 Pipelined Floating Point Operations 62
3.3.4 Hardware Accelerator Module Architecture 65
3.4 ODE-based High-level Synthesis 70
3.4.1 ODoST Overview 71
3.4.2 Input Model Format 73
3.4.3 Analysis Phase 74
3.4.4 Generation Phase 79
3.4.5 System Integration 87
3.5 Evaluation 88
3.5.1 Models 89
3.5.2 Experimental Setup 90
3.5.3 Synthesis Results 92
3.5.4 Performance Results 95
3.5.5 Power Efficiency 100
3.6 Conclusions 102

4 PERFORMANCE OPTIMISATION AND RESOURCE UTILISATION 103
4.1 Introduction 104
4.2 Related Work 106
4.3 HAM and ODoST 108
4.3.1 Biomedical Model Overview 108
4.3.2 Hardware Accelerator Module 109
4.3.3 ODE-based Domain Specific Synthesis Tool 110
4.4 Compiler Optimisation 111
4.4.1 Local Optimisations 112
4.4.2 Common Subexpression Elimination 114
4.4.3 Higher-order powers 114
4.4.4 Exponential Function Simplification 115
4.4.5 Source-to-source Optimiser 116
4.5 Resource Fitting and Balancing 117
4.5.1 FPGA Resource Capacity 118
4.5.2 Floating Point Cores 119
4.5.3 Resource Allocation Techniques 121
4.6 Multiple Pipelines 127
4.6.1 Single Pipeline 128
4.6.2 Extended Pipeline 129
4.6.3 Parallel Pipelines 130
4.6.4 Implementation 131
4.7 Evaluation 132
4.7.1 Experimental Setup 132
4.7.2 Synthesis Results 135
4.7.3 Performance Results 139
4.7.4 Power Efficiency 141
4.8 Conclusions 145
5 CONCLUSIONS 147
A EXAMPLE CELLML MODELS 151
A.1 Hodgkin-Huxley Model 151
A.1.1 Mathematics 151

A.1.2 C-code Representation 153
A.2 Beeler-Reuter Model 155
A.2.1 Mathematics 155
A.2.2 C-code Representation 157
A.3 Hilemann-Noble Model 161
A.3.1 Mathematics 161
A.3.2 C-code Representation 166
A.4 TNNP Model 175
A.4.1 Mathematics 175
A.4.2 C-code Representation 183

BIBLIOGRAPHY 195

LIST OF FIGURES

Figure 1.1 CellML model structure 3
Figure 1.2 A schematic cell diagram describing the Hodgkin-Huxley model 5
Figure 1.3 A schematic diagram describing the Beeler-Reuter model 6
Figure 1.4 A schematic diagram describing the Hilemann-Noble model 7
Figure 1.5 A schematic diagram describing the model of human ventricular myocyte 9
Figure 1.6 Typical hybrid hardware acceleration system. 12
Figure 1.7 Stratix IV FPGA architecture 14
Figure 1.8 Stratix IV FPGA ALM 14
Figure 1.9 Altera's FPGA application design flow 16
Figure 1.10 PCI express layered architecture 17
Figure 1.11 IP compiler for PCI express with Avalon-MM interface 20
Figure 1.12 IEEE-754 floating point format 21
Figure 2.1 Abstract view of model interaction 38
Figure 2.2 Pipeline scheduling for *sodium_channel_m_gate* component 40
Figure 2.3 CellML hardware model core structure 41
Figure 2.4 A block diagram of the overall system architecture 42
Figure 2.5 Flow of *CellMLWrapper* 44
Figure 2.6 State machines for read and write controllers 45

Figure 2.7 Performance results of CellML hardware model computation 50
Figure 3.1 General flow of model computation 61
Figure 3.2 Pipeline scheduling for *sodium_channel_m_gate* integration 64
Figure 3.3 Hardware accelerator module system architecture 66
Figure 3.4 Flow of software module 67
Figure 3.5 State machine for hardware data control 69
Figure 3.6 Hardware accelerator structure 71
Figure 3.7 Overview of ODoST 72
Figure 3.8 Design flow of ODoST 72
Figure 3.9 C representation of model 73
Figure 3.10 Generation structure 80
Figure 3.11 Hardware accelerator nested framework 81
Figure 3.12 Templated entity declaration 82
Figure 3.13 Internal signals declaration 82
Figure 3.14 Signal shifting 83
Figure 3.15 Operations initiation and mapping 83
Figure 3.16 Equations initiation and mapping 84
Figure 3.17 On-chip memory allocation 86
Figure 3.18 Synthesis resource usage results of the generated HAMs 93
Figure 3.19 Synthesis performance results of the generated HAMs with their pipeline latencies 94
Figure 3.20 Average execution time per iCell of the HAMs over number of cells and micro time steps 96
Figure 3.21 Processing speed of the generated HAMs compare to the CPU implementations 98
Figure 3.22 Processing speed of the HAM for the Beeler-Reuter model compared to the GPU implementations 98
Figure 3.23 Power consumption of the generated HAMs compared to the CPU and GPU implementations 101
Figure 4.1 Hardware accelerator module system architecture 110

Figure 4.2 Exemplary transformations done by LLVM 113
Figure 4.3 Single pipeline flow 128
Figure 4.4 Extended pipeline flow 129
Figure 4.5 Parallel pipeline flow 131
Figure 4.6 Synthesis resource usage results of the HAMs for the Beeler-Reuter model 136
Figure 4.7 Synthesis resource usage results of the non-optimised and optimised HAMs for the TNNP model 138
Figure 4.8 Processing speed of the HAMs compared to the CPU and GPU implementations for the Beeler-Reuter model 140
Figure 4.9 Processing speed of the HAMs compare to the CPU implementations for the TNNP model 142
Figure 4.10 Power consumption of the HAM, CPU and GPU implementations for the Beeler-Reuter model 143
Figure 4.11 Power consumption of the HAM and CPU implementations for the TNNP model 145

LIST OF TABLES

Table 1.1 CellML model metrics 8

Table 1.2 IEEE-754 special case numbers 21

Table 1.3 IEEE-754 single and double precision formats 22

Table 2.1 Number of equations and floating point operations in the Hodgkin-Huxley model components 36

Table 2.2 Synthesis results of the floating point operations for Altera EP4SGX230 device 48

Table 2.3 Synthesis results of the Hodgkin-Huxley CellML hardware model 48

Table 2.4 Synthesis results of the complete hardware system for Altera EP4SGX230 device 49

Table 3.1 FloPoCo resource use and performance for Stratix IV device 62

Table 3.2 Metrics of the considered biomedical models 90

Table 3.3 Stratix IV EP4SGX530KH40C2 device specifications 90

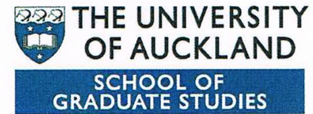
Table 3.4 Power requirements for the three testing platforms 101

Table 4.1 Resource capability for selected devices 118

Table 4.2 Altera single precision Floating Point Megafunctions resource usage and frequency estimation for Stratix IV Devices 120

Table 4.3 Resource usage and frequency estimation of FloPoCo generated single precision floating point cores for Stratix IV Devices 121

Table 4.4 Resources percentage usage of the three variations of floating point multiplication 122
Table 4.5 Schemes for reducing PT used in the greedy algorithm 125
Table 4.6 Evaluation results for the resource balancing example for a different numbers of multipliers 127
Table 4.7 Operations and I/O of Beeler-Reuter models show increasing linearly with the number of pipelines 133
Table 4.8 Operations and I/O of an optimised TNNP model against the original model 134
Table 4.9 Estimated resource consumption of TNNP HAM before and after resource allocation optimisation 137
Table 4.10 Predicted clock frequencies for the HAMs of the Beeler-Reuter model 138
Table 4.11 Power requirement for the Beeler-Reuter model on the three testing platforms 143
Table 4.12 Power requirement for the TNNP model on the two testing platforms 144



Co-Authorship Form

Graduate Centre
ClockTower – East Wing
22 Princes Street, Auckland
Phone: +64 9 373 7599 ext 81321
Fax: +64 9 373 7610
Email: postgraduate@auckland.ac.nz
www.postgrad.auckland.ac.nz

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 2 of this thesis is published as a conference paper in International Conference on Field Programmable Technology, FPT'13 as:

Hardware acceleration of biomedical models with OpenCMISS and CellML, DOI: 10.1109/FPT.2013.6718390

Nature of contribution by PhD candidate	The concepts, ideas and strategies of hardware acceleration of biomedical models, design, development and evaluation of the hardware accelerator and manuscript writing.
Extent of contribution by PhD candidate (%)	>70%

CO-AUTHORS

Name	Nature of Contribution
Chris Bradley	Academic advisor on biomedical acceleration, co-supervision of the research and manuscript revision.
Oliver Sinnen	The concepts, ideas and strategies of hardware acceleration, supervision of the research and manuscript revision.

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

Name	Signature	Date
Chris Bradley	[signature]	31/03/2015
Oliver Sinnen	[signature]	31/03/2015
		Click here
		Click here
		Click here
		Click here

Last updated: 25 March 2013

Co-Authorship Form

Graduate Centre
ClockTower – East Wing
22 Princes Street, Auckland
Phone: +64 9 373 7599 ext 81321
Fax: +64 9 373 7610
Email: postgraduate@auckland.ac.nz
www.postgrad.auckland.ac.nz

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 3 of this thesis is submitted for publication as a research article in ACM Transactions on Reconfigurable Technology and Systems (currently under review) as:

ODoST: Automatic Hardware Acceleration for Biomedical Model Integration

Nature of contribution by PhD candidate	The concepts, ideas, algorithms and strategies of hardware acceleration of biomedical models and domain-specific high-level synthesis, design and development of ODoST, evaluation of the ODoST generated hardware accelerator modules and manuscript writing.
Extent of contribution by PhD candidate (%)	>70%

CO-AUTHORS

Name	Nature of Contribution
Chris Bradley	Academic advisor on biomedical acceleration, co-supervision of the research and manuscript revision.
Oliver Sinnen	The concepts, ideas and strategies of hardware acceleration and domain-specific high-level synthesis, supervision of the research and manuscript revision.

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

Name	Signature	Date
Chris Bradley	[signature]	31/03/2015
Oliver Sinnen	[signature]	31/03/2015
		Click here
		Click here
		Click here

Last updated: 25 March 2013

Co-Authorship Form

Graduate Centre
ClockTower – East Wing
22 Princes Street, Auckland
Phone: +64 9 373 7599 ext 81321
Fax: +64 9 373 7610
Email: postgraduate@auckland.ac.nz
www.postgrad.auckland.ac.nz

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 4 of this thesis is submitted for publication as a research article in Concurrency and Computation: Practice and Experience (currently under review) as:

Performance Optimisation Strategies for Automatically Generated FPGA Accelerators for Biomedical Models

Nature of contribution by PhD candidate	The concepts, ideas and strategies of hardware acceelration of biomedical models and hardware optimisations, design, development and evaluation of the optimisation strategies and manuscript writing.
Extent of contribution by PhD candidate (%)	>70%

CO-AUTHORS

Name	Nature of Contribution
Julian Oppermann	Implementation of the source-to-source optimiser.
Chris Bradley	Academic advisor on biomedical acceleration, co-supervision of the research and manuscript revision.
Oliver Sinnen	The concepts, ideas and strategies of hardware acceleration and optimisation, supervision of the research and manuscript revision.

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

Name	Signature	Date
Julian Oppermann	[signature]	29/03/2015
Chris Bradley	[signature]	31/03/2015
Oliver Sinnen	[signature]	31/03/2015
		Click here
		Click here

1 INTRODUCTION

The traditional approach in high performance computing (HPC) is to build parallel systems that consist of a large number of general purpose processors (GPPs). However, such systems usually involve high financial cost and energy consumption. Systems with a small number of processors can normally achieve a near-linear speedup. However for systems with a large number of processors, the speedup can flatten out into a constant value [48]. Power and cooling demands can also restrict the number of processors that are affordable [18, 44]. These limitations push HPC engineers to look for other computing technologies such as dedicated computation hardware acceleration for special application areas like bioengineering and scientific computing. A more flexible approach is to use reconfigurable hardware based on Field Programmable Gate Arrays (FPGAs), which can improve performance and reduce power consumption in HPC applications. FPGAs are highly configurable devices with logic blocks and interconnects. The logic blocks are programmable and can incorporate parallelism into arbitrary digital circuits such as being arranged into pipelines or replicated for task and data parallelism.

1.1 BIOMEDICAL MODELLING AND SIMULATION

Biomedical models involve sets of mathematical equations that describe a biomedical system of interest. Biomedical simulations often use numerical com-

putations of these equations to simulate dynamic systems and helping researchers understand different physiological functions. Due to the increased complexity of models and accuracy requirements, the number of variables or Degrees-Of-Freedom (DOF) used for modern biomedical models has rapidly increased in recent times. Complex models with fine mesh size and short time steps require a significant amount of computation, which can result in very long run times even with today's fastest CPUs [94]. However, such models often contain a small and fixed portion of code that executes a large number of times using different data. These code portions are ideally suited for hardware acceleration with FPGAs. In this thesis, CellML is used to describe biomedical models and develop hardware acceleration modules (HAMs) based on FPGAs for these models. These HAMs are to be used with the biomedical modelling environment, OpenCMISS [24], in order to simulate multi-scale physiological systems.

1.1.1 *Biomedical Modelling with CellML*

CellML [34] is an XML based model description language for specifying and exchanging biophysically based systems of Ordinary Differential Equations (ODEs) and Differential Algebraic Equations (DAEs). It takes advantage of the extensibility of the XML language and incorporates other XML-based standards, including MathML [17], XLink [41], and Resource Description Framework (RDF) [25].

1.1.1.1 *CellML Model Structure*

CellML contains its own defined elements for describing the model structure. Other information is incorporated into the model document using existing standards. For example, MathML is used to encode the mathematics of the model, XLink is used to establish the connection between the original model and the importing model, and background information, or metadata, is included via RDF [34].

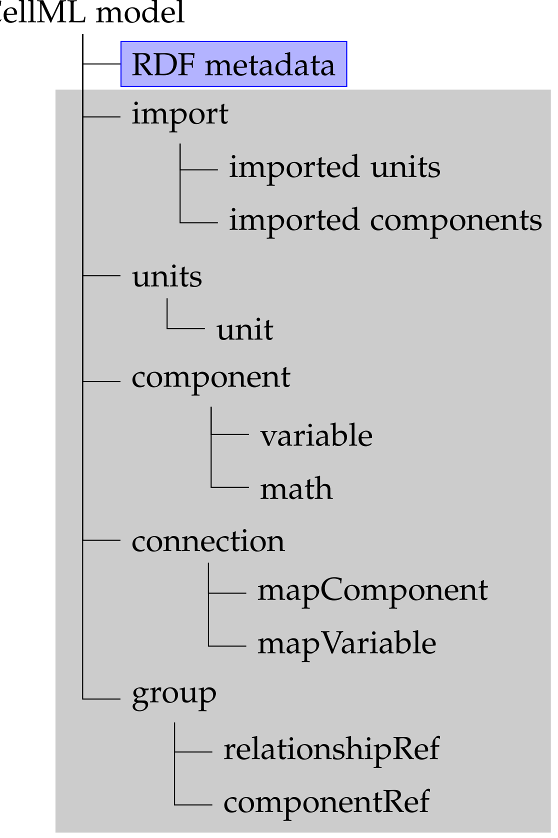


Figure 1.1: CellML model structure.

The structure of a CellML model is illustrated in Figure 1.1. A CellML model is represented by a set of interconnected components. A component is the functional unit of a CellML model that contains variables and mathematical equations. A variable is associated with a unit that is defined in the units entity. The mathematical equations are expressed using MathML that is embedded within the CellML framework. Biochemical reactions between substrates are organized into components that represent the reactants and products of the reactions, the reactions themselves, and the enzymes or inhibitors that influence the reaction rates. The properties of a reaction—such as its reactants, products, enzymes, and inhibitors—and the reaction kinetics are all captured by the variables and the mathematical equations of a component [34]. Connections are used to link two components by mapping the variables inside one component with variables inside the other component. Grouping adds structure to a model by defining named relationships between components. Importing provides authors with the ability to reuse parts of other models by importing components

or units from other models. RDF metadata is included in CellML to provide structured descriptive information such as the model author, literature reference, copyright, etc., and to facilitate searches of collections of models and model components from the CellML model repository [64].

1.1.1.2 *Mathematical Representation*

Mathematically, a CellML model describes a vector system, $\mathbf{F}$, of DAEs in the form of:

$$\mathbf{F}(t, \mathbf{x}, \mathbf{x}', \mathbf{a}, \mathbf{b}) = \mathbf{0} \tag{1.1}$$

where t is the independent variable, $\mathbf{x}$ is a vector of state variables, $\mathbf{x}'$ is a vector of the derivatives of state variables with respect to the independent variable, $\mathbf{a}$ is a vector of independent parameters/constants, and $\mathbf{b}$ is an optional vector of intermediate/algebraic "output" variables from the model. All the variables are defined in the *variable* entity under each component.

1.1.1.3 *Example CellML Models*

Four CellML model examples are described here. The four models are selected from the CellML model repository[1] with each model having a different level of complexity. The mathematics and C-code representation for each example model are shown in Appendix A. Of the four example models, the first two simple models, the Hodgkin-Huxley model and the Beeler-Reuter model are used as the case studies for model investigation and hardware design. These two models together with two more complex models, the Hilemann-Noble and the TNNP model are also used as the test cases for the evaluation of the research work throughout the thesis.

HODGKIN-HUXLEY MODEL The Hodgkin-Huxley Model was developed by Hodgkin and Huxley [53] in 1952. The model describes the flow of electric current through the surface membrane of the giant nerve axon of a squid.

1 http://www.cellml.org/model

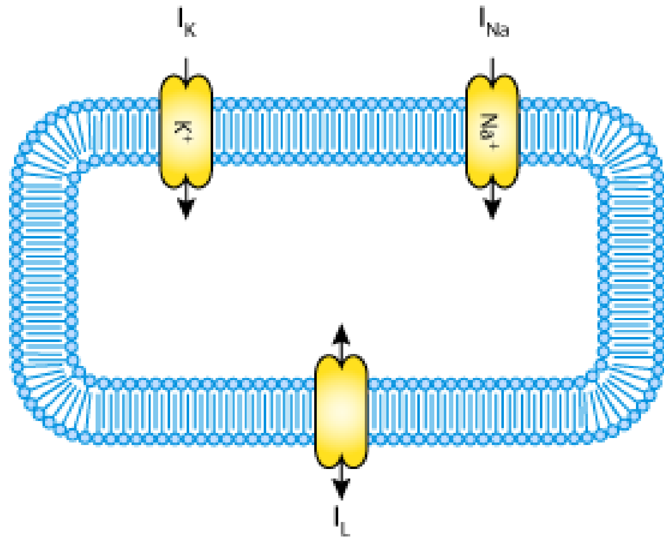


Figure 1.2: A schematic cell diagram describing the current flows across the cell membrane that are captured in the Hodgkin Huxley model [52].

The schematic diagram of the model is shown in Figure 1.2. The model describes the flow of ions across a cell membrane (the ionic current). The ionic current is divided into components carried by sodium and potassium ions (I_{Na} and I_K), and a small 'leakage current' (I_L) carried by chloride and other ions. Each component of the ionic current is determined by the transmembrane potential (a driving force which may conveniently be measured as an electrical potential difference between the inside and outside of the cell) and a permeability coefficient which has the dimension of conductance. Thus the sodium current (I_{Na}) is equal to the sodium conductance (g_{Na}) multiplied by the difference between the membrane potential (V) and the equilibrium potential for the sodium ion (E_{Na}). Similar equations apply to I_K and I_L. This model has been used as the basis for almost all other ionic current models of excitable tissues, including cardiac atrial and ventricular muscle. The Hodgkin-Huxley model is the simplest of the four models.

BEELER-REUTER MODEL The Beeler-Reuter Model was developed by Beeler and Reuter [21] in 1977. The model describes the membrane action potentials of mammalian ventricular myocardial fibres. The total ionic flux is divided into four discrete, individual ionic currents as shown in Figure 1.3. The main

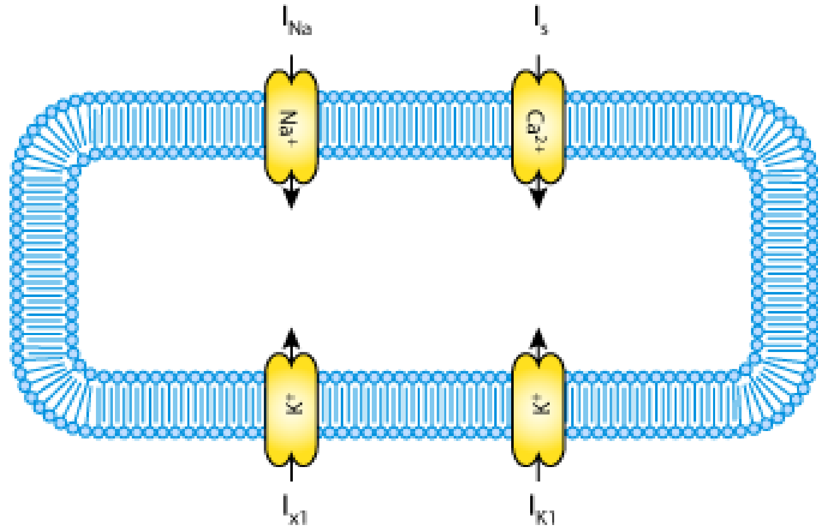


Figure 1.3: A schematic diagram describing the current flows across the cell membrane that are captured in the Beeler-Reuter model [20].

additional feature of the Beeler-Reuter ionic current model compared to the Hodgkin-Huxley model is its inclusion of a representation of the intracellular calcium ion concentration. The model incorporates two voltage-dependent and time-dependent inward currents: the excitatory inward sodium current, I_{Na}, and a secondary, or slow inward, current, I_s, which is primarily carried by calcium ions. A time-independent outward potassium current, I_{K1}, exhibiting inward-going rectification, and a voltage-dependent and time-dependent outward current, I_{x1}, primarily carried by potassium ions, are further elements of the model.

HILEMANN-NOBLE MODEL The Hilemann-Noble Model was developed by Hilemann and Noble [51] in 1987. The model describes the interactions of electrogenic sodium-calcium exchange, calcium channel and sarcoplasmic reticulum in the mammalian heart which occur when the extracellular calcium transients are stimulated with tetramethylmurexide in the rabbit atrium. The schematic diagram of the model is shown in Figure 1.4.

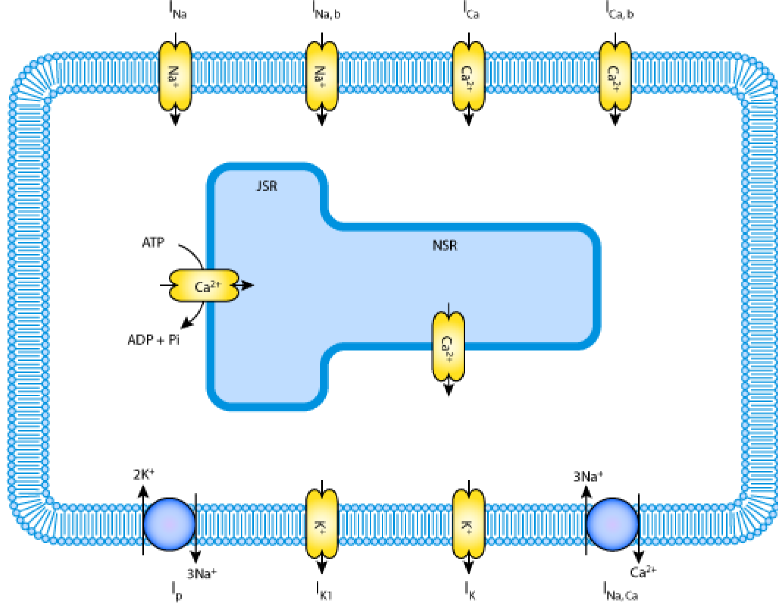


Figure 1.4: A schematic diagram describing the current flows across the cell membrane that are captured in the Hilemann-Noble model [50].

TUSSCHER-NOBLE-NOBLE-PANFILOV MODEL The Tusscher-Noble-Noble-Panfilov (TNNP) model for human ventricular tissue was developed by Ten Tusscher et al. [96]. This model describes the action potential of human ventricular cells including a high level of electrophysiological detail, and can be applied in large-scale spatial simulations for the study of reentrant arrhythmias. The model is based on the experimental data on most of the major ionic currents: the fast sodium, L-type calcium, transient outward, rapid and slow delayed rectifier, and inward rectifier currents, and it also includes a basic calcium dynamics, allowing for the realistic modeling of calcium transients, calcium current inactivation, and the contraction staircase. A schematic diagram of the model is shown in Figure 1.5.

MODEL METRICS The model metrics with the number of components, equations, parameters/variables and operations for the four example models from the CellML model repository are presented in Table 1.1. These metrics are used in model analysis and evaluation in later chapters.

Model	Hodgkin-Huxley	Beeler-Reuter	Hilemann-Noble	TNNP
Components	8	13	23	30
Equations	14	30	45	84
State variables	4	8	15	17
Parameters/rate constants	8	12	55	46
Algebraic variables	10	18	40	67
Operation: $+$	9	41	47	112
Operation: $-$	11	34	72	64
Operation: $\times$	17	52	134	139
Operation: $\div$	10	28	52	129
Operation: e^x	6	25	21	52
Operation: x^y	2	1	7	26
Operation: $\log(x)$	-	1	4	4

Table 1.1: CellML model metrics.

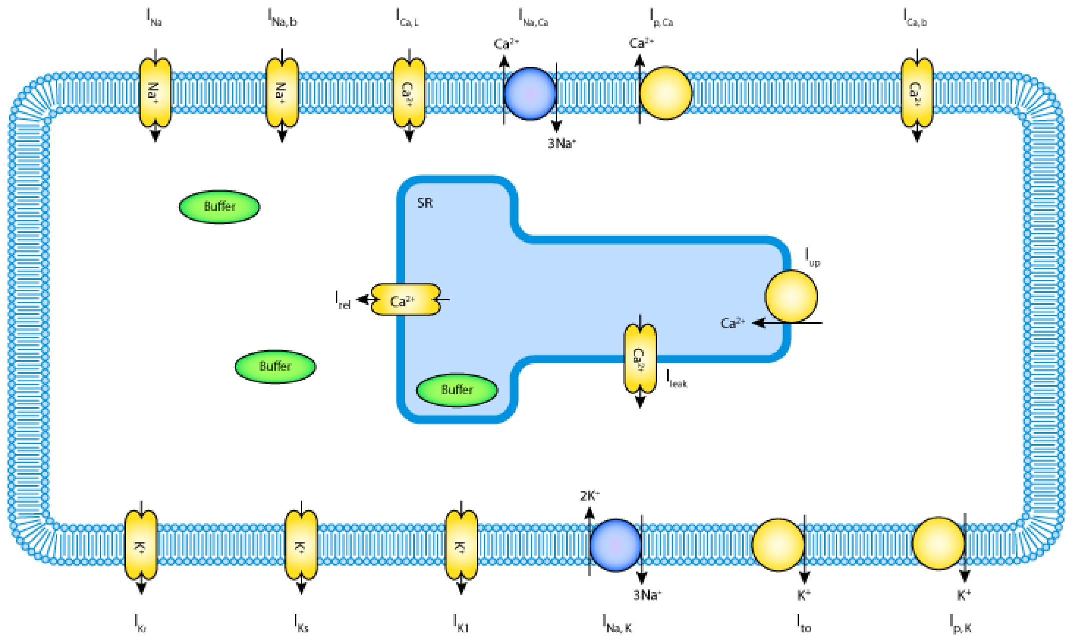


Figure 1.5: A schematic diagram describing the ion movement across the cell surface membrane and the sarcoplasmic reticulum, which are described by the Ten Tusscher et al. 2004 mathematical model of the human ventricular myocyte [95].

1.1.1.4 *CellML API*

For CellML models to be useful, tools which can process them correctly are needed. Therefore, an Application Programming Interface (API), and a good implementation of that API, are required for supporting CellML. The developed CellML API [67] allows for the information in CellML models to be retrieved and/or modified. It also contains a series of optional API extension, for tasks such as simplifying the handling of connections between variables, dealing with physical units, validating models, and translating models into different procedural languages e.g., the C language.

1.1.2 *Biomedical Simulation with OpenCMISS*

OpenCMISS [24] is a general modelling environment with particular features for biomedical simulations. It consists of two main parts: a graphical and field manipulation library, OpenCMISS-Zinc, and a parallel computational library

for solving partial differential and other equations using a variety of numerical methods, OpenCMISS-Iron. OpenCMISS-Iron is a re-engineering of the CMISS (Continuum Mechanics, Image analysis, Signal processing, and System identification) computational code that has been developed and used for over 30 years.

1.1.2.1 *OpenCMISS Fields*

In OpenCMISS fields are the central mechanism that describe and store information of physical problems. OpenCMISS fields are in hierarchical structure, with each field containing a set of field variables and each field variable containing a set of field variable components. A field is defined over a domain which is, conceptually, an entire computational mesh representing the model of interest. However, when executing in parallel, the mesh is decomposed into a number of computational domains depending on the number of computational nodes. OpenCMISS allows each field variable component to have different forms of DOFs structures including:

- constant structure (one DOF for the component);
- element structure (one or more DOFs for each element);
- node structure (one or more DOFs for each node);
- Gauss point structure (one or more DOFs for each Gauss or integration point);
- data point structure (one or more DOFs for each data point).

OpenCMISS collects all the DOFs from all the field variable components and stores them as a single distributed vector. The DOFs stored in the distributed vector include those from the computational domain and also a layer of "ghosted" DOFs (local copies of the value of DOFs in a neighbouring domain). To ensure consistency of data OpenCMISS handles the updates between computational nodes if a computational node changes the value of a DOF, which is ghosted on a neighbouring computational node [24].

1.1.2.2 *Use of CellML Models in OpenCMISS*

In biomedical simulations using OpenCMISS, CellML allows for the "plug and play" of mathematical models and model configurations. OpenCMISS uses the CellML API [67] to interact with CellML models. In OpenCMISS-Iron, a higher level CellML interface is defined with the use of the CellML API, and this interface is used by the OpenCMISS core library [71].

Since models in OpenCMISS are defined using a collection of fields, CellML models are integrated into OpenCMISS through these fields. The CellML variables are mapped with OpenCMISS models field variable components. Depending on the direction of dataflow, there are two types of maps. A "known" CellML variable represents a map link from OpenCMISS to CellML (input variable to the CellML model) and a "wanted" CellML variable represents a map link from CellML to OpenCMISS (output variable from the CellML model). A map is specified by identifying a particular OpenCMISS field variable component and the name of a CellML variable in the CellML model. OpenCMISS looks at each DOF in each field variable component that has been mapped and determines the DOF location (i.e., the position of the node) for each instances of a CellML model [71].

1.2 HARDWARE ACCELERATION WITH RECONFIGURABLE HARDWARE

Hardware acceleration is the use of computer hardware to perform particular functions faster than if they are executed on a more general-purpose CPU. Normally, processors execute instructions one by one in sequence. The performance of sequential processors can be improved by various techniques and including hardware acceleration. Programming at the hardware level enables optimal parallel processing by removing the architectural constraints of a traditional CPU and its operating system layers [87].

Hardware accelerators are designed for computationally intensive software code, such as those with repetitive mathematical calculations e.g., integrations.

Examples of devices that are commonly used as hardware accelerators are Graphics Processing Units (GPUs), FPGAs and Application Specific Integrated Circuits (ASICs). Compared to GPPs, there is a trade-off between flexibility and efficiency, with hardware accelerators. Implementing an application in hardware increases efficiency but decreases flexibility.

1.2.1 *Hybrid Acceleration System*

Hardware accelerators like FPGAs yield fast performance. However, large applications implemented on a GPP may be more area efficient and require less designer's effort, albeit at the expense of slower performance. A hybrid hardware acceleration system (or hardware/software co-design system) is a system combining a GPP and one or more custom coprocessors through an interconnect. The system enables the critical computational region of a given application to be put into a coprocessor and to keep the rest in the GPP to achieve an implementation that best satisfies requirements of performance, area and designer effort. Figure 1.6 illustrates a typical hybrid hardware acceleration system.

Figure 1.6: Typical hybrid hardware acceleration system.

1.2.2 *Field Programmable Gate Arrays*

Computation in the computer and electronic world is usually performed in two ways: via hardware and via software. Computer hardware, like ASICs, provides high performance for critical tasks but it is permanently configured to the specified application. On the other hand, computer software provides flexibility for performing different tasks/applications, but is orders of mag-

nitude worse than ASIC implementations in terms of performance and power usage. FPGAs fill the gap between the two and blend the benefits of computation via both hardware and software. FPGAs implement circuits just like ASICs, providing huge power and performance benefits over software, and yet can be reprogrammed in order to implement a wide range of tasks.

Like other computational hardware, FPGAs are essentially integrated circuits that are able to implement computations spatially and simultaneously so that millions of operations can be executed by resources distributed on a silicon chip. Furthermore, since the dynamic power consumed by a FPGA depends on clock frequency, the overall power consumption is lower than either a CPU or GPU as the FPGA's clock frequency is typically hundreds of MHz compared to a CPU's or GPU's GHz.

1.2.2.1 *Architecture*

In general, a FPGA contains a matrix of logic blocks, with interconnects between these logic blocks. The logic blocks may be named differently depending on vendors. Altera[2] and Xilinx[3] are the two dominating manufacturers of FPGAs. In Altera FPGAs, these logic blocks are called Logic Array Blocks (LABs) and for Xilinx FPGAs, they are named as Configurable Logic Blocks (CLBs). A high-level overview of an Altera Stratix IV is shown in Figure 1.7. This FPGA serves as the basis for experimental work conducted in this thesis.

In Stratix IV FPGAs, each LAB consists of ten Adaptive Logic Modules (ALMs). An ALM is the basic building block of Stratix IV FPGAs and each ALM contains an 8-input fracturable Look-Up Table (LUT), two embedded adders and two registers as shown in Figure 1.8. The 8-input fracturable LUT can be used to implement any logic function with up to six inputs and certain functions with seven inputs. The two dedicated full adders are capable of two two-bit or two three-bit additions. Two programmable registers are directly embedded in the ALM for an optimal logic-to-register ratio. Signals are transferred between ALMs within the same LAB through local interconnects and between neighbouring LABs through direct link interconnects [11].

2 http://www.altera.com
3 http://www.xilinx.com

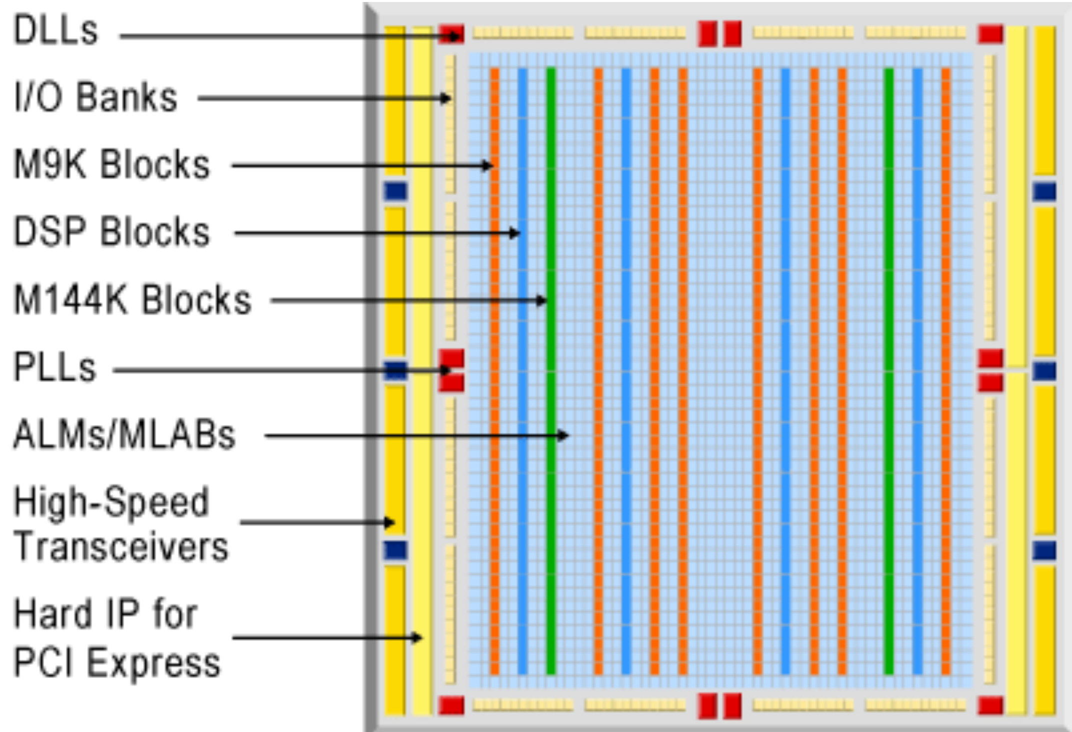


Figure 1.7: Stratix IV FPGA architecture [12].

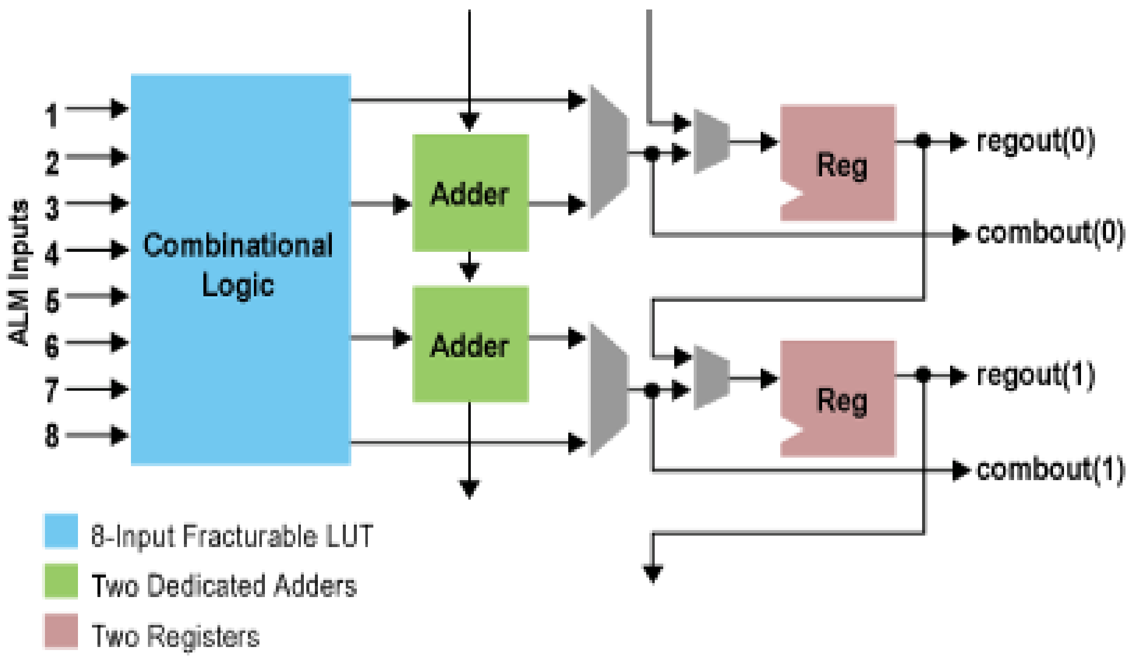


Figure 1.8: Stratix IV FPGA ALM [11].

Stratix IV FPGAs provide two to seven columns of Digital Signal Processing (DSP) blocks that can be used to implement floating point arithmetic functions more efficiently than ALMs alone. The DSP blocks can provide functions like multiplication, multiply-add, multiple-accumulate (MAC) and dynamic shift functions. Biomedical models usually require a large number of mathematical computations which can be efficiently performed by DSPs.

Two types of storage resources, registers and memory blocks, are embedded in FPGAs. Compared to registers, resources for on-chip memory are, in general, abundant. Stratix IV FPGAs offer three different memory types, namely Memory Logic Array Block (MLAB), M9K and M144K, each having differing memory capacities and bandwidths. An increase in memory capacity normally results in a decrease in bandwidth. MLABs have the highest memory bandwidth but their size is limited to only 640 bits. They are useful for shift registers, small First-In First-Out (FIFO) buffers and filter delay lines. M9K memory blocks have 9 kb and are used for general-purpose memory, packet headers or cell buffers. M144K memory blocks have 144 kb of memory and are used for larger general-purpose memory, packet headers or cell buffers.

All the resource elements discussed above are embedded within a switched routing fabric including many short-distance links and a few fast global links which interconnect the elements within the device. This is often referred as *island-style* architecture and is common in modern FPGAs.

1.2.2.2 *Hardware Description Language*

A Hardware Description Language (HDL), such as VHSIC Hardware Description Language (VHDL) or Verilog, can be used to program the structure, design and operation of digital logic circuits. A HDL enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and simulated testing of an electronic circuit. It also allows for the compilation of a HDL program into a lower level specification of physical electronic components, such as the set of masks used to create an integrated circuit [15].

Similar to other programming languages, a HDL is a textual description consisting of expressions, statements and control structures. One important differ-

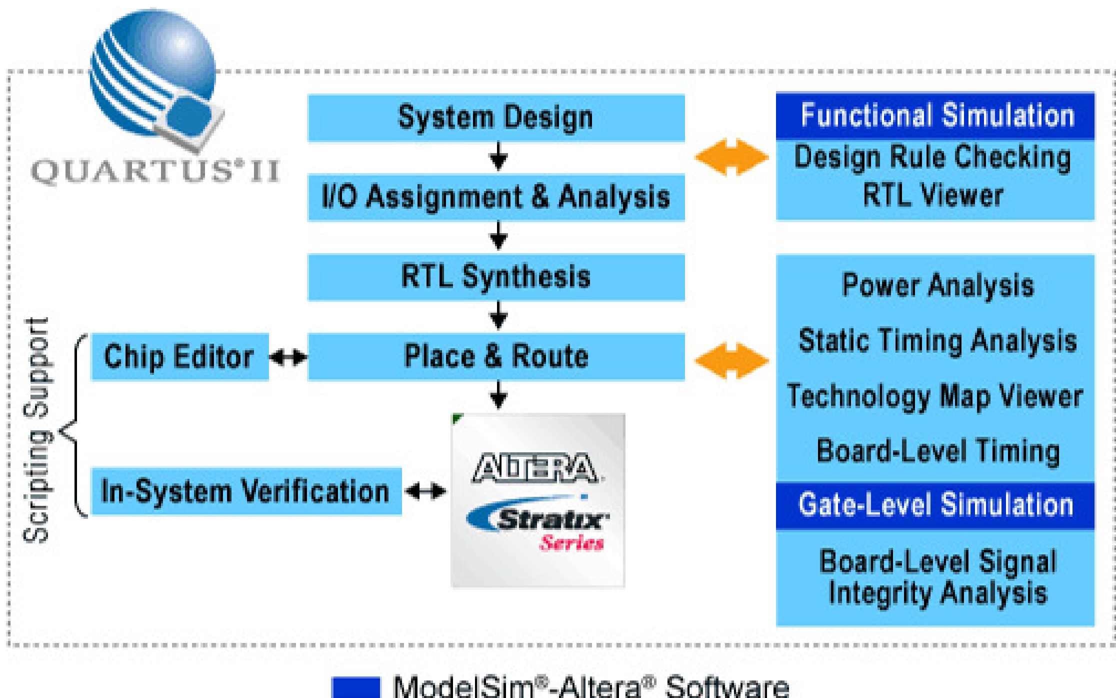


Figure 1.9: Altera's FPGA application design flow [9].

ence between most programming languages and HDLs is that HDLs explicitly include the notion of time, which is the primary attribute in an integrated circuit.

1.2.2.3 *Design Flow*

The FPGA design flow for a typical application including system design, I/O assignment and analysis, Register Transfer Level (RTL) synthesis, place-and-route process, programming and simulations at multiple stages throughout the design process, is shown in Figure 1.9.

After an application has been developed in a HDL, the synthesis engine compiles the design from the HDL sources to an architecture-specific netlist. The design components will then be put through an automated place-and-route procedure to generate a pinout, which will be used to interface with components outside of the FPGA. The final step is the generation of a bitstream programming file in a format that can be downloaded to the target device. The bitstream file is programmed to the FPGA though a connection cable, such as USB-JTAG.

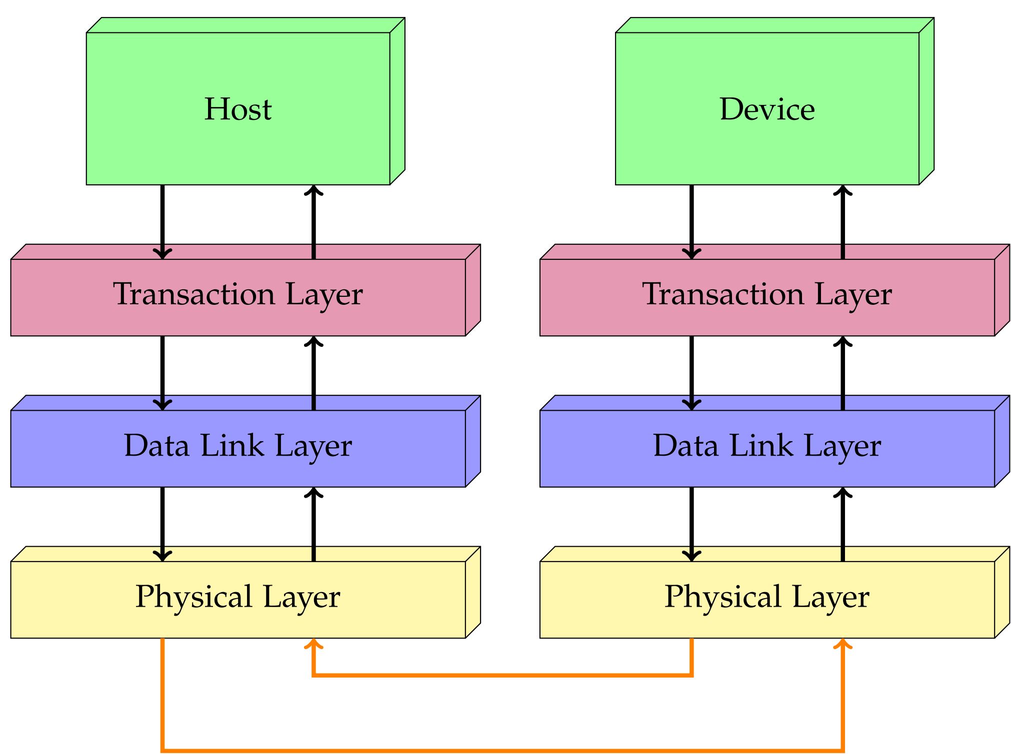


Figure 1.10: PCI Express layered architecture.

1.2.3 *PCI Express*

PCIe (Peripheral Component Interconnect Express) is a scalable, chip-to-chip, high-speed serial expansion bus protocol used in computing and communication. PCIe is based on point-to-point topology, with separate serial links connecting every device to the host machine [14].

1.2.3.1 *Architecture*

The PCI Express architecture is specified in layers as shown in Figure 1.10. The host normally uses a software model or driver to generate read and write requests that are transported through the transaction layer, the data link layer and finally the physical layer to the I/O devices using a packet-based, split-transaction protocol.

TRANSACTION LAYER The transaction layer receives read and write requests from a software model or driver and creates request packets for transmission to the data link layer. All requests are implemented as split transac-

tions and some of the request packets also require a response packet. The transaction layer receives response packets from the link layer and matches these with the original software requests. Each packet has a unique identifier that enables response packets to be directed to the correct originator. The packet format offers 32-bit memory addressing and extended 64-bit memory addressing.

DATA LINK LAYER The primary role of a data link layer is to ensure reliable delivery of the packet across the PCI Express link(s). The data link layer is responsible for data integrity and adds a sequence number and a Cyclic Redundancy Codes (CRC) to packets initiated by the transaction layer.

PHYSICAL LAYER The physical layer contains the fundamental PCI Express links with a dual simplex channel (refered to as a lane), implemented as a transmit pair and a receive pair. A data clock is embedded using the 8b/10b encoding scheme to achieve very high data rates with an initial bandwidth of 2.5 Gb/s/direction (Generation 1). The rates are doubled in each successor generation. The physical layer transports packets between the data link layers of two PCI Express agents. The physical layer provides x1, x2, x4, x8, x12, x16, and x32 lane widths, which conceptually splits the incoming data packets among these lanes. Each byte is transmitted with 8b/10b encoding across the lane(s). This data disassembly and reassembly is transparent to other layers. During initialization, each PCI Express link is set up following a negotiation of lane widths and frequency of operation by the two agents at each end of the link.

1.2.3.2 *PCI Express IP Core*

Interfacing a FPGA to the PCIe bus is not a simple task. Fortunately, there are numerous PCIe cores available provided by FPGA vendors and third parties. In this thesis, the Intellectual Property (IP) Compiler for PCI Express, available in the Qsys design flow provided by Altera, is used. Figure 1.11 shows the block

diagram of the IP core with an Avalon Memory Mapped Interface (Avalon-MM) [7].

Custom variations, generated by the IP Compiler for PCI Express in the Qsys design, provides a bridge interface between the PCI Express transaction layer and other components across the system interconnect fabric through an Avalon MM interface. The hard IP implementation of the PHY (Physical), MAC (Media Access Control) and data link layers of the Open Systems Interconnection (OSI) model communicates with a soft IP implementation of the transaction layer optimized for the Avalon-MM protocol. The Avalon-MM interface helps the PCIe IP core to remove some of the complexities associated with the PCIe protocol and to abstract the addressing, transfer size and packet rules of PCIe [6].

1.2.4 *Floating Point Unit*

Floating-point functionality is used to achieve a high degree of numeric precision and dynamic range that most high performance applications (e.g., radar, sonar, biomedical simulation and financial modelling) require. As many of these applications use FPGAs there is a demand for floating-point capabilities. There are a good number of existing floating point cores provided either by the vendors of FPGAs, or independently developed third party floating point platforms. These cores typically exploit the freedom of a FPGA by allowing for the customisation of variable widths of exponent and mantissa to meet designers specifications. They also offer IEEE-754 standard single and double precision cores.

1.2.4.1 *IEEE-754 Standard for Floating Point Format*

IEEE-754 standard floating point is the most common representation for real numbers. It is used in computer systems ranging from large servers to small embedded systems and is supported by all major operating systems and programming languages. In this thesis, the floating point numbers in our hardware accelerator follow the IEEE-754 standard. This allows the accelerator to produce accurate results which are compatible with pure software solutions.

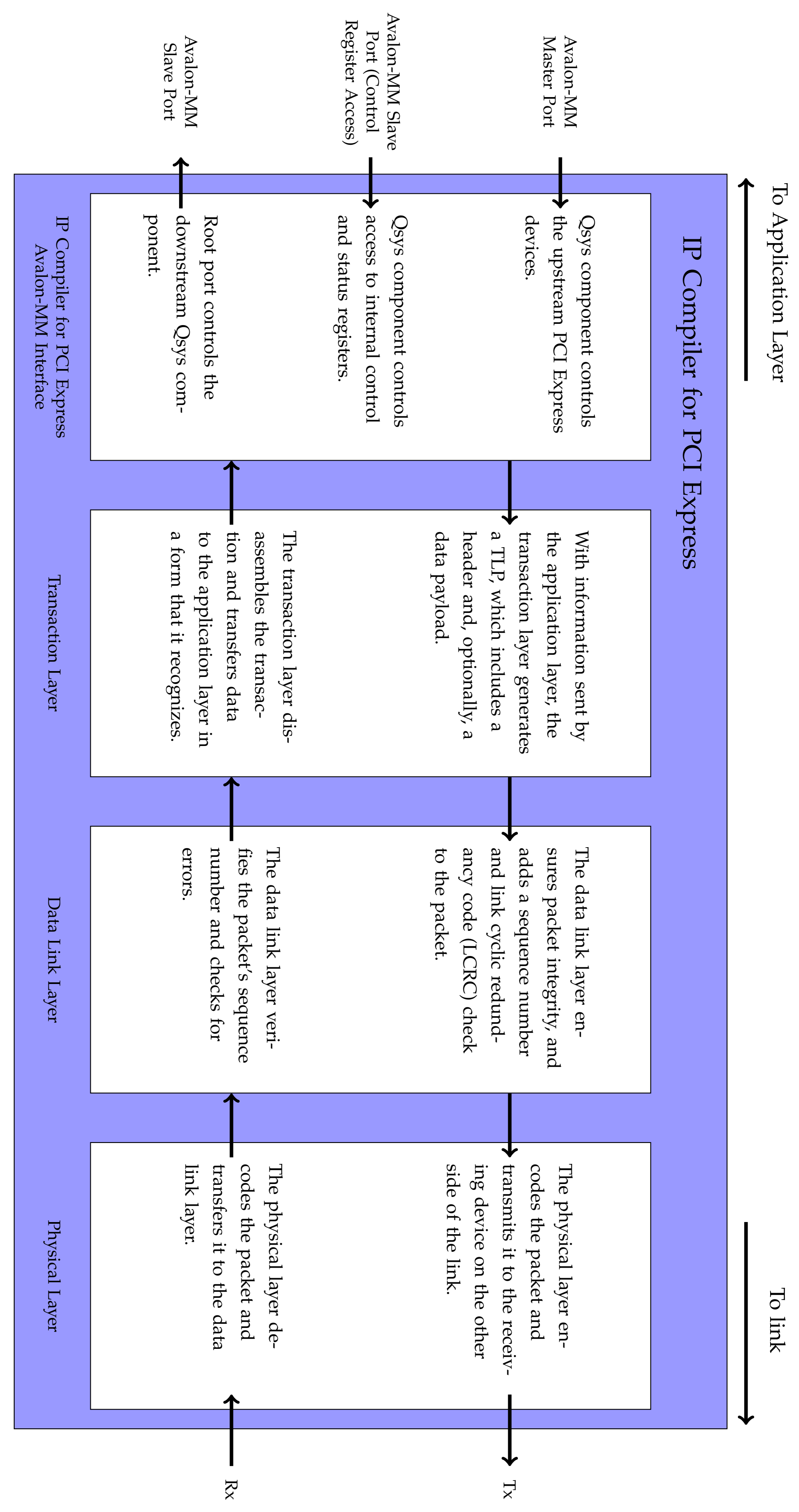


Figure 1.11: IP compiler for PCI express with Avalon-MM interface [6].

Figure 1.12: IEEE-754 floating point format (S represents a sign bit, E represents an exponent field, and M is the mantissa field).

Numbers	Sign	Exponent	Mantissa
0	Don't care	All 0's	All 0's
Positive Denormalised	0	All 0's	Non-zero
Negative Denormalised	1	All 0's	Non-zero
Positive Infinity	0	All 1's	All 0's
Negative Infinity	1	All 1's	All 0's
Not-a-Number (NaN)	Don't care	All 1's	Non-zero

Table 1.2: IEEE-754 special case numbers.

The IEEE-754 floating point formats have binary patterns of the form shown in Figure 1.12. A normal floating point number can be represented by the following equation where b represents the exponent bias and m represents the number of bits in the mantissa field:

$$value = (-1)^S (1 + \sum_{i=1}^{m} M_{m-i} 2^{-i}) \times 2^{(E-b)}$$

In addition to the normal numbers, the IEEE-754 standard also defines some special case numbers as shown in Table 1.2.

The IEEE 754-1985 standard provides definitions for four levels of precision, of which the two most commonly used are single precision and double precision. Table 1.3 describes the binary patterns, range and accuracy of these two precisions.

1.2.4.2 *Altera Floating Point Megafunctions*

Altera provides IEEE 754-compliant floating-point megafunctions for its device family. The key floating point megafunctions Altera supports include addition and subtraction, multiplication, division, square root, compare, logarithm, exponential function, inverse, etc. All the floating point (FP) cores are pipelined. The performance of a floating point computation is influenced by the fre-

<table>
<tr><td rowspan="4">Single Precision</td><td>Width</td><td>Sign Bit</td><td>Exponent Bits</td><td>Mantissa Bits</td><td>Bias</td></tr>
<tr><td>32</td><td>31</td><td>30...23</td><td>22..0</td><td>127</td></tr>
<tr><td colspan="3">Range</td><td colspan="2">Precision</td></tr>
<tr><td colspan="3">$\pm 1.18 \times 10^{-38}$ *to* $\pm 3.4 \times 10^{38}$</td><td colspan="2">approx. 7 decimal digits</td></tr>
<tr><td rowspan="4">Double Precision</td><td>Width</td><td>Sign Bit</td><td>Exponent Bits</td><td>Mantissa Bits</td><td>Bias</td></tr>
<tr><td>64</td><td>63</td><td>62...52</td><td>51...0</td><td>1023</td></tr>
<tr><td colspan="3">Range</td><td colspan="2">Precision</td></tr>
<tr><td colspan="3">$\pm 2.23 \times 10^{-308}$ *to* $\pm 1.80 \times 10^{308}$</td><td colspan="2">approx. 15 decimal digits</td></tr>
</table>

Table 1.3: IEEE-754 single and double precision formats.

quency at which the operators run at and the pipeline latency of the operator hardware. When designing for maximum FP performance in a FPGA, the total number of operators that can be placed in a FPGA is vital. As such, the Altera floating-point megafunctions can be customised in many different ways to fine-tune FP performance, power consumption and area usage to meet the application-specific requirements [5]. The configurable features include:

- Single and double-precision selection;
- Single extended configurable precision;
- Operator latency versus area tradeoff;
- Reduced functionality;
- Optional denormalized number support;
- Reduced rounding accuracy;
- Optional indefinite support;
- Support for dedicated multiplier circuitry (multiplier only);
- Optional add or subtract-only mode (adder or subtracter only).

1.2.4.3 *FloPoCo*

FloPoCo [37] is an open source generator of arithmetic cores for FPGAs. In contrast to IEEE floating point representations, FloPoCo has a special floating point format which has an additional two-bit prefix. The two bits are only used

to signal special case numbers, namely 00 for zero, 01 for normal numbers, 10 for infinities, and 11 for NaN. In IEEE-754 format, these exception signals are handled by the exponent and mantissa as shown in Table 1.2. The advantage of the FloPoCo format is that it saves quite a lot of decoding/encoding logic. The main drawback of this format is when results have to be stored in memory as they require two additional bits. However, as the FPGA embedded memory can accommodate 36-bit data, the addition of two bits to a 32-bit IEEE-754 format is harmless as long as data resides within the FPGA.

FloPoCo supports floating point operations including addition and subtraction, multiplication, division, square root, logarithm, exponential function, power, etc. FloPoCo is used as a command-line tool to create arithmetic cores. A VHDL based floating point core can be created with the numbers of bits in the exponent and mantissa specified by the user. For example, an execution of

```
flopoco FPDiv 8 23
```

will produce a file $flopoco.vhdl$ containing the floating point division core with single precision format.

In addition, FloPoCo also provides customisation options that allows users to manipulate resources, frequency and latency to suit their applications. For example:

- *-target=Stratix4* sets the target device to the Stratix IV family. Having this option set will target the highest speed grade available for the device family;
- *-pipeline=yes* instructs FloPoCo to produce a pipelined core;
- *-frequency=300* sets the target frequency (in MHz), this option is used when the *-pipeline* option is set. FloPoCo will try to pipeline the operators to the target frequency;
- *-useHardMult=yes* instructs FloPoCo to use hard multipliers or DSP blocks wherever possible;
- *-unusedHardMultThreshold=0.3* instructs FloPoCo to use a hard multiplier (or DSP block) if less than 30% of the hard multipliers are unused. The

ratio is between 0 and 1, such that 0 means: any sub-multiplier that does not fully fill a DSP goes to logic; 1 means: any sub-multiplier, even very small ones, will consume a DSP.

A FloPoCo operator can be either combinational or pipelined, which is controlled by the *-pipeline* and *-frequency* options. With the pipelined implementation, registers maybe inserted to reach a target frequency. However, the pipeline built by FloPoCo may depend on the target device and the effort is always tentative [36].

1.3 HIGH-LEVEL SYNTHESIS

High-level Synthesis (HLS) is an automated design process that interprets a high-level description of a design and creates digital hardware that implements that design [32]. The synthesis begins with a high-level specification of the problem, for example, high-level languages like C, state diagrams or logic networks. The code is analysed, architecturally constrained, and scheduled to create a Register Transfer Level (RTL) HDL, which is then, in turn, commonly synthesized to the gate level by the use of a conventional logic synthesis tool.

1.3.1 *Benefit*

It is common knowledge that the RTL creation process for hardware implementations is much more time consuming and error prone than an equivalent software development. The main benefit of HLS is to avoid this problem by automating the RTL implementation process and providing an error-free path from an abstract specification to RTL and hence significantly reducing the design and verification efforts.

1.3.2 *Design Processes*

The HLS process consists of a number of stages. Different HLS tools may vary their design process or order. Some frequently used processing stages are discussed below [66].

LEXICAL PROCESSING HLS synthesis begins with an algorithmic description of the design expressed in a high-level language. Lexical processing parses the high-level language source code and transforms it into an internal representation which is similar to the high-level language compilation.

DESIGN OPTIMIZATION Optimizations that can be performed on the design itself include common subexpression elimination and constant folding. Many of these optimizations are commonly used in high-level language compilers or parallelising compilers.

CONTROL/DATAFLOW ANALYSIS The inputs, outputs, and operations of the design are identified and the data dependencies between them are determined. The result of this process is usually a Control/Dataflow Graph (CDFG) which determines the order of the computation.

LIBRARY PROCESSING The RTL implementation produced by HLS will depend on the capabilities and characteristics of the library of parts available for the specific implementation technology to be used. Library processing reads the available libraries and determines the functional, timing, and area characteristics of the available parts.

RESOURCE ALLOCATION Resource allocation establishes a set of functional units that will be adequate for implementation of the design. In many behavioural synthesis systems, an initial resource allocation is performed and subsequently modified during scheduling and/or binding.

SCHEDULING Scheduling introduces parallelism and the concept of time. It transforms the algorithm into an FSM (Finite State Machine) representation. Using the data dependencies of the algorithm and the latencies of the functional units in the library, the operations of the algorithm are assigned to specific clock cycles. There are often many possible schedules. Directives that constrain the result with respect to latency, pipelining, and resource utilization will affect the schedule that is chosen.

FUNCTIONAL UNIT BINDING Binding assigns the operations of the algorithm to specific instances of functional units from the library.

REGISTER BINDING In cases where values are produced in one clock cycle and consumed in another, these values must be stored in registers or memory. The register binding process allocates registers as needed and assigns each value to a physical register. Analysis of the lifetime of each data value can identify opportunities to use the same physical register to store different values at different times. This is done to reduce the size of the resulting design.

OUTPUT PROCESSING The datapath and finite state machine resulting from all of the previous steps are written out as RTL source code in the target language. This code can be structured in a number of ways to optimize the downstream logic synthesis process or to enhance the readability of the code.

1.4 THESIS MOTIVATION AND CONTRIBUTIONS

1.4.1 *Motivations*

Benefiting from the development of HPC in recent decades, the number of degrees-of-freedom (DOFs) used in biomedical modelling has increased rapidly in response to increased model complexity and increased model accuracy requirements. In order to reduce run times, parallel computing is now becoming increasingly important as individual CPUs reach the physical lim-

its of processor technology. Simulations involving the numerical integration of models (such as using CellML with OpenCMISS to simulate large multi-scale physiology) are generally limited by the available computational hardware and the acceptable duration of simulation. For certain simulations in OpenCMISS, a CellML model needs to be evaluated a very large number of times, resulting in a significant computational time. In order to reduce this time, we can take advantage of the fact that each instance of a CellML model at a particular DOF is completely independent from the CellML models at every other DOF and so it is possible to evaluate the models in parallel. Therefore, special purpose hardware, in particular FPGAs, is very promising for accelerating these kinds of computations and are expected to lead to higher performance at lower cost and less power consumption.

Unfortunately, there are two major problems that have to be overcome. First, developing a FPGA hardware design for a given application is much more complex, time consuming and error prone than programming general purpose processors. Second, integrating the general purpose processors in parallel computing systems with the reconfigurable computing capacity of the FPGAs is not trival. Therefore, despite the benefits that heterogeneous computing has to offer in the area of science and technology, the existing tools that enable development for FPGA-platforms are highly dependent on hardware design expertise, i.e., an excellent understanding of a HDL and fine-grained digital hardware architecture. This impedes the ability of biomedical scientists/engineers to explore acceleration on such platforms, and creates a wide gap between their speciality and the vast computational capacity of FPGAs.

Being able to automatically generate Hardware Accelerator Modules (HAMs) from existing high-level model descriptions, e.g., CellML models, would open up the use of the FPGAs to biomedical scientists/engineers. This vision leads to the key motivations for this thesis:

- To provide a high performance hardware/software co-design framework for biomedical simulations;
- To design and develop a domain specific high-level synthesis process that enables the generation of the above framework in an automated way;

- To demonstrate that domain specific high-level optimisation can deliver competitive performance and lower energy costs;
- To investigate and develop automatic methods and technologies for optimised automatic hardware generation.

1.4.2 *Contributions*

The following major novel and innovative contributions have been made while undertaking the research presented in this thesis:

- Exploration into the current state of the art of biomedical modelling and simulation with hardware acceleration and HLS, identifying areas of improvement and new ways to exploit parallelism;
- Design and development of **a parallel floating point pipeline based on a hardware accelerator module** for biomedical models. The presented module is embedded in a proposed hardware/software co-design framework that can be integrated with biomedical simulators;
- Investigation and design of **a domain specific HLS tool for biomedical modelling** to automatically create the designed hardware accelerating modules from high-level descriptions of biomedical models. The tool is named ODoST, standing for ODE-based Domain-specific Synthesis Tool, and it allows biomedical scientists/engineers (without hardware design expertise) to perform simulations with the use of HAMs;
- Investigation of **performance optimisation and resource utilisation strategies** for large hardware computing designs. These strategies are used in the HLS processes to create hardware acceleration modules with better performance and resource usage. Furthermore, the framework is general and can be adopted for other applications with floating point computations.
- Extensive **experimental evaluation on real hardware of the generated hardware accelerator modules** regarding resource usage, scalability, per-

formance and power consumption, including a comparison of performance and power consumption with the corresponding models in CPU and GPU implementations.

1.5 THESIS STRUCTURE

This thesis is structured as a compilation of publications. References in the papers have been adjusted to cross-references within this thesis. The work is presented following the outline below.

This chapter describes, in detail, the background of the research. It presents a brief overview of CellML and OpenCMISS. A number of representative CellML models are explained and selected as the base models for the HAM development and evaluation in later chapters. The chapter then presents the concepts, techniques and tools of reconfigurable computing used in the thesis. At the end, a summary of the motivation and contributions of this research is presented.

Chapter 2 presents the initial design and development of the hardware accelerator module with a hardware/software co-design framework. This module is implemented manually, and evaluations are performed to obtain preliminary results of the design. The content of this chapter is published at the Field-Programmable Technology (FPT) Conference [100].

In Chapter 3, a domain-specific high level synthesis tool called ODoST is investigated and designed, mainly based on the accelerator design discussed in Chapter 2. HAMs are generated automatically using ODoST and an in-depth evaluation is performed, including a comparison with pure software and GPU designs. The content of this chapter is submitted as a manuscript and is currently under review for publication in the ACM Transactions on Reconfigurable Technology and Systems.

Chapter 4 proposes several general optimisation strategies, including source-to-source compiler optimisation, resource balancing and parallel pipelines to further increase the performance of HAMs and to better use the capabilities of the target devices. The proposed strategies have in common that they still

can maintain the automatic nature of the overall process of the FPGA implementations. The optimised HAMs are evaluated and compared against CPU and GPU designs as well as non-optimised HAM implementations. This work is submitted for publication as a research article in the Journal of Concurrency and Computation: Practice and Experience.

Finally, Chapter 5 concludes this thesis. The contributions and outcomes of this work are summarised and reconsidered within the context of the motivations, and a number of directions and suggestions for future research are presented.

2 HARDWARE ACCELERATOR MODULE

This chapter presents the initial design and development of the hardware accelerator module, along with a hardware/software co-design framework. The contents of this chapter are based on the published paper in Proceedings of the International Conference on Field Programmable Technology, FPT'13 [100].

Contributions in this chapter are: (i) investigation of biomedical models for code portions that are suitable for hardware acceleration, (ii) design of the hardware/software co-design framework purposed for the hardware accelerator, and (iii) development of a manual implementation of a hardware accelerator module based on the co-design framework for the identified computation kernel.

Preliminary evaluation results show that (i) the hardware accelerator module gains significant speedup compared to a pure software implementation, (ii) the scalability of performance results indicates the potential for further performance improvements with a more complex designs, and (iii) a manual implementation of the module is impractical and an auto generation process is required.

CHAPTER ABSTRACT

OpenCMISS is a mathematical modeling environment designed to solve field based equations and link subcellular and tissue-level biophysical processes to organ-level processes. It employs a general purpose parallel design, in particular distributed memory, for its computations. CellML is a mark up language based on XML that is designed to encode lumped parameter, biophysically based, systems of ordinary differential equations and nonlinear algebraic equations. OpenCMISS allows CellML models to be evaluated and integrated into models at various spatial and temporal scales. With good inherent parallelism, hardware acceleration based on FPGAs has a great potential to increase the computational performance and to reduce the energy consumption of computations with CellML models integrated into OpenCMISS. However, with over several hundred CellML models, manual hardware implementation for each CellML model is complex and time consuming. The advantages of FPGA designs will only be realised if there is a general solution or a tool to automatically convert CellML models into hardware description languages such as VHDL. In this chapter the architecture for the FPGA hardware implementation of CellML models are described and the first results related to performance and resource usage based on a variety of criteria are evaluated.

2.1 INTRODUCTION

OpenCMISS[1] is a general purpose computational library for solving field based equations with an emphasis on biomedical applications [24]. It uses a distributed memory system architecture in order to solve large scale coupled models, such as an electrical activation problem at high spatial resolutions.

OpenCMISS is typically designed to link subcellular and tissue-level biophysical processes into organ-level processes. It uses CellML[2] [34], an open standard mark up language based on XML, to define custom mathematical

1 http://www.opencmiss.org
2 http://www.cellml.org

models to form parts of a larger model. Variables in CellML models are linked to field variable components directly and define the value of each degree-of-freedom (DOF). Mathematical models represented by CellML, by their nature, are regular, relatively small but performance-critical and highly data parallel. As such, special purpose hardware, in particular FPGAs with large amounts of fine-grained parallelism, are very promising for accelerating CellML models. Integrating the use of FPGA's into the parallel processing of OpenCMISS has the potential to lead to higher performance with reduced energy consumption.

However, compared to technologies such as multicore processors and GPUs, FPGAs are not widely adopted to accelerate applications. There are two major reasons for this. First, developing a FPGA hardware design of a given application is much more complex, time consuming and error prone than programming general purpose processors. Second, it is hard to integrate general purpose processors in parallel computing systems with FPGAs (referred to as hybrid systems).

The hardware acceleration of OpenCMISS and CellML applications involves three components: the CellML hardware acceleration component which is basically a number of iterated floating point ODEs (Ordinary Differential Equations) computed with FPGAs, the data path acceleration framework which is represented by the FPGA-CPU heterogeneous architecture and the generation tool to automatically create the first two components from a specific CellML model.

In this chapter, a FPGA-CPU heterogeneous architecture for OpenCMISS is proposed to link with CellML hardware models via a PCIe interface. The design has been implemented on an Intel workstation using an Altera DE4 FPGA board. The implementation is a functioning proof of concept system which is yet to be optimised. Initial performance and resource usage results have been obtained, and the scalability of the system has been analysed.

The chapter is organized as follows. Related work is discussed in Section 2.2. In Section 2.3, a typical OpenCMISS example using a CellML model and the CellML hardware architecture are discussed and analysed. The implementation of the heterogeneous architecture especially its data path is described in

Section 2.4. In Section 2.5, the first experimental results are presented and the potential optimization strategies are discussed. The chapter is concluded in Section 2.6.

2.2 RELATED WORK

There are a number of works on the floating point optimization of FPGA based systems. Some studies focus on the optimization of one or several floating point operations on FPGAs [1, 38, 60] while the others use those floating point tools or generators to optimize mathematical problems [84, 88]. Several floating point libraries including Altera's Megafunctions [5], DSP Builder [3] and FloPoCo [36] are considered. In this chapter, FloPoCo is used since it alone offers the unique combination of features required: it scales from single precision to double precision, it is pipelined, and it is open-source. However, our approach is general and open to other tools or their combination.

Several heterogeneous acceleration frameworks for energy efficient scientific computing have been proposed in recent years. Kapre and DeHon [58] have presented a parallel, FPGA-based, heterogeneous architecture customized for the spatial processing of sparse, irregular floating-point computations. They reported that their architecture performed better than conventional processors because of better resource utilization and lower-overhead dataflow with fine grained tasks. Anandaroop, Somnath and Swarup [47] have proposed a heterogeneous mapping framework that uses embedded memory blocks in a FPGA and proved that such a system significantly improved the energy efficiency of applications which are dominated by complex data paths and/or functions. Nallamuthu et al. [69] have used a FPGA-based coprocessor to accelerate the compute-intensive calculations of a popular biomolecular simulator, LAMMPS, and achieved a 5.5 times speed-up.

To the best of our knowledge, this research is the first implementation of a CellML hardware model based on a FPGA-CPU heterogeneous architecture, although OpenCMISS has also considered GPGPUs for code acceleration. The CUDA results are promising compared to the CPU only implementation. Note,

however, that a number of case studies [45, 59, 63] have shown that FPGAs can achieve lower energy consumption when compared to GPUs and CPUs and are well suited for small, highly parallel and performance critical kernels such as CellML models.

2.3 CELLML HARDWARE MODEL

2.3.1 *A Motivating Example*

The motivation of our study came from an estimation of the future electrical activation problem of the human heart. The average human heart volume is approximately $8.19 \times 10^5\,\mathrm{mm}^3$ and assume that 50% of the human heart volume is ventricular tissue. To discretise the ventricle volume into grids with $100\,\mu\mathrm{m}$ spacing would require 4.23×10^8 grid points. At each grid point a system of ODEs needs to be solved at each time instance. If a model with 30 ODEs is used and assuming that 100 FLOPS are required for one ODE calculation, to simulate the model at each time instance would require 1.27×10^{12} FLOPS. With a 1 ms time stamp, to simulate one minute of real activation would require 7.62×10^{16} FLOPS. If, for example, a processor could compute 20 GFLOPs per core [78] then a single core would require approximately 44 days for a simulation.

2.3.2 *Model Overview*

CellML models by their nature are regular and ideally suited for parallelisation as each CellML model is independent and thus can be integrated in parallel. A CellML model can be divided into components. Components are represented by a number of equations and each component is itself a CellML model which can be reused in the future studies or other models. OpenCMISS encapsulates all interaction with CellML models within a CellML environment.

For the purposes of this chapter the Hodgkin-Huxley CellML model of a giant squid axon [53] is considered. The model contains 8 components. We com-

bine the *"sodium_channel"*, *"potassium_channel"*, *"leakage_current"* and *"membrane"* components to a super *"membrane_potential"* component because the *"membrane"* component is dependent on the other three. The *"environment"* component is left out since there is no equations in the component. Therefore, it eventually ends up with four components as listed in Table 2.1..

Component Name (ID)	Equations	Add/Sub	Mul	Div	Exp
membrane_potential (V)	5	6	10	1	0
sodium_channel_m_gate (m)	3	4	4	2	2
sodium_channel_h gate (h)	3	4	3	3	2
potassium_channel_ngate (n)	3	4	4	3	2
Total	14	18	21	9	6

Table 2.1: Number of equations and floating point operations in each component of the Hodgkin-Huxley model (the equations have been optimised with common subexpression extraction and power elimination, see Chapter 4 for the details).

From the model consider the following equations which represent the *"sodium_m_gate"* component in the Hodgkin-Huxley model.

$$alpha_m = \frac{0.1 \times (V + 25)}{e^{\frac{V+25}{10}} - 1} \tag{2.1}$$

$$beta_m = 4 \times e^{\frac{V}{18}} \tag{2.2}$$

$$\frac{dm}{dt} = alpha_m \times (1 - m) - (beta_m \times m) \tag{2.3}$$

where V is the trans-membrane voltage and m is a state variable for the sodium channel activation gate. This component is aimed at calculating the rate of the change for the state variable m at time t. $alpha_m$ and $beta_m$ are first calculated and stored as the intermediate variables. $\frac{d}{dt}(m)$ is the rate variable that corresponds to the state variable m and is calculated depending on the two intermediate variables (also called algebraic variables). After variable $\frac{d}{dt}(m)$ is calculated, a numerical integration method will be used to approximate the state value of m at time $t + \triangle t$. There are a variety of such numerical integra-

tion algorithms and in this thesis, Euler's method is used. The computation of the state variable m at time $t + \triangle t$ is represented in Eq. (2.4).

$$m_{t+\triangle t} = m_t + \triangle t \times \frac{d}{dt}(m) \tag{2.4}$$

The interaction between OpenCMISS and CellML is illustrated in Figure 2.1. The OpenCMISS framework for a simulation consists of one or more regions containing high spatial resolution meshes. The equation sets are formed using fields defined over these meshes. For a simulation OpenCMISS integrates each cell spatially. The modeller chooses the CellML variables that interact with OpenCMISS fields and marks them as "known" or "wanted". Once the known or wanted status of each CellML variable has been set, the CellML model is ready to be generated. Upon finishing the creation of the CellML environment in a region, OpenCMISS invokes the code generation service of the CellML API. This service automatically generates a C or Fortran function/subroutine from the MathML description of the CellML model. This function/subroutine is then compiled and dynamically linked into the OpenCMISS exectuable. During the simulation, as shown in Figure 2.1, OpenCMISS calls $cellml_integrate()$ and $spatial_solve()$ for each time step. For each $cellml_integrate()$ call, OpenCMISS passes the values of "known" variables stored in the fields to a $cell_integrate()$ function which will call the C or Fortran function/subroutine $cell_calculate()$ which has been generated by the CellML code generation service.

For the Hodgkin-Huxley model, V (the transmembrane voltage) is required for the spatial solve. The m state variable is used for determining i_Na, the sodium current, which in turn changes the transmembrane voltage. If each $cell_calculate()$ call computes 1 ms of the cell activation then, in order to achieve more accurate simulation results, the period is divided into 1000 smaller time intervals and compute one cell with a 1 μs time interval for each iteration. After each iteration, numerical integration method such as Euler's method is used to integrate the m variable. At the last iteration, m returned back to OpenCMISS for spatial integration.

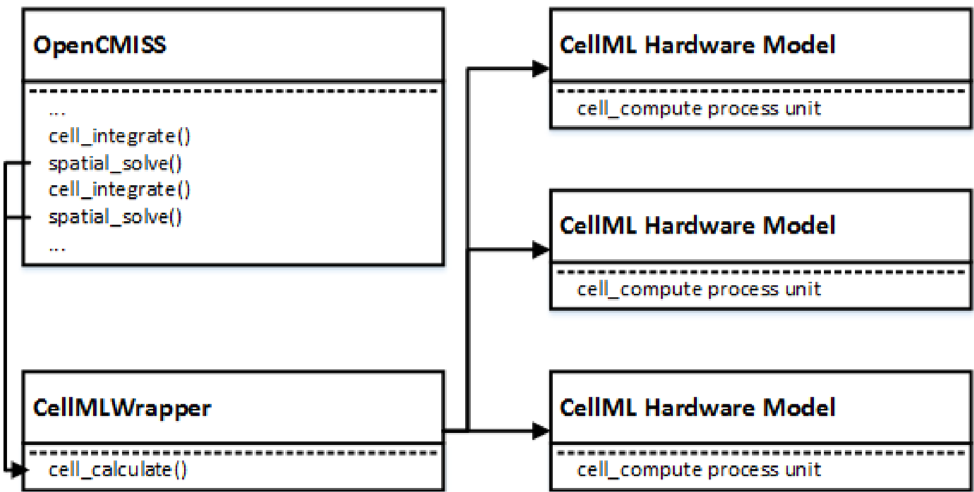


Figure 2.1: Abstract view of model interaction.

A hardware/software integrated CellML model is developed to replace the pure CellML software implementation. The rest of this section explains the CellML hardware model architecture for the FPGA side and Section 2.4 describes the overall system architecture, including how data are exchanged between the host computer and the FPGA. The ultimate aim for this research is to implement a CellML hardware generator that will be an add-on for the CellML code generation service. The service is aimed at automatically generating the hardware/software co-design CellML model and the strategies for this are discussed in Section 2.5.

2.3.3 *Pipelined Floating Point Operations*

Each CellML model contains a set of ODEs and arithmetic operations are, hence, key components of a CellML hardware model. Frequency and area are the two main factors that measure the quality of an arithmetic operation on FPGAs. As each CellML model is independent, they can be integrated in parallel. In addition the computational logic in CellML hardware models can use a pipelined architecture for increased performance.

During the computation, the number of pipeline stages is negligible compared to the number of cells passed into the pipeline data path and hence all pipeline stages are active most of the time. Therefore, latency in the model is not a significant criterion and the objective is to generate a circuit with high throughput. In turn, throughput is determined by the number of parallel cell hardware models in the FPGA and the frequency they operate at.

Our CellML hardware model uses FloPoCo, a floating point core generator, to create the pipelined arithmetic operators. This tool provides great flexibility for generating floating point operations in VHDL from C++ code. In order to generate a floating point core, FloPoCo receives an input of the core operation features, such as target frequency, use of a pipeline, single or double precision, enable or disable the Digital Signal Processing (DSP) blocks and the FPGA manufacturer and model. The output is a synthesizable VHDL file with the required input features. With this tool it is possible to change from a single precision to a double precision pipelined floating point core by only changing the core generator parameters and thus saving rework.

The ASAP (As Soon As Possible) clock cycle scheduling algorithm is adopted as shown in Figure 2.2. It presents the pipelined datapath flow for Eqs. (2.1 - 2.4) as discussed in Section 2.3.2. In the framework, each operation has its own associated latency and are all different from each other. For example, $fadd$ has a latency of 12 cycles and $fmul$ has a latency of 4 cycles. This is because the$fmul$ block is implemented using the hard DSP blocks which are very area efficient, but the $fadd$ block is implemented solely with FPGA logic elements. To ensure a high operating frequency, circuits implemented with FPGA logic elements such as $fadd$, $fdiv$ and $fexp$ should be pipelined to a greater degree than those dominated with the DSP blocks like $fmul$. In order to balance the pipeline, register delays are inserted. In the computation of the intermediate variable $alpha_m$, a 29-stage register path is inserted into the graph to fully balance the pipeline. Register paths are also inserted during the rate of change calculation and the numerical integration.

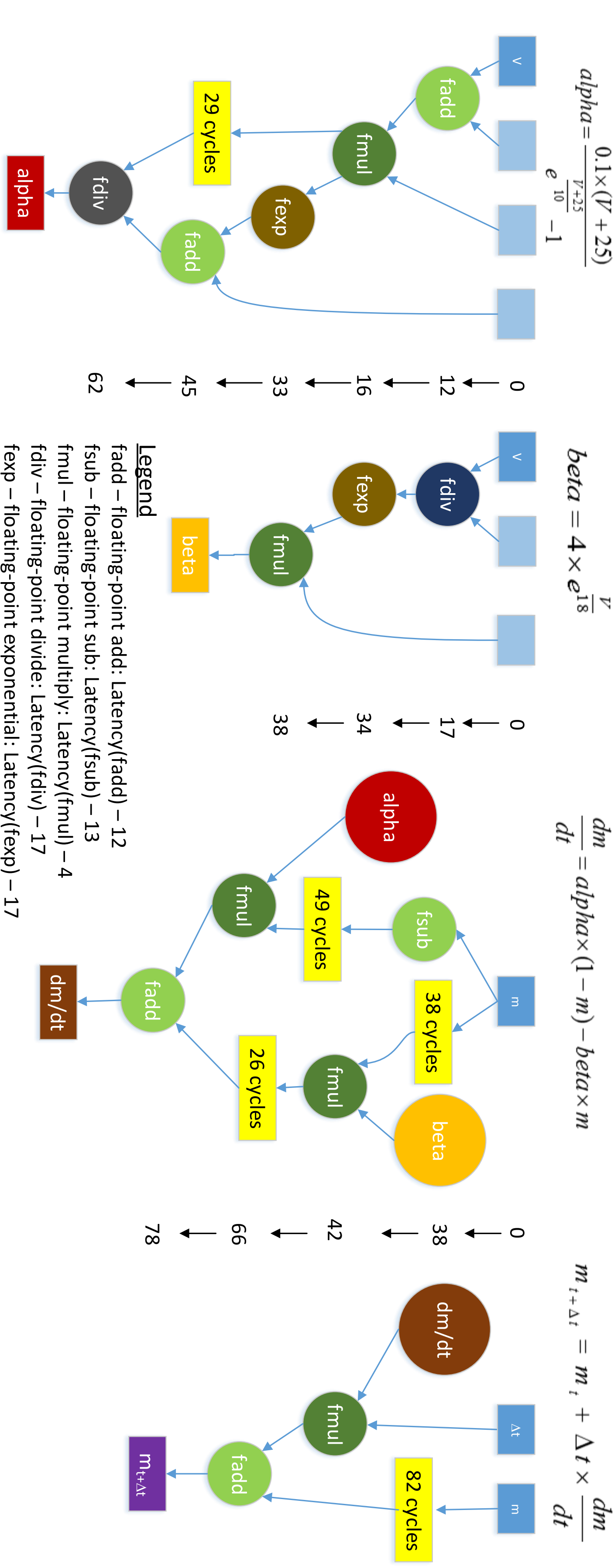


Figure 2.2: Pipeline scheduling for *sodium_channel_m_gate* component.

2.3.4 *The Hardware Model Architecture*

The simplified CellML hardware model architecture is shown in Figure 2.3. The *CellMLCore* reads the input variables from the I/O interface. The dashed box contains the fully pipelined arithmetic components and represents one complete computation iteration for a CellML model. A multiplexer is used for input control where the inputs for the first iteration of a cell are from the initial value of the time step m_T and the rest is from the outputs of the previous iteration. The *control* is used to select the right inputs. After each iteration computation is finished, the output $m_{t+\Delta t}$ is passed into a demultiplexer. The output from the last iteration is directly passed to the I/O interface and the outputs from the other iterations are passed back to the multiplexer for the next iteration computation. A *counter* is used to determine when $m_{T+\Delta T}$ is available.

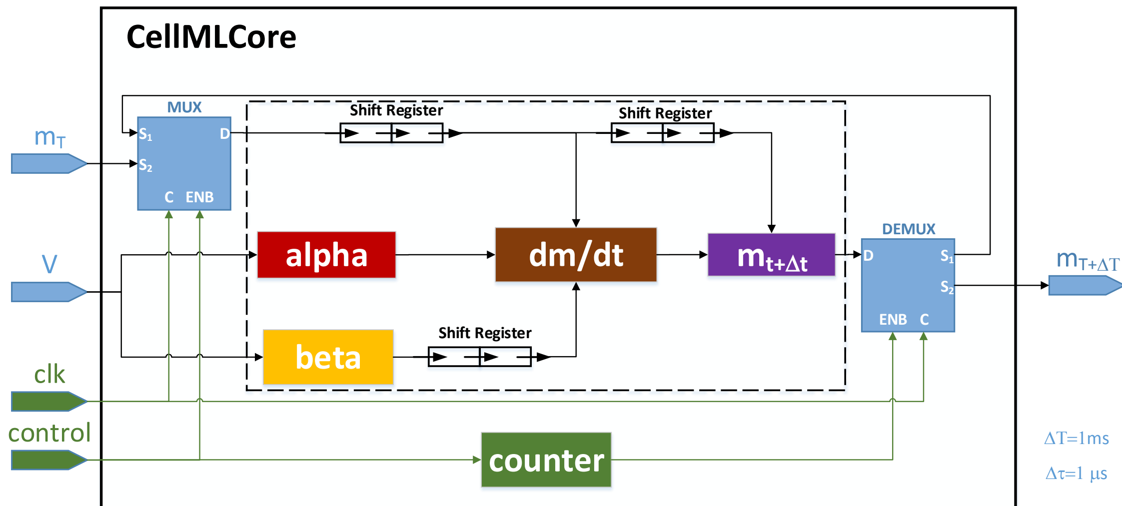


Figure 2.3: CellML hardware model core structure.

2.4 SYSTEM DESIGN AND IMPLEMENTATION

2.4.1 *Overall System Architecture*

The block diagram of the overall framework of the system is shown in Figure 2.4. It is composed of a host computer and a FPGA board connected through the PCIe interface. The arrows indicate the datapath throughout the system. As described in Section 2.3, OpenCMISS stores variables in fields and interacts with the *CellMLWrapper* by calling the *cell_calculate*() function. The *CellMLWrapper* is used as a bridge application and interacts with the FPGA by sending and receiving data through the PCIe interconnects.

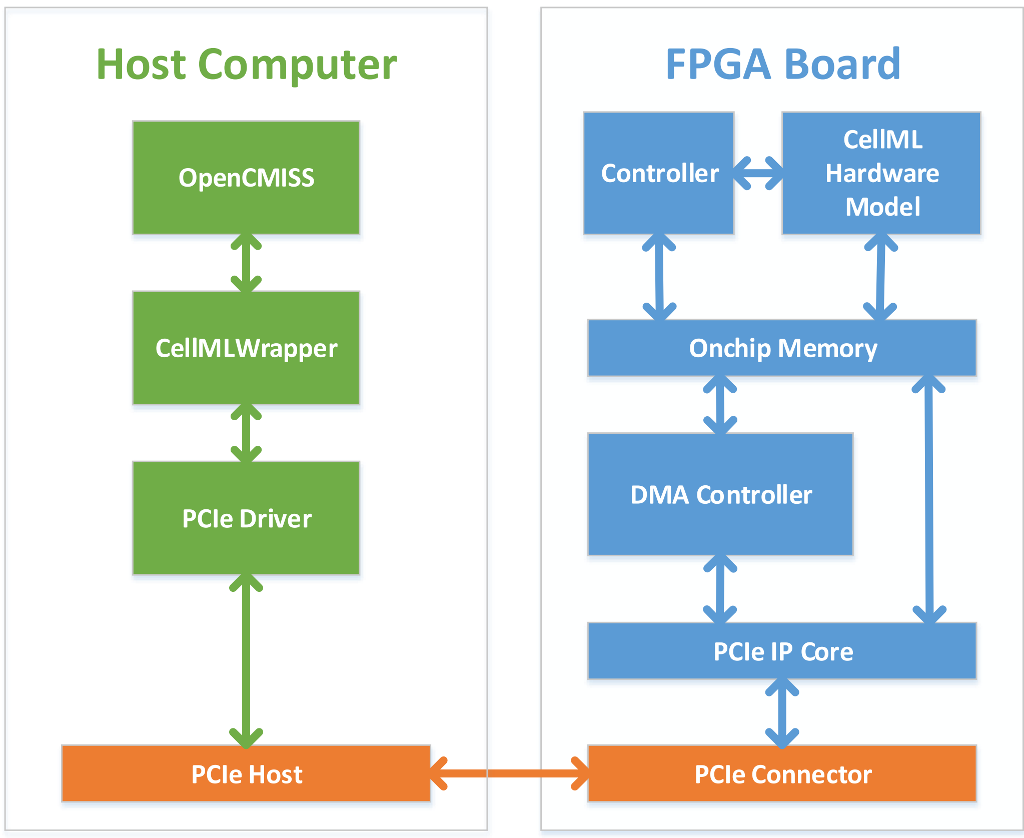


Figure 2.4: A block diagram of the overall system architecture.

On the FPGA side, there is a PCIe IP core that interacts with the PCIe connector and maps to on-chip memory together with the DMA controller for Direct Memory Access. The data received from the host computer is writ-

ten into on-chip memory through the DMA controller. A controller is used to send/receive signals to/from the host computer and interact with the CellML hardware model to control the data transfer.

2.4.2 *Host Computer Design*

On the host computer, there are three major executing components: the simulation software, OpenCMISS, the *CellMLWrapper* and the PCIe driver. OpenCMISS provides the "known" variables to the CellML model which returns "wanted" variables to it. In this research, the focus is on the $cell_integrate()$ function/subroutine that calls *CellMLWrapper* from OpenCMISS, so the entire design and implementation of OpenCMISS is encapsulated and can be ignored here.

The *CellMLWrapper* interacts with OpenCMISS by providing the $cell_calculate()$ function. It transfers data to and from the on-chip memory on the FPGA using a DMA controller through the PCIe interconnects. To achieve this, it calls PCIe functions provided by the PCIe driver. Figure 2.5 shows the flow of *CellMLWrapper*. After initialising the PCIe connection, it sets the control signal to the host computer, adds the "known" variable values to a DMA transfer and queues the transfer into the DMA controller. Once the designated amount of data has been added, the selected DMA controller starts performing all the DMA transfers in the queue, and uses either polling or interrupts to check whether a transfer is finished.

Ideally, for convenient control, the size of data to be added into the queue of the DMA controller should be a multiple of the number of cells required to fill the pipeline. The data size, however, will also depend on the input size of the CellML model.

2.4.3 *FPGA Design*

The hardware infrastructure is shown on the right-hand side of Figure 2.4. The CellML hardware model is connected to the controller and the on-chip memory through the memory mapped I/O interfaces. The controller is also

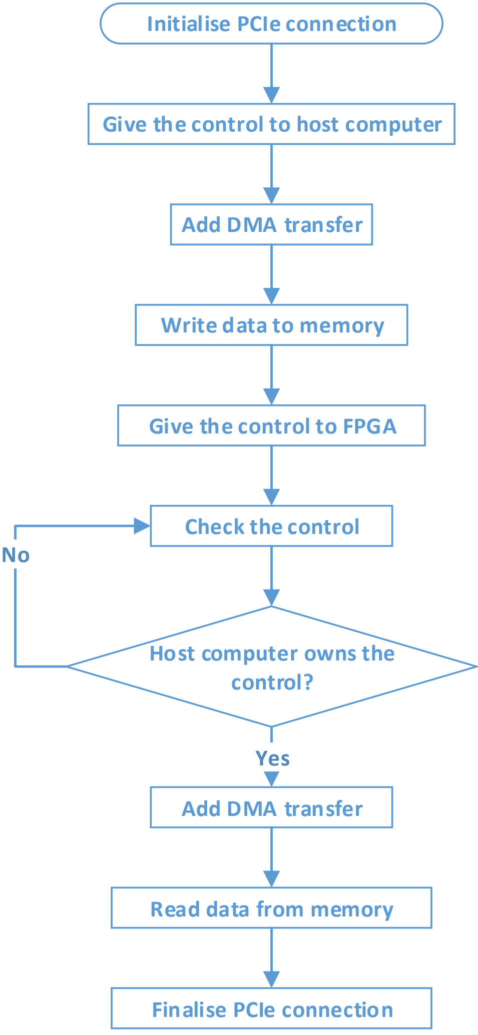


Figure 2.5: Flow of *CellMLWrapper*.

mapped to the on-chip memory to share the control signal with the host computer. The PCIe IP core is interfaced with the physical PCIe interconnects and transfers data to or from the on-chip memory through a DMA controller. Based on Altera's recommended method [4], two types of DMA controllers, the ordinary DMA controller and the SGDMA (Scatter-Gather Direct Memory Access) controller, are used and are exchangeable in the design. For large data that requires multiple transfers, the SGDMA controller is used instead of the regular DMA controller. This is because, for the SGDMA, multiple transfers are handled by the hardware itself instead of by intervention from a host. This typically reduces the downtime between transfers to a single clock cycle.

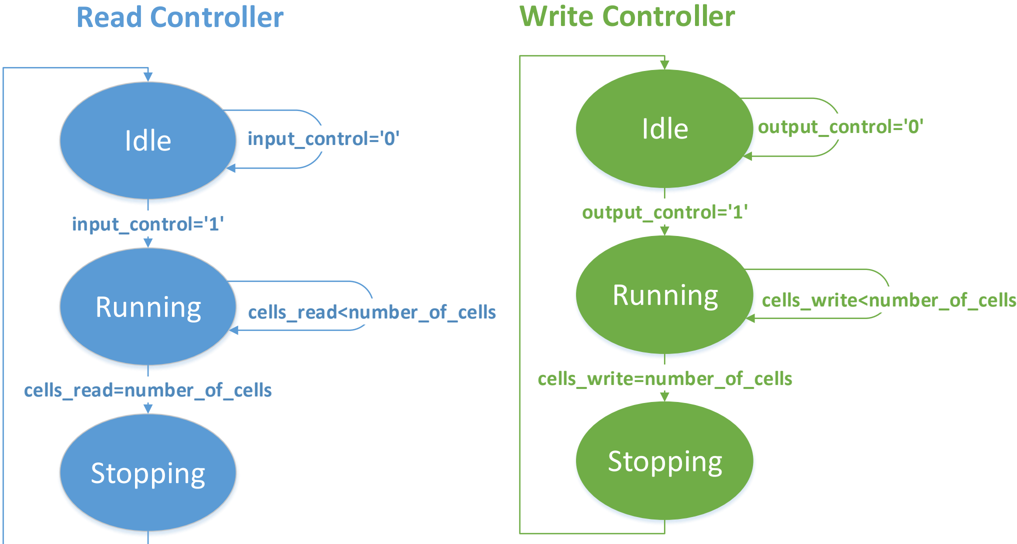


Figure 2.6: State machines for read and write controllers.

Once the data transfer from the host computer to the on-chip memory has finished, control is given to the controller by the host computer. The controller is connected with the CellML hardware model through the memory mapped interface. Once it receives the control, it immediately sets the configurations such as input data address, output data address and passes the GO signal to the status register of the CellML hardware model.

The CellML hardware model uses two memory mapped I/O interfaces to read and write data from and to the on-chip memory respectively. Within the model, two state machines are used to control the data transfer. Both state machines comprise three states: *Idle*, *Running* and *Stopping* as shown in Figure 2.6.

For the read control state machine:

- **Idle:** This is the reset state. The state machine waits for the *input_control* signal from the controller to be active. Upon moving to the Running state the read address is loaded in from the controller;

- **Running:** Data is read from the on-chip memory. The read address is incremented and the number of cells read is tracked. Once the specified number of cells have been read the state machine moves to the Stopping state.

- **Stopping:** This state tells *CellMLCore* that the inputs from the on-chip memory have all been loaded. From now on, the iterative computation should use the previous iterations output as the inputs. It then moves back to the Idle state.

For the write control state machine:

- **Idle:** This is the reset state. The state machine waits for the *output_control* signal from *CellMLCore* to be active. Upon moving to the Running state the write address is loaded in from the controller;
- **Running:** Data is written into the on-chip memory. The write address is incremented and the number of cells written is tracked. Once the specified number of cells have been written the state machine moves to the Stopping state.
- **Stopping**: This state tells the controller that the outputs from *CellMLCore* have all been loaded to the on-chip memory and the controller can give the control back to the host computer for the DMA transfers.

2.5 EXPERIMENTS

The experiments section is organized as follows. The hardware used in the experiments is defined and the tests conducted (Section 2.5.1). Next, the synthesis results are presented (Section 2.5.2) and the performance computed as speedup over the single core CPU only implementation. Lastly, the results are discussed and the potential speedup for a variety of the CellML models is estimated.

2.5.1 *Experimental Setup*

The experimentation was hosted on an Altera Nios II Qsys RC environment, chosen for its robustness and backed by a powerful tool chain facilitating the rapid exploration of both hardware and software. The Nios II processor acts

as the controlling system on the FPGA which is represented as the Controller in Figure 2.4.

In the configuration the clock frequency was set at 100 MHz for the entire system and tests were performed on the Terasic DE4 development board featuring an Altera Stratix IV EP4SGX230 FPGA. The DE4 board is connected with a 3.2 GHz Intel Core i5 3470 CPU and 16GB of RAM on the host machine through a PCIe x8 interface. The host machine is running Ubuntu 12.10 and is also used for the CPU only comparison tests.

The CellML hardware model implemented was based on the Hodgkin-Huxley model described in Section 2.3.2. Two variations of the model were created with and without the hard DSP blocks using generated floating point operations from FloPoCo. The designs are compiled through Altera's Quartus II v12.1 to synthesize, place and route 1 - 4 components on the Stratix IV EP4SGX230 device. Table 2.1 shows the number of equations and individual floating point operations used for each component. From Quartus II, we have extracted the total logic, registers and DSP blocks used to implement each design and the maximum operating frequency (f_{max}) of the final placed and routed circuit.

To evaluate both variations of the CellML hardware model, 12 test cases are used which varied the number of components (1 - 4) and the number of iterations (10, 100 and 1000). These test cases were written in C. The data inputs for these test cases are randomly generated single-precision floating-point values. The test cases are executed for both the CellML hardware variations (with and without DSP blocks) and the CellML software model. The total time taken for the data transfer and cell computation are recorded. The performance results are presented as the speedup compared to the pure software model. The CellML software model is compiled with gcc 4.7.2 with -O3 level optimization.

2.5.2 *Synthesis Results*

The results of the individual floating point operations are shown in Table 2.2. These results are generated using FloPoCo. According to the results, $fmul$ with

Operations	Latency	ALUTs	Reg	DSPs	f_{max}(MHz)
fadd	12	269	622	0	531
fdiv	17	1171	1407	0	313
fmul (with DSPs)	4	73	219	4	851
fmul (no DSPs)	5	893	524	0	389
fexp (with DSPs)	17	436	854	2	198
fexp (no DSPs)	17	819	978	0	252

Table 2.2: Synthesis results of the floating point operations for Altera EP4SGX230 device.

Comps.		Latency	ALUTs	Reg	DSPs	Area%	f_{max}
1	D	98	5.7k	9.2k	24	6%	194
	-	98	9.6k	10.6k	0	9%	237
2	D	98	9.7k	15.2k	48	10%	192
	-	98	16,8k	17.9k	0	15%	242
3	D	98	13.2k	20.8k	76	13%	185
	-	98	24.1k	25.4k	0	21%	229
4	D	98	17.0k	23.4k	120	16%	189
	-	98	35.7k	30.7k	0	29%	223

Table 2.3: Synthesis results of Hodgkin-Huxley CellML hardware model with one to four components using Altera EP4SGX230 device (D: with DSP blocks).

hard DSP blocks uses the fewest logic resources and it represents the simplest placement and routing problem. Therefore, it achieves the highest operating frequency. On the other hand, $fdiv$ and $fexp$ use more logic resources and are more complex to place and route and hence require a lower operating frequency.

Table 2.3 presents the results of the CellML hardware models as discussed in Section 2.3 with 1 to 4 components chosen from the Hodgkin-Huxley model. The results shows that the implementations using the DSP blocks generally are more efficient in terms of area, but use a lower operating frequency. This is an odd result since fewer logic resources represents simpler placement and routing and hence should achieve higher operating frequency. However, according to Table 2.2, $fexp$ with DSP blocks achieves the lowest operating frequency of 198 MHz, which restricts and lowers the overall operating frequency.

Comps.		Latency	ALUTs	Reg	DSPs	Area%	f_{max}
0	-	-	9.2k	9.4k	4	8%	99
1	D	98	13.5k	14.5k	28	12%	80
	-	98	17.2k	16.3k	4	14%	111
2	D	98	17.8k	19.6k	52	15%	98
	-	98	24.5k	22.5k	4	20%	124
3	D	98	21.7k	24.0k	80	18%	116
	-	98	32.0k	29.5k	4	26%	100
4	D	98	25.1k	27.9k	124	21%	102
	-	98	42.9k	37.6k	4	34%	104

Table 2.4: Synthesis results of the complete hardware system for Altera EP4SGX230 device (D: with DSP blocks).

The synthesis results for the overall system are discussed in Section 2.4 and are presented in Table 2.4. The results shows that the maximum operating frequencies are lower than the CellML hardware model shown in Table 2.3. This is because other modules such as the IP Compiler of PCIe, Nios II processor or DMA controllers use a more complex design and lower the operating frequency.

2.5.3 *Performance comparison*

The performance results of the overall system containing the CellML hardware model with 1 to 4 components are illustrated in Figure 2.7. For the hardware model, the speedup is measured by the total time taken for the data transfer between host machine and the FPGA device plus the cell computation time within the FPGA is divided by the total time take for the CPU computation.

2.5.4 *Discussion*

From the performance results, the CellML hardware model has consistently performed as fast or faster than the pure software model. The hardware implementation has attained the speedup of up to 4.2x. This is a significant yet not a dramatic speedup, since the Hodgkin-Huxley model requires relatively

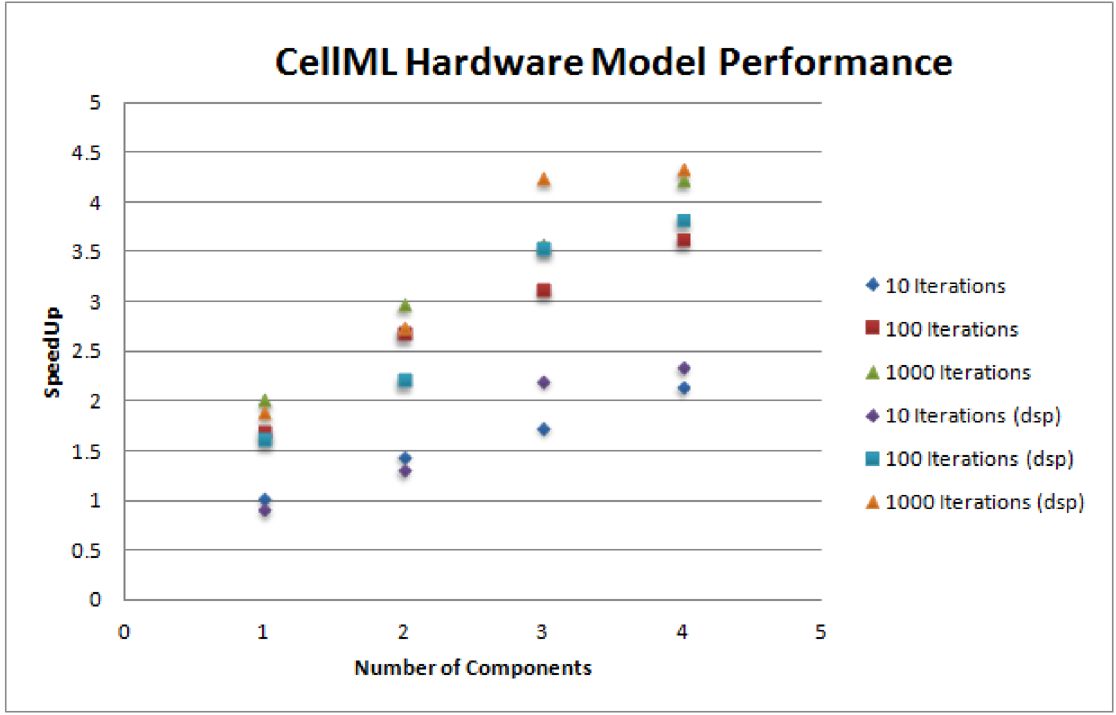


Figure 2.7: Performance results of CellML hardware model computation.

few floating point operations compared to other CellML models. From the performance results, the speedup is nearly linear against number of the components. Thus, within the resource capacity, larger models show more benefit with hardware acceleration by attaining a greater speedup compared to pure software models.

The synthesis results show that the hardware implementations using the hard DSP blocks are more resource and area efficient than implementations with pure logic elements. This means that more models can be fit into one FPGA. However, the number of DSP blocks within one device is limited and, so whether to use implementations with DSP blocks or not is not always predetermined.

The current system implementation still has room for improvement and optimization strategies include: increasing the operating frequency, parallelism with multiple CellML hardware models and overlapping communication with computation.

2.6 CONCLUSIONS

This chapter proposes an approach for the hardware acceleration of biomedical model calculations. Based on a CellML description of ODEs, hardware implementations of these ODEs are generated. A software/hardware co-design is developed to integrate the CellML hardware model with the software elements. The design is general and flexible and can be used for all kinds of CellML models.

Using the Hodgkin-Huxley CellML model as a case study, an application performance improvement of a factor of 4x has been achieved compared to the pure-software CellML model. According to the scalability shown in the speedup results, there is potential for further performance improvement with more complex CellML models. In terms of the usability and feasibility of the design, the focus is on using this general design in an automatic, rather than manual way.

3 ODE-BASED DOMAIN-SPECIFIC SYNTHESIS TOOL

This chapter continues on from the hardware accelerator module designed in Chapter 2, and presents a domain-specific high-level synthesis tool called ODoST to automatically generate the hardware accelerator module. The contents of this chapter have been submitted for publication as a research article, *ODoST: Automatic Hardware Acceleration for Biomedical Model Integration*, to ACM Transactions on Reconfigurable Technology and Systems.

Contributions of this chapter are: (i) improvement of the hardware accelerator module to be adopted in the auto generation framework. In the manual module, a Nios II soft processor is used to handle the data flow on the FPGA, while in this work, the processor is removed and replaced by a dedicated controller, (ii) design, development and test of the domain-specific high-level synthesis tool to generate the hardware accelerator modules from the high-level description of biomedical models.

Experimental results of this chapter show that (i) FPGA implementations have a significant performance advantage compared to single CPU and multicore implementations and a compatible processing speed against the GPU implementation, (ii) FPGAs deliver much higher energy efficiency compared to CPUs and GPUs.

CHAPTER ABSTRACT

Numerical integration of biomedical models is employed by researchers to simulate dynamic biomedical systems. Models are often described mathematically by Ordinary Differential Equations (ODEs) and their integration is often one of the most computationally intensive parts of biomedical simulations. With high inherent parallelism, hardware acceleration based on FPGAs has great potential to increase the computational performance of the model integration, whilst being very power efficient. However, with variant biomedical models, manual hardware implementation is too complex and time consuming. The advantages of FPGA designs can only be realised if there is a general solution which can automatically convert these biomedical models into hardware description languages. In this chapter a domain specific high-level synthesis tool called ODoST is proposed that automatically generates a FPGA-based hardware accelerator module (HAM) from the high-level description. The investigation also includes a general hardware architecture for this application domain. The generated HAMs on real hardware are evaluated based on their resource usage, processing speed and power consumption. The HAMs are compared with single threaded and multicore CPUs with/without SSE optimisation and a graphics card. The results show that FPGA implementations are faster than all the CPU solutions for complex models and perform similarly to an auto-generated GPU solution, whilst the FPGA implementations are significantly more power efficient than the CPU and GPU solutions.

3.1 INTRODUCTION

Biomedical modelling often uses numerical integration of biomedical models to simulate dynamic biomedical systems in order for researchers to understand different physiological functions. Recently, the number of degrees-of-freedoms (DOFs) used for mathematical models has increased rapidly due to the increasing complexity of models and an increased accuracy requirement. To simulate a model with a fine mesh size and running for millions of time

steps includes a significant amount of computation which can be very time consuming, even with today's fastest CPUs [94].

The models used in such simulations, by their nature, are regular, relatively small but performance-critical and highly data parallel. As such, special purpose hardware, in particular FPGAs with a large amount of fine-grained parallelism, have promise for accelerating these models. Integrating the use of FPGAs into the parallel processing of the simulations has the potential to lead to higher performance with reduced energy consumption.

However, when compared to technologies such as multicore processors and General Purpose Graphics Processing Units (GPGPUs), FPGAs have not been widely adopted for accelerating applications. There are two major reasons for this. First, developing a FPGA hardware design for a given application is much more complex, time consuming and error prone than programming general purpose processors. Second, it is hard to integrate general purpose processors in parallel computing systems with FPGAs (referred to as hybrid systems). Although previous studies [23, 57, 35] show that GPUs generally outperform the FPGA architectures for streaming applications and enjoy a higher floating-point performance, there is still a growing interest in research into using FPGAs as an accelerator tool due to its unrivaled flexibility, technology trends and low power consumption.

In this thesis, a hardware accelerator with a FPGA-CPU heterogeneous architecture is proposed for models described by CellML [34], an open standard mark-up language based on XML. CellML is used by a variety of tools to describe biomedical models, e.g., OpenCMISS, a general purpose computational library for solving field based equations with an emphasis on biomedical applications [24]. To reduce the effort in implementing accelerators from models, an ODE-based Domain-specific Synthesis Tool (ODoST) is designed and implemented to automatically create the accelerator framework. In this chapter, the performance and synthesis results of the model accelerators are considered for three models, each with increasing complexity. The performance of one of the FPGA models is also compared with a GPU implementation of that model obtained from a previous study.

The chapter is organized as follows. Related work is discussed in Section 3.2. In Section 3.3, a typical biomedical hardware accelerator module from a model described by CellML is analysed and described. The implementation of the ODoST is described in Section 3.4. In Section 3.5, the experimental results are evaluated. The chapter is concluded in Section 2.6.

3.2 RELATED WORK

Biomedical models and simulations are used to understand normal and abnormal functions of animals and humans. Mathematical models based on Ordinary Differential Equations (ODEs) are not only used in the biomedical field, but also extensively in other physical systems such as weather prediction, mobile computing and thermal analysis. There are a number of modelling languages that have been developed for storing and interchanging biological mathematical models. For example, the Mathematical Modelling Language (MML) [72], the Systems Biology Markup Language (SBML) [54] and CellML [34]. Simulation tools have also been developed to simulate models written in these modelling languages. Some of the tools are focused on validation and visualisation, such as JSim [19], OpenCell [46], Virtual Cell [83], Matlab [65], LabView [70] and Mathematica [98]. Other tools emphasise large scale and continuum simulations that require high performance computation. For example, the Cancer Heart and Soft Tissue Environment (CHASTE) [79] and OpenCMISS [24]. This thesis has designed and built an accelerator model based on CellML and is intended to be used by OpenCMISS and other simulation packages.

Many case studies have been proposed and conducted using FPGAs to accelerate the simulation of biomedical or mathematical models. Yoshimi et al. [99]'s accelerator of a fine-grained biochemical simulation achieved a 100x speedup compared to a single processor during that time. Osana et al. [76] developed a solver-based tool, ReCSiP, for biochemical simulation using Xilinx's XC2VP70-5 and reported a 50 to 80 times speedup compared to Intel's Pentium 4 processor. Thomas and Amano [93] proposed a pipelined architecture for a stochastic simulation of chemical systems and reported that their architecture

was 30-100 times faster compared to a pure software simulator. de Pimentel and Tirat-Gefen [39] estimated a real time simulation of a heart-lung system model which was expected to be 90 times faster than a PC. However, their evaluation was calculated theoretically based on performance of the multiplier on the device rather than a real implementation. Chen et al. [28] implemented a Runge-Kutta ODE solver using FPGAs and Simulink that resulted in a 100x speedup compared to a 2.2 GHz desktop. Most of these studies used a manual design and implementation to develop a specialised accelerator model. Manual design is impractical in biomedical/mathematical simulations since it is time consuming and requires hardware development skills, often not found in those with a biological background.

Apart from studies investigating acceleration of biomedical simulations with FPGAs, researchers and software developers have favoured increasing the performance of complex biomedical simulations using multicore and GPUs as they require less programming. For instance, a 768-core SGI Altix 4700 shared memory computing system simulated five milliseconds of a two billion equation heart activation problem in two hours [97]. Okuyama et al. [75] described two acceleration methods for their physiological simulator, Flint, and gained 37x and 55x speedup compared to single threaded CPU. Shubhranshu [90] established the superiority and cost effectiveness of a GPU based solution for a CellML model simulation through a comparative analysis. His results are used as a performance comparison with our results in Section 3.5. While multicore processors, distributed systems and GPUs are all capable of doing parallel computation in a time efficient way, they consume much more power thanFPGAs. Chen and Singh [27] compared the board power for an Intel Xeon W3690, NVidia Tesla C2075 and Altera Stratix IV 530 and concluded that a FPGA used about one fifth of power when compared to a multicore CPU and one tenth of the power when compared to a GPU. Kestur et al. [59] tested BLAS on a FPGA, CPU and GPU and the results showed that FPGAs offer comparable performance as well as 2.7 to 293 times better energy efficiency. Betkaoui et al. [22] compared the energy efficiency for high productivity computing on FPGAs

and GPUs against CPUs and obtained 3.7x efficiency with FPGAs and 2x efficiency with GPUs against single threaded CPU implementations.

Due to the high requirement of development efforts and skills, many tools have been developed for implementing applications on FPGAs through High Level Synthesis (HLS) such as SPARK [49], DRFM [30], GAUT [31], LegUp [26] and polyAcc [81]. These tools are used to automatically generate hardware circuits from a high level representation, e.g. C, Matlab, Java, etc. In this thesis, a domain specific synthesis tool called ODoST was designed and implemented. This tool focuses on ODE-based mathematical models and aims to create the complete datapath of a given model including the data communication and software interfacing.

3.3 BIOMEDICAL HARDWARE ACCELERATOR MODULE

3.3.1 *A Motivating Example*

The motivation for this study came from an estimation of an electrical activation problem in the human heart. The approximate volume of a human heart is $8.19 \times 10^5\,\mathrm{mm}^3$. To discretise the volume of the ventricles (about half of the heart) into grids with $100\,\mu\mathrm{m}$ spacing would require 4.23×10^8 grid points. At each grid point a system of ODEs needs to be solved for each time instance. If a model with 30 ODEs is used and assuming that 100 FLOPS are required for one ODE evaluation, to simulate the model at each time instance would require 1.27×10^{12} FLOPS. With a 1 ms time step, to simulate one minute of real activation would require 7.62×10^{16} FLOPS. If, for example, a processor could compute 20 GFLOPs per core [78], a single core would require approximately 44 days for a simulation.

3.3.2 *Biomedical Model Overview*

Biomedical models are often represented by a set of ODEs describing time varying variables and parameters. For the purpose of analysis and experi-

ments, four models from the CellML model repository which contains 300+ models are selected. Each CellML model is component based and the components are represented by one or more mathematical equations. In this section, the Hodgkin-Huxley model of a giant squid axon[1] is considered. The model consists of 14 mathematical equations with 12 inputs and 14 outputs. For the purpose of this thesis, neither the underlining biophysical concepts nor the complete model will be explained here, but instead, the "*sodium_channel_m_gate*" component is extracted which is a good representive example of an ODE computation from the model. The equations for this component are:

$$alpha_m = \frac{0.1 \times (V + 25)}{e^{\frac{V+25}{10}} - 1} \tag{3.1}$$

$$beta_m = 4 \times e^{\frac{V}{18}} \tag{3.2}$$

$$\frac{dm}{dt} = alpha_m \times (1 - m) - (beta_m \times m) \tag{3.3}$$

$alpha_m$ and $beta_m$ are the rate constants and are intermediate variables. V and m are state variables and $\frac{dm}{dt}$ is the rate of change for m at time t. The rate of change for V is computed by another component in the model. For a single time step of model integration, the values of the intermediate variables are computed, first based on the state variables (and parameters if they are required). The rate of change for the state variable is then computed which is dependent on the intermediate variables. Once the value of rate is available, a numerical integration method is used to approximate the state value at next time step. A variety of such numerical integration algorithms exists and, in this thesis, a forward Euler's method is used. The computation of the state variable m at time $t + \triangle t$ is represented in Eq. (3.4).

$$m_{t+\triangle t} = m_t + \triangle t \times \frac{dm}{dt} \tag{3.4}$$

1 http://models.cellml.org/exposure/5d116522c3b43ccaeb87a1ed10139016

In order to achieve accurate results, the above process is performed and integrated in fine time steps. For example, to integrate 1 ms of the model at one grid point, the time interval is divided into 1000 time steps and each time step takes 1 µs. At each time step for the "*sodium_channel_m_gate*", the computations of Eqs. (3.1 - 3.3) are performed to obtain the rates of change and then numerical integration is performed to find the new states after 1 µs. The new state variables are then passed to the next step for the next time integration and so on. During this integration process, each grid point is integrated individually and independent of other grid points.

The computational workflow is described in Figure 3.1. At the initialisation phase, a predefined model is loaded. Analysis data (state variables and parameters) are initialised and passed to the model integration to obtain the state variables and intermediate variables after one macro time step. Once finished, the simulation time is incremented to the next macro time step and the new state variables are passed to the spatial solver for numerical techniques such as finite element analysis. On completion, the simulator updates the model and passes new state variables and parameters to the model integrator for the next macro time step integration.

The process of model integration as described is illustrated in the zoomed in box in Figure 3.1. Δt represents the macro time step of 1 ms and $\Delta t'$ represents the micro time step of 1 µs. The algorithm requires spatial solving with every macro time step. The overall problem then becomes a huge sequential bottleneck since improving modelling accuracy by increasing the temporal resolution results in a long overall computation time. To solve this problem and retain reasonable accuracy, one macro time step is divided into a number of micro time steps (e.g., 1000), spatial solving is performed every macro time step and numerical integration is performed every micro time step. While each individual model integration is sequential, they are all independent of each other on a spatial level and hence massive parallelism supported by FPGAs can be applied to the model integration over many, many grid points.

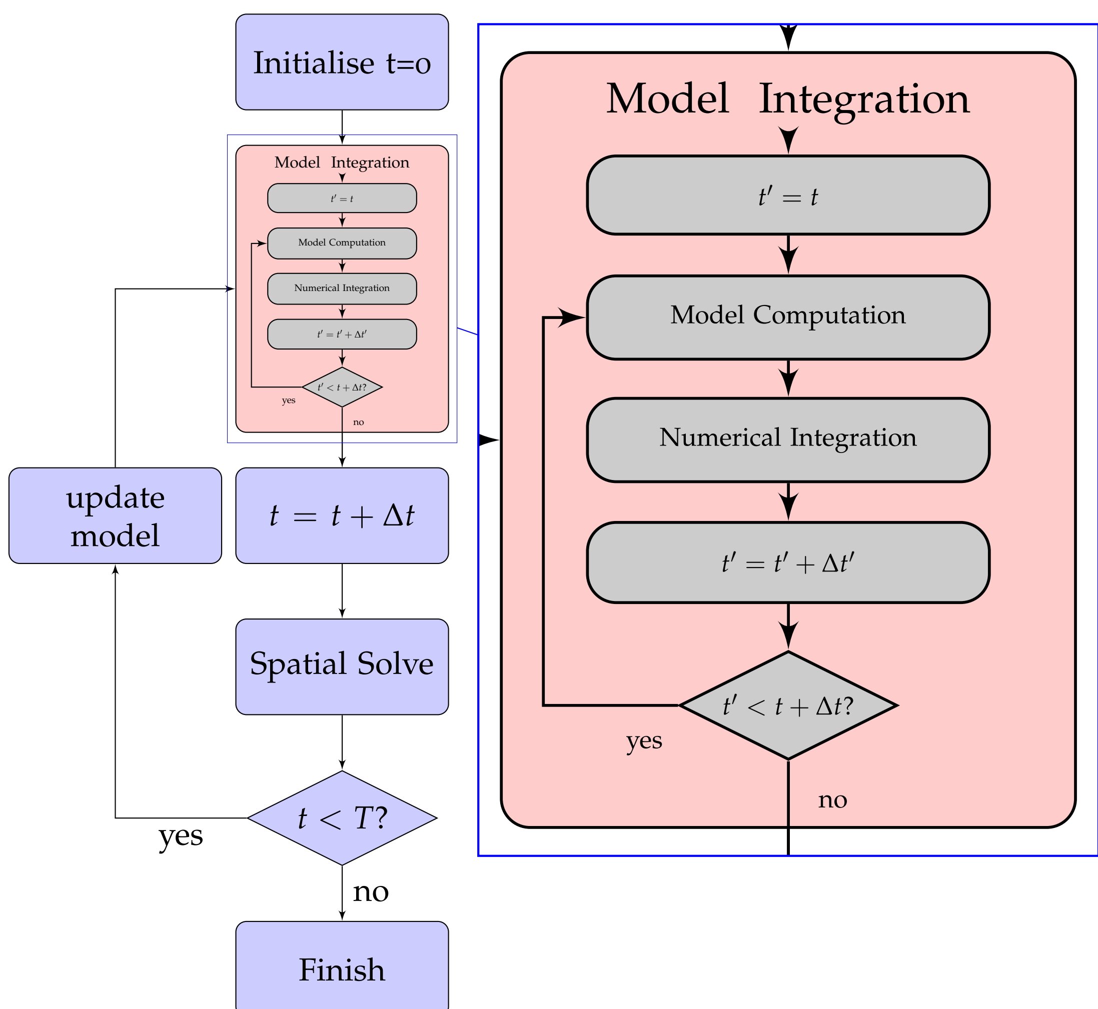


Figure 3.1: General flow of model computation.

Function	Latency	Logic	Registers	DSPs	f_{max}(MHz)
FPAdd	12	269	622	-	523
FPDiv	17	1188	1407	-	308
FPMult	4	73	219	4	835
FPExp	17	436	878	2	195
FPLog	21	831	1210	18	175
FPPow	45	1808	3307	31	177

Table 3.1: FloPoCo resource use and performance for Stratix IV device.

3.3.3 *Pipelined Floating Point Operations*

As mentioned, biomedical models often contain a set of ODEs and hence arithmetic operations are the key components for their equivalent hardware accelerator modules. Frequency and area are the two main factors that measure the quality of an arithmetic operation on FPGAs. As each grid point computation is independent, they can be integrated in parallel. In addition the computational logic in the hardware accelerator model can use a pipelined architecture for increased performance.

During the computation, the number of pipeline stages is negligible compared to the number of datasets passed into the pipeline data path and hence all pipeline stages are active most of the time. Therefore, latency in the model is not a relevant criterion and the objective is to generate a circuit with high throughput. In turn, throughput is determined by the number of parallel cell models in the FPGA and the frequency they operate at.

There are numerous floating point cores provided by the vendors of FPGAs or third party floating point platforms. These cores typically exploit the freedom of an FPGA by providing the customisation of variable widths of exponent and mantissa to meet the designers' specifications. They also offer IEEE standard single and double precision cores that are used in the hardware accelerator. In this thesis, FloPoCo [36], a floating point core generator, is used to create the pipelined arithmetic operators. This tool provides great flexibility for generating floating point operations in VHDL from C++ code. In order to generate a floating point core, FloPoCo receives an input of the core operation

features, such as target frequency, use of a pipeline, single or double precision, enable or disable the Digital Signal Processing (DSP) blocks and the FPGA manufacturer and model. The output is a synthesizable VHDL file with the required input features. With this tool it is possible to change from a single precision to double precision pipelined floating point core by only changing the core generator parameters and thus saving rework. Table 3.1 displays the resource usages and performance for the floating point cores on Stratix IV Device generated by FloPoCo. In the table, "Logic" refers to the combinational ALUTs (Adaptive Look-up Tables), "Registers" refers to the dedicated logic registers and "DSP" corresponds to the 18-bit DSP blocks embedded within the device.

The ASAP (As Soon As Possible) clock cycle scheduling algorithm is adopted as shown in Figure 3.2. It presents the pipelined datapath flow for Eqs. (3.1 - 3.3) discussed in Section 3.3.2. In each diagram, the horizontal axis is the time in unit of cycles. One cycle is also one pipeline stage since it is fully pipelined. The vertical axis represents the data sets that enter into the pipelines. One data set is needed for one cell computation. Only the first three data sets are shown for illustration but, in practice, there are many more. The offset between two consequent data sets is one pipeline stage, which means data sets are pushed into the pipeline every cycle until it is completely filled. The floating point operations are symbolically represented in the diagrams and their widths represent number of cycles required to complete the operation. Each equation is implemented separately with its own data set, datapath and output. Some equations may contain sub branches. For example, in Eq. (3.3), $alpha_m \times (1 - m)$ and $beta_m \times m$ can be executed in parallel. Figure 3.2e illustrates the complete sodium_channel_m_gate integration. It connects the datapaths from individual equations into a long datapath. In order to balance the pipeline, register delays are inserted. For example, as $beta_m$ finishes 24 cycles earlier than $alpha_m$, a 24-stage register path is inserted into the datapath in order to balance the pipeline. Therefore, a pipeline system typically requires many registers which will eventually become a bottleneck. To solve this problem, RAM-based shift registers are used instead. According to our results, this type of shift register

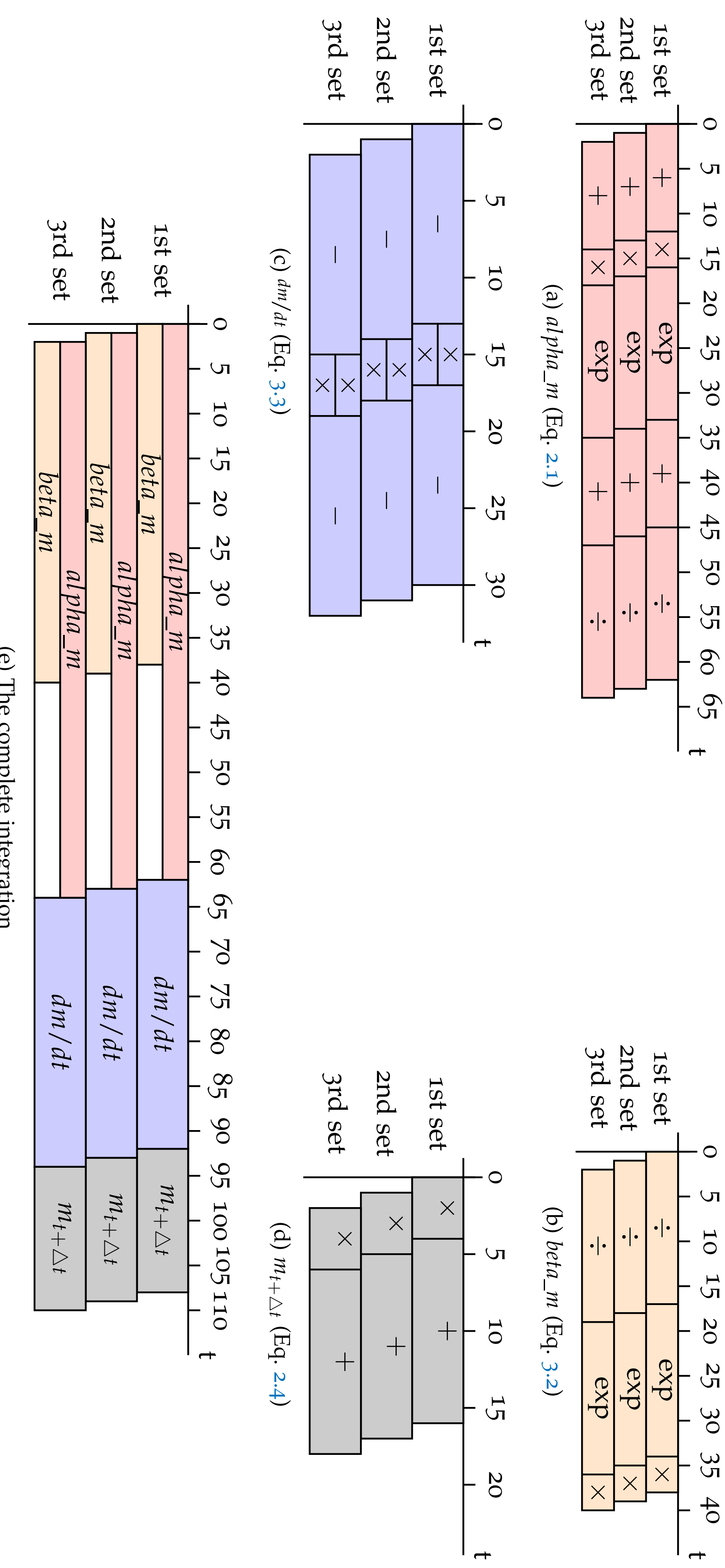


Figure 3.2: Pipeline scheduling for *sodium_channel_m_gate* integration.

saves a significant amount of register consumption and has no negative impact on performance.

3.3.4 *Hardware Accelerator Module Architecture*

The system architecture of the Hardware Accelerator Module (HAM) is shown in Figure 3.3. It is composed of a host computer and a FPGA board connected through the PCIe interface. The arrows indicate the data communication throughout the system. As described in Figure 3.1, a biomedical simulator such as OpenCMISS initialises the variables and parameters and interacts with the software module by a *model_integrate* function call. The software module is used as a bridge application and interacts with the FPGA by sending and receiving data through the PCIe interconnects.

On the FPGA side, there is a PCIe IP core that interacts with the PCIe connector and maps to the on-chip memory directly for the control signals and through the DMA (Direct Memory Access) controller for the data transfer. The received data from the host computer is written into on-chip memory through the DMA controller. A controller is used to send/receive signals to/from the host computer and interact with the CellML hardware model to control the data transfer.

3.3.4.1 *Software Module*

The software module interacts with the simulator by providing the *model_integrate* function call. It partitions data from the simulator into chunks and transfers data to and from the FPGA chunk by chunk through the PCIe interconnects and uses a DMA (Direct Memory Access) controller on the FPGA to access its on-chip memory. To achieve this, it calls PCIe functions provided by the PCIe driver. Figure 3.4 shows the flow of the software module. It first initialises the PCIe connection and prepares data passed in by the simulator to the FPGA in a favourable data format by dividing the data into chunks for processing. Afterwards, it creates a control signal and grants the control to the host computer.

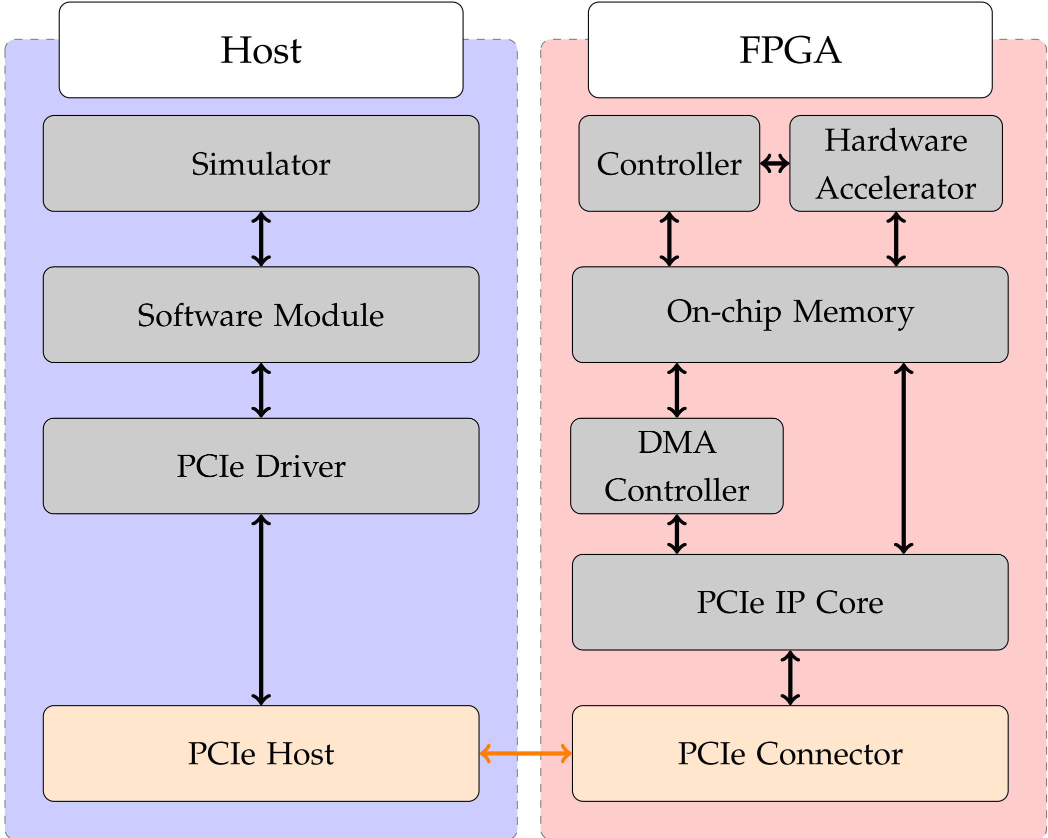


Figure 3.3: Hardware accelerator module system architecture.

The software module then allocates a DMA transfer and queues the transfer into the DMA controller. Once the designated amount of data has been added, the selected DMA controller starts performing all the DMA transfers in the queue, and uses either polling or interrupts to check whether a transfer is finished. Once all the data is written to the FPGA, the host passes the control to the FPGA board for accelerator processing and waits until it finishes. Once the host re-obtains control, the software module reads the results from the on-chip memory of the FPGA through the DMA controller. Afterwards, it prepares the data ready for simulator use and passes the next chunk for FPGA processing.

3.3.4.2 *Data Control*

The hardware infrastructure is shown on the right-hand side of Figure 3.3. The hardware accelerator is interfaced with the on-chip memory through the memory mapped I/O interfaces. The controller is also mapped to the on-chip memory to share the data control signal with the host computer. The PCIe IP

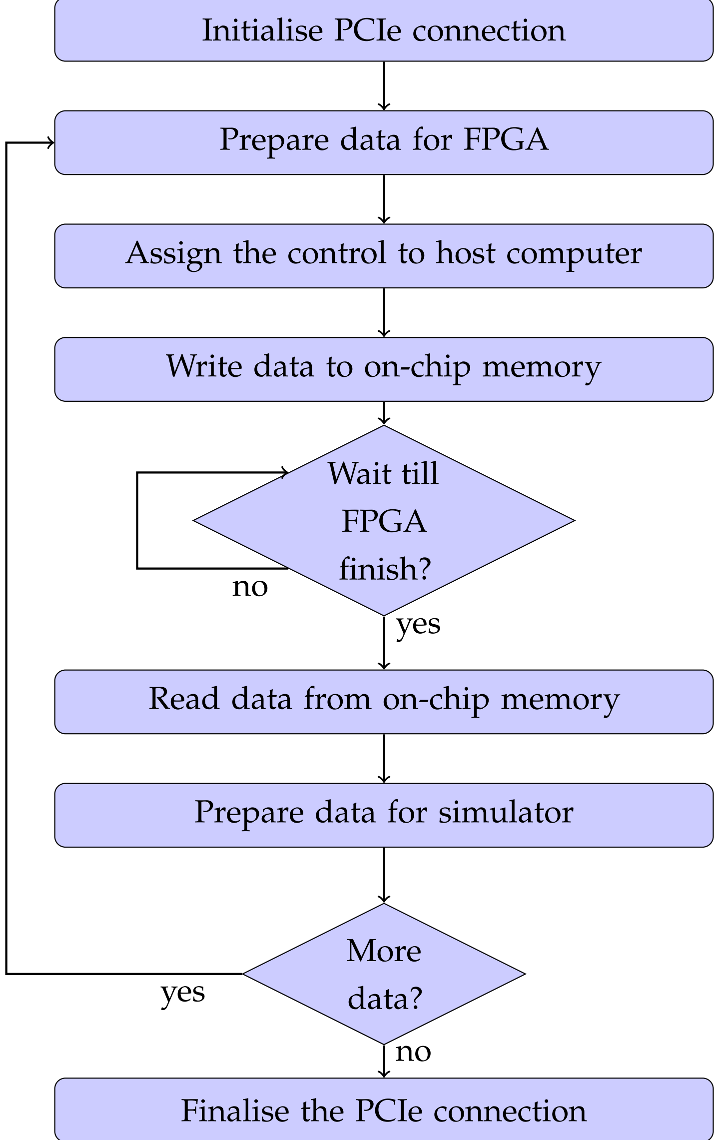


Figure 3.4: Flow of software module.

core is interfaced with the physical PCIe interconnects and transfers data to and from the on-chip memory through a DMA controller on the FPGA.

Once the data transfer from the host computer to the on-chip memory has completed, the data control is given to the hardware controller by the host computer. The controller is connected with the hardware accelerator through port mapping.

The hardware control is performed by a state machine illustrated in Figure 3.5. The state machine comprises of six states:

- **Idle:** This is the state when the host is in control. At this stage, the FPGA continues checking the relevant sector in on-chip memory to obtain the data control signal. When the FPGA board obtains the control from the host, the state machine immediately moves the state to $Read - ToFIFO$.

- **Read-ToFIFO:** To balance the computation in the pipeline datapath, the input data enters a FIFO buffer first. At this state, data is read from the on-chip memory to a FIFO buffer. The read address is incremented and the number of reads is tracked.

- **Read-FromFIFO:** Once all the inputs are in the FIFO buffer, the input data is read from the FIFO buffer into the hardware accelerator cycle by cycle. The model computation starts when the state machine enters $Read - FromFIFO$.

- **Compute:** The **Compute** state starts when all the input data sets are passed into the hardware accelerator. At each micro time step, the state variables computed from the previous micro time step enter the pipeline. The rates are then computed first according to the model and then the numerical integration is performed to compute the new states for the next time step. At the last micro time step, the intermediate and integrated state variables are written into an output FIFO buffer and the state machine moves to the $Write - ToFIFO$ state.

- **Write-ToFIFO:** During this state, the hardware accelerator is doing the final micro time step computation. The output data immediately enters a

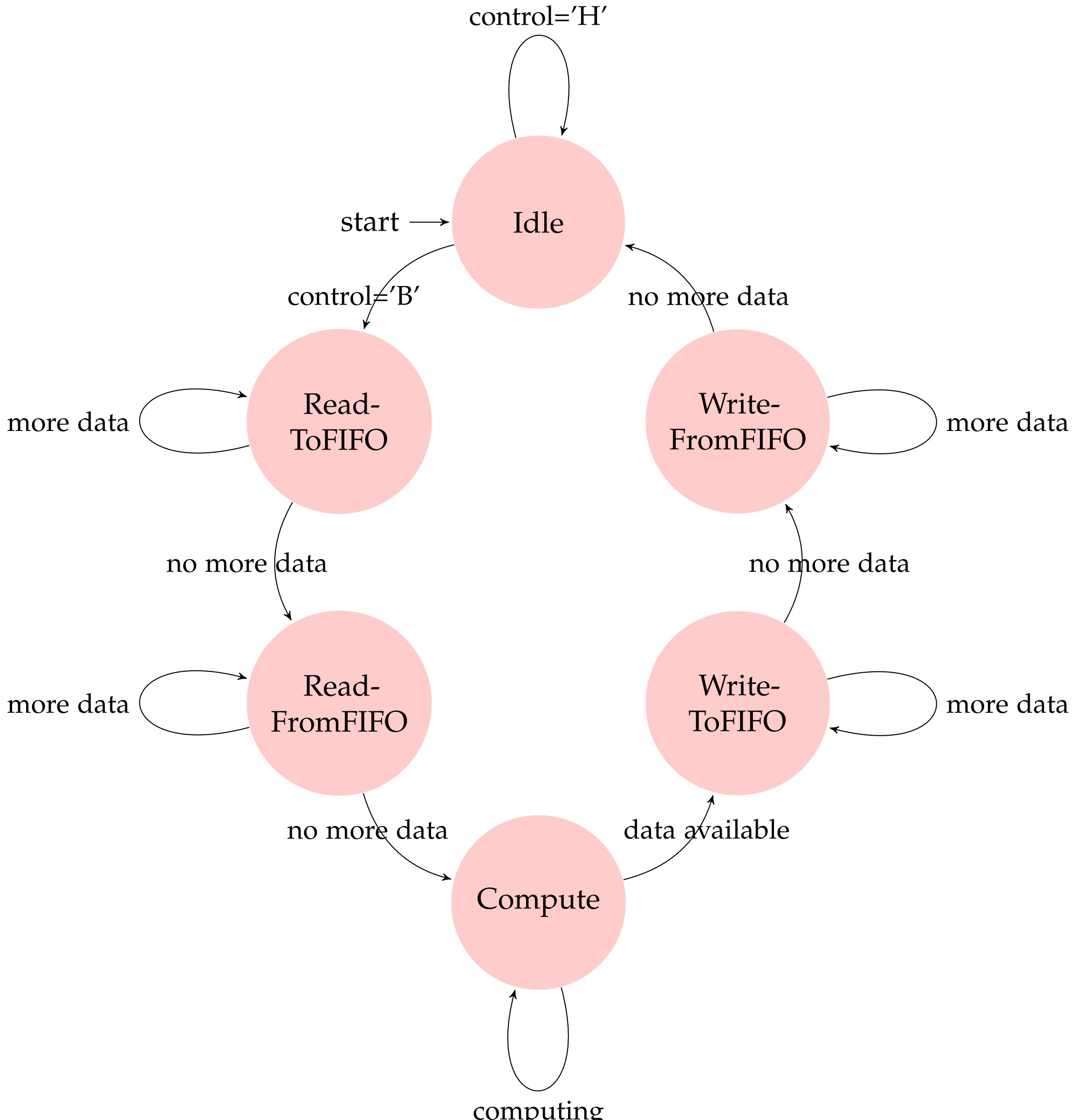


Figure 3.5: State machine for hardware data control.

FIFO buffer. Once the computation finishes, the state machine moves to the $Write - FromFIFO$.

- **Write-FromFIFO:** Data is written into the on-chip memory from the FIFO buffer. Once all the output data is available in the on-chip memory, the FPGA passes control to the host by updating a control signal in the on-chip memory and the state machine moves to the *Idle* state waiting for the next set of input data. The host captures the control signal and activates the DMA to read the output data from the on-chip memory.

3.3.4.3 *Hardware Accelerator*

The hardware accelerator is the core part in the HAM. It employs the pipelined architecture to compute and integrate biomedical models over a certain number of grid points. A simplified structure of the hardware accelerator is shown in Figure 3.6. The controller passes the input variables and control signal to the accelerator.

The main parts of the accelerator are the model computation and model integration steps which both use the pipeline architecture as illustrated in Figure 2.2. They are serially connected to form a long pipeline circuit. A multiplexer is inserted before the circuit and a demultiplexer is inserted after the circuit. The multiplexer is selected by the control signal from the controller to determine whether the data flow into the pipeline circuit is from the on-chip memory or the output of previous time step computation. The control signal is also passed into a shift counter component to generate an output control signal. This signal is used to select the demultiplexer and determine whether the results from the pipeline are outputted to the on-chip memory or passed for the next time step computation.

3.4 ODE-BASED HIGH-LEVEL SYNTHESIS

The previous sections explored a hardware accelerator for biomedical models with a HW/SW co-design structure. However, implementing such an accelerator for a given biomedical model requires enormous effort which might offset the advantages of using a FPGA. This section proposes ODoST, a domain-specific high-level synthesis (HLS) tool, for ODE-based biomedical simulations. The tool is aimed at biomedical scientists and engineers, who often have little knowledge of designing hardware, to create accelerators targeting FPGAs.

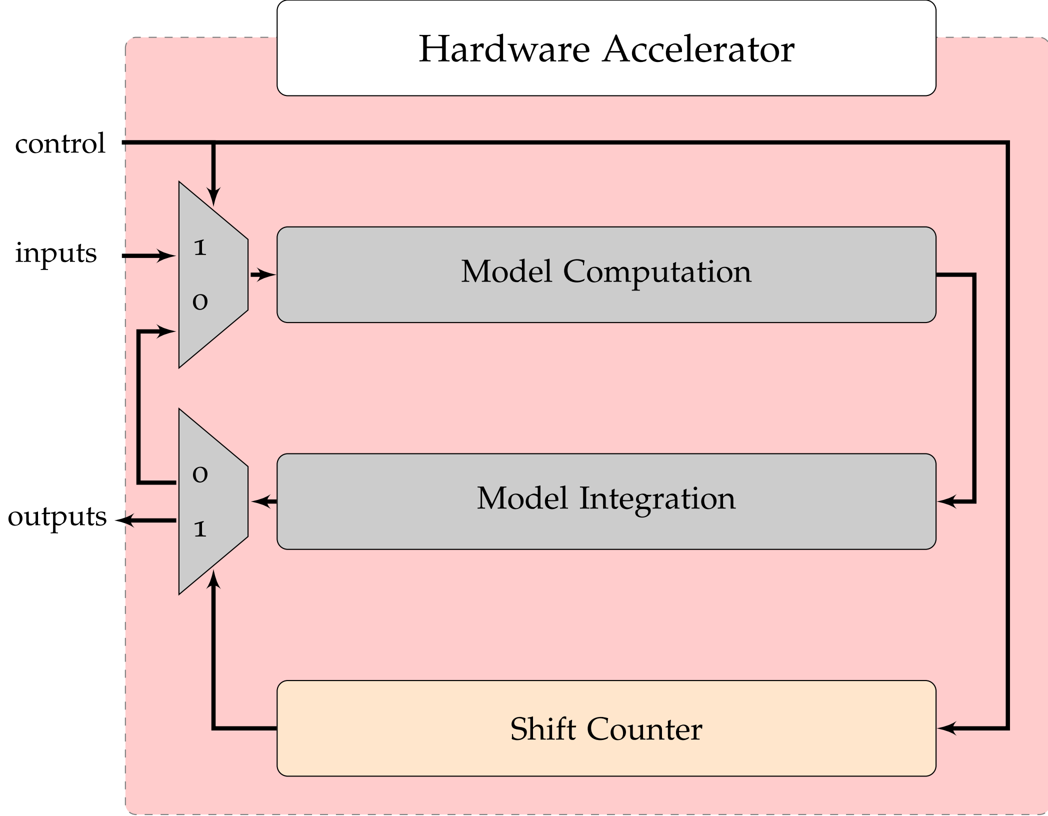


Figure 3.6: Hardware accelerator structure.

3.4.1 *ODoST Overview*

ODoST stands for **O**DE-based **Do**main-specific **S**ynthesis **T**ool. It generates the HAM described in Section 3.3 with both software and hardware modules from an ODE-based biomedical model. An overview of ODoST is shown in Figure 3.7.

The design flow of ODoST is illustrated in Figure 3.8. ODoST contains three phases: the analysis phase, generation phase and system integration phase. In the analysis phase, an input biomedical model is read and analysed. The generation phase uses the analysis results to generate the software module, HDL codes and configuration files for the hardware module. In the system integration phase, the configuration files are used to produces the entire hardware module based on the HDL files generated from the generation phase.

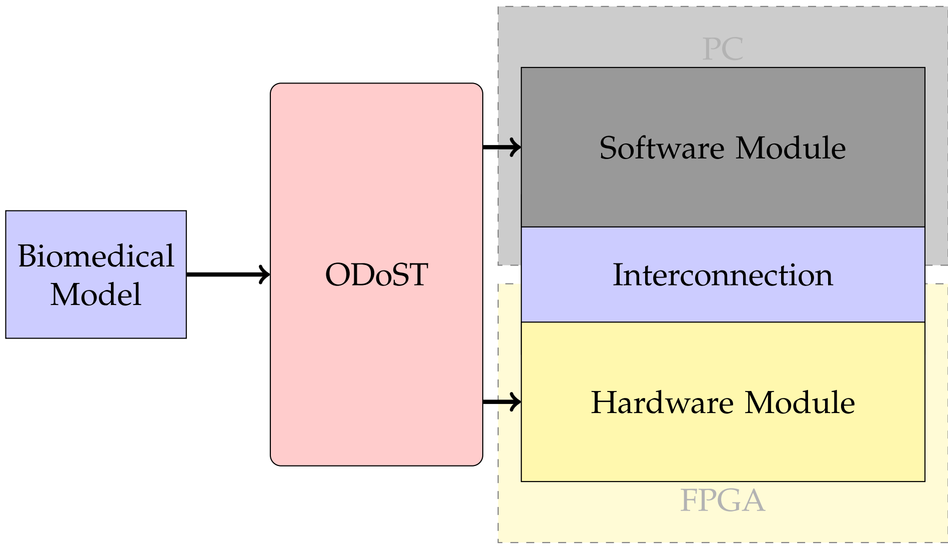


Figure 3.7: Overview of ODoST.

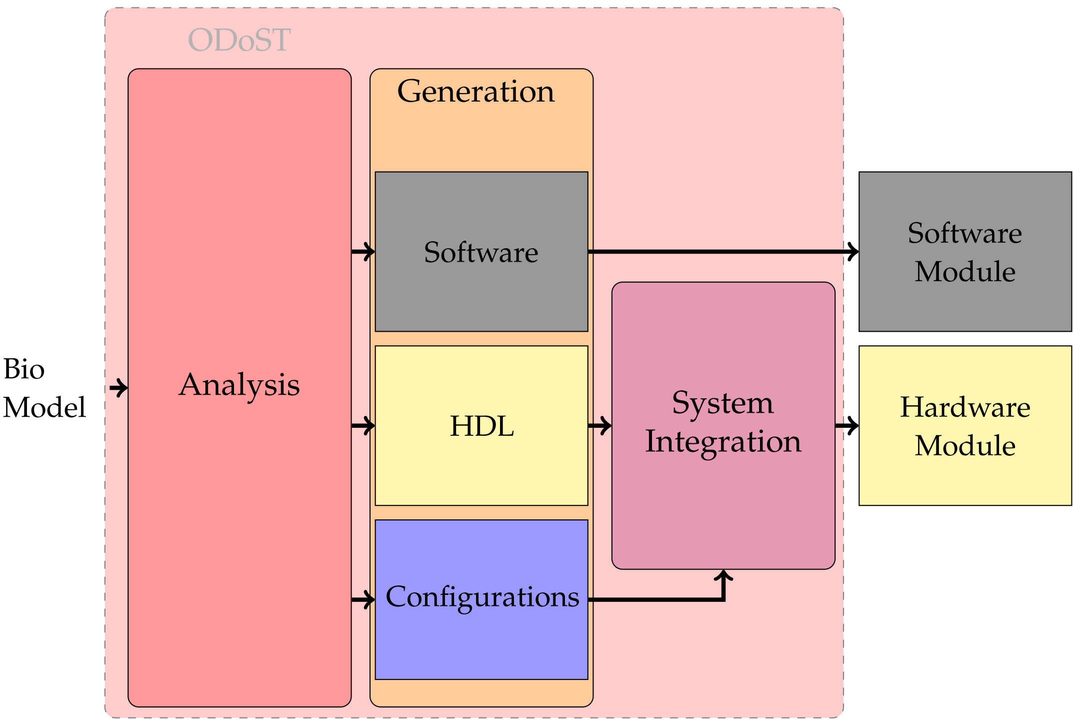


Figure 3.8: Design flow of ODoST.

```
/*
  There are a total of 10 entries in the algebraic variable array.
  There are a total of 4 entries in each of the rate and state
     variable arrays.
  There are a total of 8 entries in the constant variable array.
*/
void initConsts(float* CONSTANTS, float* RATES, float *STATES) {...}
void computeRates(float VOI, float* CONSTANTS, float* RATES, float*
   STATES, float* ALGEBRAIC) {...}
```

Figure 3.9: C representation of model.

3.4.2 *Input Model Format*

The general structure of the input model in C99 is depicted in Figure 3.9. It is derived from the C code representation of the CellML model, generated from the XML and provided as an alternative representation of the model. Inside the C code, state variables are referred to as *STATES*, intermediate variables are referred to as *ALGEBRAIC*, parameters are referred to as *CONSTANTS* and the rates of change for the states are referred to as *RATES*. Model initialisation is done by a single call to *initConsts*, performing the *CONSTANTS* initialisation and populating *STATES* at the initial condition. *computeRate* calculates *ALGEBRAIC* and *RATES* that are used to compute the next micro time step values for *STATES* using Euler's method. In terms of mathematical operations, ODoST currently supports the following:

- addition (+), subtraction (-), multiplication (*), division (/),
- (natural) exponentiation (`exp`), logarithm (`log`), power (`pow`),
- floor (`floor`), absolute (`abs`),
- greater-equal (>=), less-equal(<=), logic and (`and`), logic or (`or`).

In addition, it also supports C-like inline conditional expressions for discontinuities such as state transitions and/or changes in topology. These operations are most frequently used and cover the majority of CellML models. ODoST also provides the flexibility to add new functions if necessary.

3.4.3 *Analysis Phase*

In the analysis phase, ODoST reads through the input biomedical model, analyses the model and obtains the following information for the generation phase:

- The size of *STATES*, *CONSTANTS*, *ALGEBRAIC* and *RATES* and the total size of inputs and outputs;
- For the hardware module:
  - The duration of the critical pipeline path;
  - The equations set, containing equation specific information such as output, inputs, start cycle and end cycle, duration, dependent equations, and operations. Individual operations within the equation are represented with operands, output, operator, duration, start cycle and process stage in the equation.
- For the software module:
  - The extraction of *computeRate* and *initConsts* methods.

The *computeRate* and *initConsts* methods embedded in the biomedical model can be directly extracted. To obtain the rest of the information from the main body of the C code, the following processing steps are taken: expression extraction, RPN (Reverse Polish Notation) conversion and datapath generation.

3.4.3.1 *Expression Extraction*

In the expression extraction step, ODoST first obtains the size of variables defined in the model and reads through the mathematical equations of a biomedical model with the format described in Section 3.4.2. These equations are normally in the form of an infix expression, where operators are written in-between their operands. Infix notion is the most common representation in mathematics and is used in most computer languages [42]. The expression extraction contains the following sub-steps:

1. Identifying a statement with a mathematical equation;

2. Identifying and parsing the input and output variables;

3. Parsing the mathematical statements from string into infix tokens. This is generally straight forward but one challenge is to distinguish a negative sign from a subtraction operator. In order to do this, the following rules are used to evaluate whether a minus sign is a negative sign or subtraction operator:

   - A minus sign immediately after another operator is a negative sign;
   - A minus sign at the beginning is also a negative sign;
   - A minus sign immediately after an (opening) parentheses is a negative sign and;
   - A minus sign is a subtraction operator for all the other circumstances.

The above steps work well with systems that vary continuously. For conditional expressions, the current solution is to divide the whole expression into three chunks: the condition chunk, the true statement chunk and the false statement chunk. Each chunk represents an equation that is passed to the remaining steps individually for further processing.

After the expression extraction step, the size of $STATES$, $CONSTANTS$, $ALGEBRAIC$ and $RATES$ and total size of inputs and outputs are obtained. An equations set containing a list of equations is created with information from the $output_signal$, $input_signals$ and $infix_tokens$ extracted from each equation.

3.4.3.2 *RPN Conversion*

The second step is to convert the infix expression into postfix notation where operators are written after their operands. Postfix is also known as Reverse Polish Notation [42]. The postfix notation is easier to translate into HDL code since the operators are evaluated strictly from left to right and it obviates the need for parentheses that are required by infix notation. The RPN conversion can use any algorithms known in the art (e.g., a shunting-yard algorithm [43]).

3.4.3.3 *Datapath Generation*

The tokens of either operands or operators in the $postfix_tokens$ array are evaluated from left to right. The traditional methodology to evaluate an RPN expression is fairly straightforward. At each input token from left to right, if the token is an operand, push it onto a stack. Otherwise, if it is an operator, remove the most recent operands from stack, evaluate the operation and push the result back onto the stack. Every token in the expression is evaluated and finally the stack contains only a signal value which is the result of the expression.

In a typical biomedical model, individual equations contain variables and numeric values. Therefore, compared to the original algorithm which only calculates the values directly during the evaluation, the algorithm should separate the operands into signals for variables and values and focus on the signal mappings to build the datapath circuit. Furthermore, operations within a mathematical equation may be dependent, but some independent operations can be executed in parallel. Different operations require different times to complete. Since the hardware accelerator is designed with a pipeline infrastructure, shift registers are required to balance the pipeline. Thus, the number of cycles for each shift register should also be accurately estimated.

The datapath representation is built by using Algorithm 3.1. Due to the time-sliced fashion of the pipeline infrastructure, input data is continued to be pushed into the pipeline. Since each operation takes a different amount of cycles, time-based dependencies are implicitly introduced. The input signals arrive to the circuit at the same time (i.e., the same clock cycle) but they are consumed during different cycles depending on the stage of the operations. In general, these are resolved by using registers to buffer the values of intermediary signals until the set of inputs to a execution core are all valid. In the algorithm, it is achieved by breaking the datapath into a set of execution stages via an auxiliary stage counter.

Reading the postfix tokens from left to right, the algorithm first checks whether the given token is an operator, a variable operand or a value operand. All the available operators are predefined in the *operators* tuple. Variable

Algorithm 3.1 Postfix to datapath representation

```
Require: operators, variable_definitions and value_definitions
Ensure: operations and internal_signals data structures
 1: kernel BUILD-DATAPATH(tokens)
 2:     stack ← []
 3:     stage ← 0
 4:     for all token ∈ tokens do
 5:         if token ∈ value_definitions then
 6:             PUSH(stack, token)
 7:         else if token ∈ variable_definitions then
 8:             input_sig ← RECORD-INPUT-SIGNAL(token);
 9:             PUSH(stack, input_sig)
10:         else if token ∈ operators then
11:             operands ← []
12:             variable_operands ← []
13:             for i ← 0, NUM-OPERANDS(token) do
14:                 operands[i] ← POP(stack)
15:                 if operands[i] ∈ variable_definitions then
16:                     variable_operands[i] ← operands[i]
17:                 end if
18:             end for
19:             for all sig ∈ variable_operands do
20:                 if n ← STAGE-AVAILABLE(sig) ≥ stage then
21:                     stage ← n + 1
22:                 end if
23:             end for
24:             for all sig ∈ variable_operands do
25:                 MARK-SIGNAL-CONSUMED(sig, stage)
26:             end for
27:             out_sig ← MAKE-INTERNAL-SIGNAL();
28:             RECORD-OPERATION(token, operands, out_sig, stage)
29:             PUSH(stack, out_sig)
30:         end if
31:     end for
32: end kernel
```

names are defined as upper case characters and values are defined as digits. An operand *stack* is used to hold previous operands. If a token is a value, it is pushed onto the *stack* directly (L5-6). If a token is a variable, an intermediate input signal with unique name is created. Both the input token name and generated signal name are recorded into the *internal_signals* data structure. The generated signal name is then pushed into the *stack* (L7-9).

If an operator is encountered, the necessary number of operands are popped (L13–18) from the *stack*. The variable operand(s) extracted (L15-17) are checked with their availability with the current stage. If one of the operands is not ready, the operation is delayed to the next stage (L19-23). The variable operand(s) in the *internal_signals* data structure are updated with the calculated stage and marked as consumed (L24-26). An intermediate output signal with a unique name is created and recorded into the *internal_signals* data structure (L27). The operation is finally recorded into the *operations* data structure with the operator, operands, output and stage (L28) and the output signal pushed onto the *stack* for further processing (L29).

After the complete evaluation of the RPN expressions, the algorithm ends with two data storage structures *operations* and *internal_signals* which are ready to be passed into the generation phase for further processing. The *operations* data structure holds the information of each operation within the mathematical equation including the name of the functional core, required execution time in the form of clock cycles, operands in the form of signal names or numerical values, name of the output signal, stage and starting cycle. The *internal_signals* data structure contains all the internal signals that are created within the algorithm, the cycle when they are produced and the cycle and stage when they are consumed, the name of the signal it receives the value from and the name of the signal it passes the value to. There are three types of internal signals generated and different prefixes are used to distinguish them (XX represents the index of the signal):

- iXX: represents a signal coming from an input of the equation generated by Record-Input-SignaL();

- zXX: represents a signal from the output of an operation generated by MAKE-INTERNAL-SIGNAL();
- pXX: for the operations with their outputs being registered, e.g. $(A + B) * (C/D)$, the result from addition should be registered waiting for the division to finish, an internal signal pXX is also created and recorded within MAKE-INTERNAL-SIGNAL().

As discussed in Section 3.3, a general biomedical model is a set of ODEs and mathematical functions. An ODE describes how a biomedical state, such as ion concentration and membrane potential, changes over time. For the numerical integration, the rate of change is first computed which is dependent on other intermediate variables. A numerical method such as Euler's method estimates the state value at the next time instance based on the input rate and current state value. To link these dependent equations, the *duration* of the individual equations are also captured and passed to the generation phase for further processing. Also, the partition size of processing is equal to the total duration of the complete one micro time step model computation in terms of cycles. It is recorded for control and memory size allocation.

3.4.3.4 *Equations Aggregation*

Once the datapaths of individual equations are built, ODoST loops through the *operations* data structure for all the equations, builds the dependencies between the equations and records them in an *equations* data structure. Each element within *equations* holds the information of an individual equation including inputs, output, start cycle, duration and depending equation(s). Also, an *equations_internal_signal* data structure is built and records the internal input/output signals between the equations.

3.4.4 *Generation Phase*

In the generation phase, a templating engine is used to perform the code and configurations generation. A template is a plain-text file with embedded place-

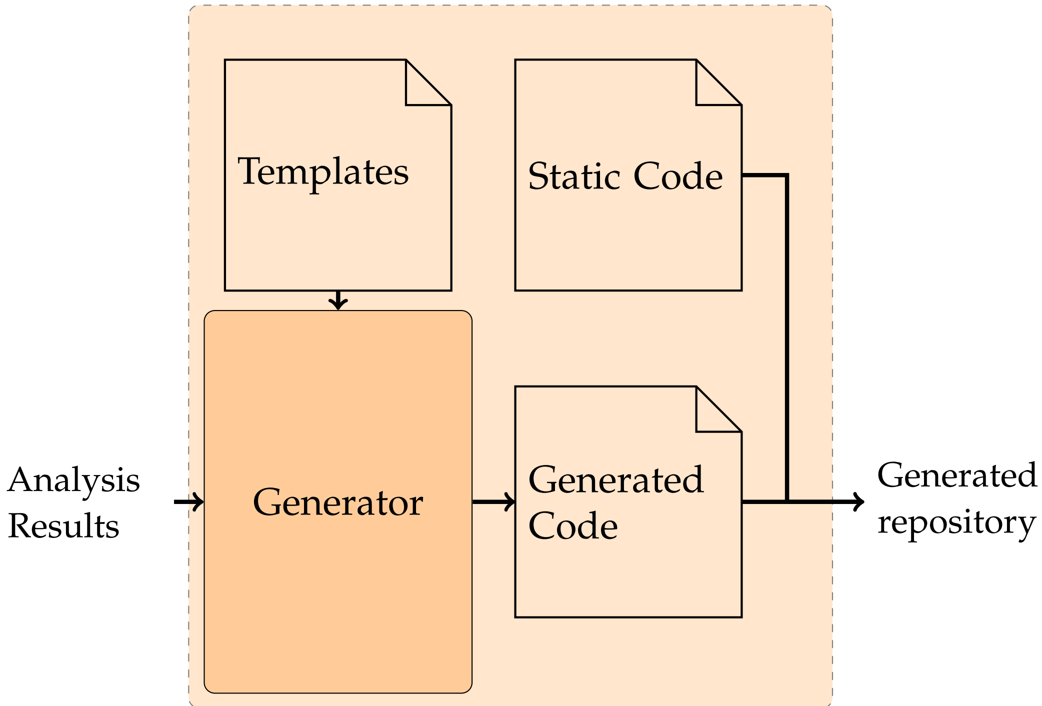


Figure 3.10: Generation structure.

holder blocks that must be substituted or processed by the templating engine. Figure 3.10 shows the general generation structure. The results from the analysis phase are passed in and rendered into the pre-defined templates to create software, HDL code or configurations using the templating engine. Jinja2 [85], a Python based templating tool, is used for this work.

3.4.4.1 *HDL Generation*

In the HDL generation, ODoST creates the entire hardware accelerator, as shown in Figure 3.3, that is to interact with the on-chip memory. The abstract structure of the hardware accelerator is illustrated in Figure 3.6. The hardware accelerator possesses a nested framework as shown in Figure 3.11. *ModelWrapper* is the outer wrapper that interacts with the on-chip memory for data exchange and controls the data flow to and from the hardware accelerator. *ModelCore* handles the iterative integration of the model and *ModelUnit* is responsible for one micro time step of model computation and model integration. *ModelCompute* contains an aggregate of *ALGEBRAIC* and *RATES* computation. It uses shift registers in-between dependent equations to ensure

accurate pipeline flow. Similarly, *ModelIntegration* contains an aggregate of numerical integration of *STATES* with Euler's method.

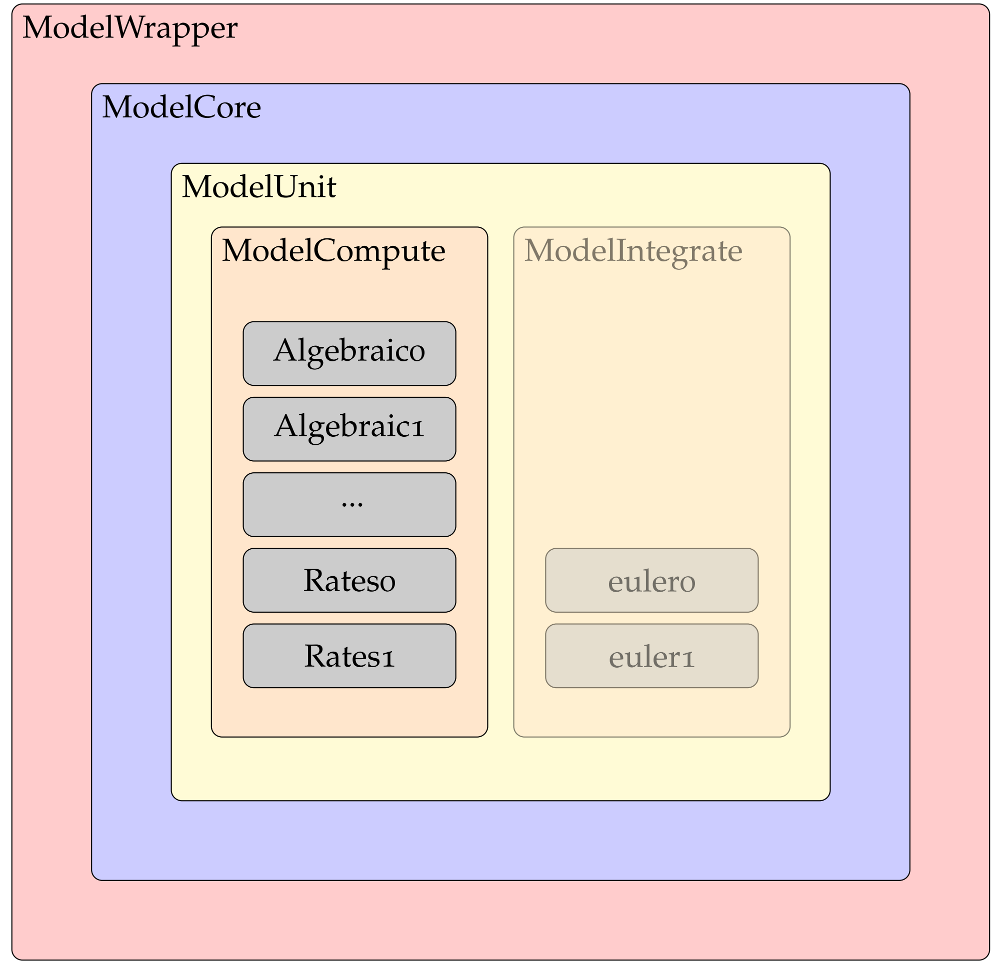


Figure 3.11: Hardware accelerator nested framework.

According to the design illustrated in Figure 3.11, five HDL templates are developed to be processed for the customisation. General codes without customisation requirements are developed directly and referred to as "Static code" in Figure 3.10, e.g., *ModelIntegrate* and *EulerMethod* in Figure 3.11, floating point cores and shift registers etc. The major parts of the HDL templates to be processed are:

EQUATIONS COMPUTATION Equations Computation is the main template in HDL generation. Computations for individual equations are generated based on this template. It is customised and builds one VHDL file for each ordinary equation and three files for equations with a conditional expression for

```
entity {{output_signal}}_comp is
  generic ( S        :    integer := 32);
  port ( clk         :    in  std_logic;
         rst         :    in  std_logic;
         {%- for i in input_signals %}
         {{i}}                   :    in  std_logic_vector(S+1 downto 0);
         {%- endfor %}
         {{output_signal}}       :    out std_logic_vector(S+1 downto 0));
end entity;
```

Figure 3.12: Templated entity declaration.

```
{%- for s in internal_signals %}
signal  {{s['name']}}           :    std_logic_vector(S+1 downto 0);
{%- endfor %}
```

Figure 3.13: Internal signals declaration.

evaluation, i.e., the true statement and false statement respectively. Examples of customisations in the template are illustrated below:

- Entity declaration: Figure 3.12 shows a template fragment that declares a VHDL entity block for an equation. Within the template, there are two kinds of delimiters: {%...%} is used to execute statements and {{...}} prints the result of the expression to the template. The entity name uses the name of *output_signal* with the *_comp* suffix. The input signals and output signal of the entity are obtained from *input_signals* and *output_signal* that are obtained from expression extraction.

- Internal signal declaration, Figure 3.13 shows the declaration of all the internal signals required in the VHDL architecture block. These signal names are obtained from iterating through the *internal_signals* results from the datapath development.

- Signal shifting, Figure 3.14 shows the initiation of shift registers that are used to delay the data signal by multiple clock cycles so that it can be used in another operation. It loops through *internal_signals*, initialises the shift register with a unique name and maps the number of *cycles*, *from* signal and current signal *name* to the *shift_register* component.

```
{%- for s in internal_signals %}
map_{{s['from']}}_{{s['name']}}_reg :
  entity work.shift_register
    generic map ( cycles => {{1+s['cycles']}} )
    port map ( clk         => clk,
               enable      => '1',
               sr_in       => {{s['from']}},
               sr_out      => {{s['name']}}      );
{%- endfor %}
```

Figure 3.14: Signal shifting.

```
{%- for o in operations %}
op_{{loop.index}}_inst :
  entity work.{{o['core_name']}}
    port map ( clk                          => clk,
               rst                          => rst,
               {%- for i in o['inputs'] %}
               data{{loop.index|alphabet}}  => {{i}},
               {%- endfor %}
               R                            => {{o['output']}});
{%- endfor %}
```

Figure 3.15: Operations initiation and mapping.

- Operations initiation and mapping: Figure 3.15 shows the initiation of operations and the port mapping of the inputs and output signals/values of the operations. In particular, it loops through the *operations*, initialises the function core of each operation with a unique name and maps the *inputs* and *output* signals/values into the relevant ports.

MODELCOMPUTE *ModelCompute* contains an aggregate of the *ALGEBRAIC* and *RATES* computation. It connects the equations with input/output data and dependent equations. The template for *ModelCompute* has a similar code syntax as defined in the template for equations computation. For example, the declaration of *STATES*, *CONSTANTS*, *ALGEBRAIC* and *RATES* arrays in the entity declarations are customised according to their sizes; internal signals and shift register declarations are customised from *equations_internal_signals*; and the equation initialisations and mappings are customised from the *equations* data structure as shown in Figure 3.16.

```
{%- for e in equations %}
  entity work.{{e['output']}}_comp
    port map ( clk                          => clk,
               rst                          => rst,
              {%- for i in e['inputs'] %}
               in_{{i}}                     => {{i}}_in,
              {%- endfor %}
               out_{{e['output']}}          => {{e['output']}}_out);
{%- endfor %}
```

Figure 3.16: Equations initiation and mapping.

MODELUNIT *ModelUnit* is responsible for one micro time step model computation and model integration. It interconnects *ModelCompute* with *ModelIntegrate* directly since pipelines inside the two elements are already balanced. Apart from specifying the size of the variable arrays in the entity, the main customisation required is the cycles control for registering *STATES*, *CONSTANTS* and *VOI*, which can be obtained from the durations of the model computations and integrations.

MODELCORE *ModelCore* handles the iterative integration of the model. Apart from the variables size specification, the customisation required for the template is the total cycles for the whole macro time step integration, which is derived from the duration of the model computation and integration and then multiplied by number of micro time steps.

MODELWRAPPER *ModelWrapper* is the outer wrapper that interacts with the on-chip memory. It uses a state machine to control the data flow to and from the hardware accelerator as described in Section 3.3.4.2. The main customisation requirements of its template are the sizes of signals, FIFOs and the start and end addresses of the input/output data in the on-chip memory. The values for the customisation can be derived from the duration of the micro time step computation and integration and the size of input/output variables.

3.4.4.2 *Configuration Generation*

During the configuration generation, a set of configuration file templates are used to generate configuration files and scripts for system integration. The configuration file needs to be device specific since interconnection fabric is necessary. In our work, Qsys system integration tool provided by Altera [8] with an Avalon bus is used. The Qsys provides an automatically generated interconnect logic to connect intellectual property (IP) functions and subsystems. The Qsys configurations are stored in a *.qsys* file that contains a clock source, an on-chip memory, an IP Compiler for PCIe Express, a DMA controller and the hardware accelerator subsystem. The hardware accelerator is generated using a script written in the TCL scripting language [77]. Most subsystems and interconnections in the Qsys configuration are identical for different biomedical models except the hardware accelerator and the relevant memory allocation. ODoST provides a template for the TCL script to generate the hardware accelerator subsystem. The template sets up the module properties, file sets and connection interfaces. The only customisation required is the specification of source files for the equations, which are processed according to the sizes of $ALGEBRAIC$ and $RATES$ variables.

The on-chip memory is used as the buffer for external inputs and outputs and the information exchange between the host and the FPGA. The size of the memory varies according to the input and output size for different models. The memory uses dual port access. Figure 3.17 illustrates the allocation of the on-chip memory. Addresses 0x00000000 to 0x000001FF are reserved for information exchange between the host and the FPGA. Address 0x00000200 is the start address IA_s for the input data from the host to the FPGA. The end address of the input data, IA_e, is dependent on the input size. The offset of the end address is the size of the inputs for each biomedical model S_i multiplied by the number of cells N. This size is multiplied by four, because numbers are represented in IEEE 754 single precision floating point which takes 32 bits, hence four bytes. Therefore, the end address for the input data in the on-chip memory is:

Address	Memory
0x00000000	
0x00000001	
...	
0x00000100	Data Control
0x00000101	No. of Cells
0x00000102	No. of Cells
0x00000103	No. of Iterations
0x00000104	No. of Iterations
...	
0x00000200	Input Data
0x00000201	Input Data
0x00000202	Input Data
...	Input Data
0x00000aa0	Output Data
0x00000aa1	Output Data
0x00000aa2	Output Data
...	Output Data

Figure 3.17: On-chip memory allocation.

$$IA_e = IA_s + S_i \times N \times 4 - 1$$

The start address for the output data, OA_s, from the FPGA to the host is $IA_e + 1$. Similarly, the end address of the output data, OA_e, is dependent on the size of outputs and is calculated as follows:

$$OA_e = OA + S_o \times N \times 4 - 1$$

where S_o is the size of the outputs.

For efficient memory and logic use, additional space is added to pad the memory size to the next power of 2 bytes.

3.4.4.3 *Software Module Generation*

Each HAM includes a software module acting as a bridge between the biomedical simulator and the hardware module. The flow of the software module is discussed in Section 3.3.4.1. The module uses a PCIe library provided by Al-

tera to initialise/finalise the PCIe connection, and exchange information and data with the hardware module.

The template begins with constant definitions of the number of cells, sizes of *STATES*, *CONSTANTS*, *ALGEBRAIC* and *RATES* and offset addresses in the on-chip memory for the data control, number of micro time steps, number of cells, input data stream and output data stream. Apart from the sizes of variables that are put as placeholders and to be substituted by the analysis results, the remaining constants are predefined.

For testing purposes, a simple simulation function and a pure software comparison are included in the module. The C based template contains an {{*INIT_CONSTS*}} and {{*COMPUTE_RATES*}} placeholders that are ready for the *initConsts* and *computeRates* methods extracted directly from the model input file. The *initConsts* method is used to initialise the model for both the HAM and the pure software implementation that is used for comparison. The *computeRates* method is used as the model computation for the pure software implementation.

After the generation process, the generated module file in C format together with a Makefile and the PCIe library comprise the software module of the HAM.

3.4.5 *System Integration*

As mentioned earlier, Qsys is used for the system integration. The final system contains the following components:

- An on-chip memory for information and data exchange between the host and the FPGA;
- A customised hardware accelerator that interacts with the on-chip memory through the Avalon Memory Mapped (Avalon-MM) interface performing pipelined model computation and integration;
- A PCI Express IP core that connects the PCIe interconnection between the host and the FPGA and interacts with the on-chip memory directly via

the Avalon MM interface for signal transfer or through a DMA controller for data transfer;

- A DMA controller that interacts with the on-chip memory via the Avalon MM interface to perform memory transfer tasks between the PCI Express IP core and the on-chip memory;
- A clock source that is defined by Altera Phase-Locked Loop (PLL) IP core to synchronise the entire subsystems.

The system integration does not use any interactive GUI interface that is usually employed for digital hardware design, as in Quartus II. ODoST provides two scripts aimed at command-line system integration and synthesis without the need for user interaction. The first script executes the Qsys generator command and creates a fully integrated synthesizable hardware module. The second script executes the Quartus mapping, fitting, assembling and timing commands and creates a programming file that is ready to be loaded on the device. In other words, the generation of a hardware accelerator for a biomedical model is truly automatic, starting from the high-level model specification and resulting in an execution ready programming file for the FPGA board, complemented by the corresponding host control software.

3.5 EVALUATION

To experimentally evaluate our proposed approach, ODoST was used to generate the HAMs based on a range of biomedical models. The HAMs are assessed by their resource usage, processing speed and power efficiency. The processing speed and power efficiency are also compared with CPU and GPU implementations.

3.5.1 *Models*

Four biomedical models ranging from low to high complexity were selected from the CellML repository. The HAMs are generated by running ODoST on these models. The four models are:

- The Hodgkin-Huxley model developed by Hodgkin and Huxley [53] which describes the flow of electric current through the surface membrane of the squid giant axon;
- The Beeler-Reuter model developed by Beeler and Reuter [21] which describes the membrane action potential of mammalian ventricular myocardial fibres;
- The Hilemann-Noble model developed by Hilemann and Noble [51] which describes extracellular calcium transients with tetramethylmurexide in the rabbit atrium;
- The Tusscher-Noble-Noble-Panfilov (TNNP) model developed by Ten Tusscher et al. [96] which describes human ventricular tissues.

To simplify, the one cell and one micro time step cell computation (including the numerical integration) is defined as one *iCell* which stands for one iteration cell. As a consequence, the cost per *iCell* is the average cost for one iteration cell which includes the computation, communication and other overheads. The complexity of the four models, given by the number of variables and floating point operations for one *iCell*, are listed in Table 3.2. As can be seen, the four models vary from low complexity (Hodgkin-Huxley) to high complexity (TNNP) in order to provide a broad range of performance measurements. To quantify the scalability of a typical HAM design, the spatial density (number of cells) and temporal density (number of integration time steps) is also considered. A set of experiments are performed on the HAMs corresponding to the four models and the measured results are represented and discussed in the rest of this section.

Model Name	input(bytes)	output(bytes)	Add	Sub	Mul	Div	Exp	Log	Pow
Hodgkin-Huxley	48	56	13	11	21	10	6	0	2
Beeler-Reuter	80	104	49	34	60	28	25	1	1
Hilemaan-Noble	280	220	62	72	149	52	21	7	4
TNNP	252	336	129	64	156	129	52	26	4

Table 3.2: Metrics of the considered biomedical models.

Family	Stratix IV
Device	EP4SGX530KH40C2
Combinational ALUTs	424960
Memory ALUTs	212480
Registers	424960
Memory Bits	21233664
DSP Blocks	1024

Table 3.3: Stratix IV EP4SGX530KH40C2 device specifications.

3.5.2 *Experimental Setup*

The HAMs were generated by the ODoST tool from the available C code of the biomedical models directly. Each HAM contained a hardware module and a software module. Testbenches were generated automatically with the software module. Minimal effort was sometimes required to adjust the C code of some models into a format favourable ODoST. The ODoST tool terminates with the output of a hardware module of the core accelerator coded in VHDL, all the required external IP cores, Quartus project configurations and a script for auto synthesis. The auto synthesis script converts the hardware module to a binary FPGA configuration. Both the hardware module generation process and synthesis process require Altera's Quartus II 12.1 software suite. Although the generation processes by the ODoST tool normally takes a few minutes to complete, it is worth noting that the synthesis time for the hardware modules were significantly higher, ranging between half an hour to three hours depending on the complexity of the model. Finally, the software module generated by the ODoST tool embeds the control program and test stimulus.

In the FPGA test platform, the clock frequency was set to 100 MHz for the entire system and tests were performed on the Terasic DE4 development board [91] featuring an Altera Stratix IV EP4SGX530 FPGA. The DE4 board was connected with a 3.2 GHz Intel Core i5-3470 CPU and 16GB of RAM on the host machine. Communication was through a PCIe x4 interface which supports up to a 10 Gb/s data transfer rate. The hardware module was compiled by the Quartus synthesis tool and the software module was compiled with GCC 4.8.2. Table 3.3 lists the total device capacity.

The CPU test platform was an Intel Xeon E5-4650 @2.7 GHz with eight cores and 16 hardware threads [55]. The CPU was at a higher specification than the one used for the host machine in the FPGA test platform. In addition, the system had the Intel compiler suite installed which is one of the faster compilers for x86 and supports comprehensive auto-vectorisation using Streaming SIMD Extensions (SSE). The pure software implementations were compiled with icc 14.0.2 running on a Linux 2.6.32-358 64-bit kernel. For each biomedical model, four software test cases are measured for comparison with the relevant HAM: single thread unoptimised, single thread with SSE optimisation, sixteen threads unoptimised and sixteen threads optimised with SSE.

The results of the Beeler-Reuter Model were also compared to previous GPU results [90]. The GPU test platform that was used was a NVidia Tesla C2070 GPU with 448 Streaming processor cores and 6 GB of GDDR5 memory [73] attached to a system with an Intel Xeon X5650 @2.67 GHZ with 6 cores and 12 GB of DDR3 RAM. Shubhranshu [90] developed an unoptimised and automated GPU implementation and a hand optimised GPU implementation. The automated GPU implementation is used to compare with the same automatically generated HAM for the Beeler-Reuter Model. The hand optimised GPU implementation will be refered to when studying energy consumption in Section 3.5.5. The GPU device to host computer transfer rate configured in his experiment was 8 Gb/s.

3.5.3 *Synthesis Results*

The Quartus compiler uses a set of modules to convert the synthesizable hardware modules (in VHDL) into output files for device programming. In the experiments, a script generated by ODoST was used to automate the compilation processes with the Analysis & Synthesis, Fitter, Assembler, and TimeQuest Timing Analyzer. The resulting synthesis results are used to estimate the resource consumption and clock frequency of the HAMs.

3.5.3.1 *Resource Consumption*

The estimated resource consumptions were obtained from the Quartus Fitter. The resources were divided into categories of Logic, Registers, Memory and DSPs. The total device capacities for these resources are listed in Table 3.3. The units of resource consumption are represented as a percentage of the total device capacity in Figure 3.18. All four generated HAMs passed the first step of the Quartus compilation: analysis and synthesis. However, the HAM for the TNNP model did not pass the Quartus Fitter because its DSP requirement exceeded the DSP blocks available within the device. The percentage of resource usage for each model was observed to be consistent with its complexity. The number of floating point cores is the most critical factor that contributes to logic, registers and DSPs consumption. Of these floating point cores, multipliers, exponential functions, power functions and logarithms use DSP blocks. DSPs provide an order of magnitude higher performance with lower power consumptions than pure logic elements. However, when they are used heavily to accelerate these floating point cores, DSPs become a bottleneck compared to other resources in the device.

From Figure 3.18, there are more than 50% of resources left after simple cell models such as the Hodgkin-Huxley model and the Beeler-Reuter model have been programmed. These resources can be utilised by replicating the pipeline datapaths in the HAM so that double, triple or even more cells could be executed in parallel. For complex cell models such as the TNNP model, further optimisations can be performed before the high-level synthesis processes by

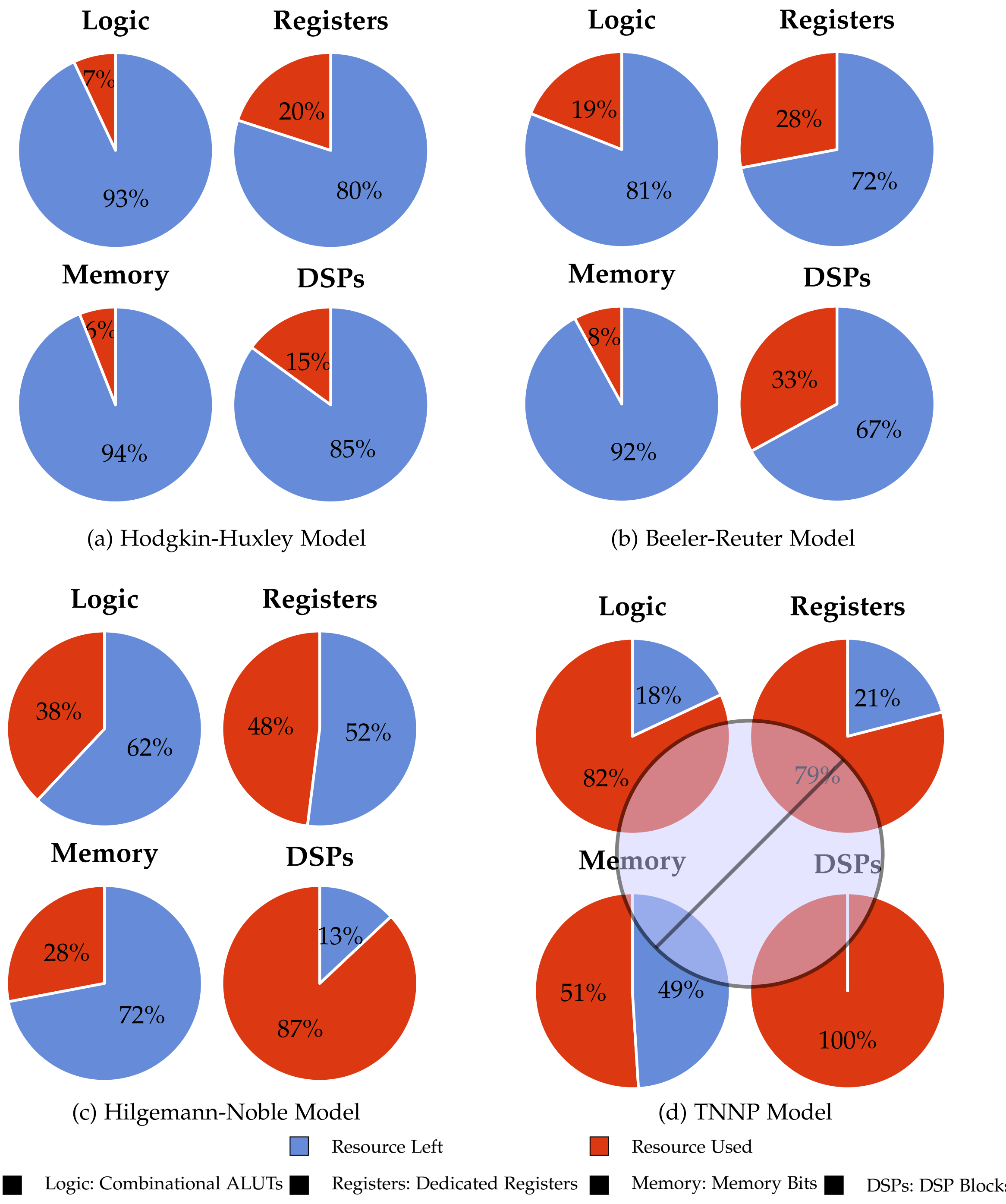


Figure 3.18: Synthesis resource usage results of the generated HAMs.

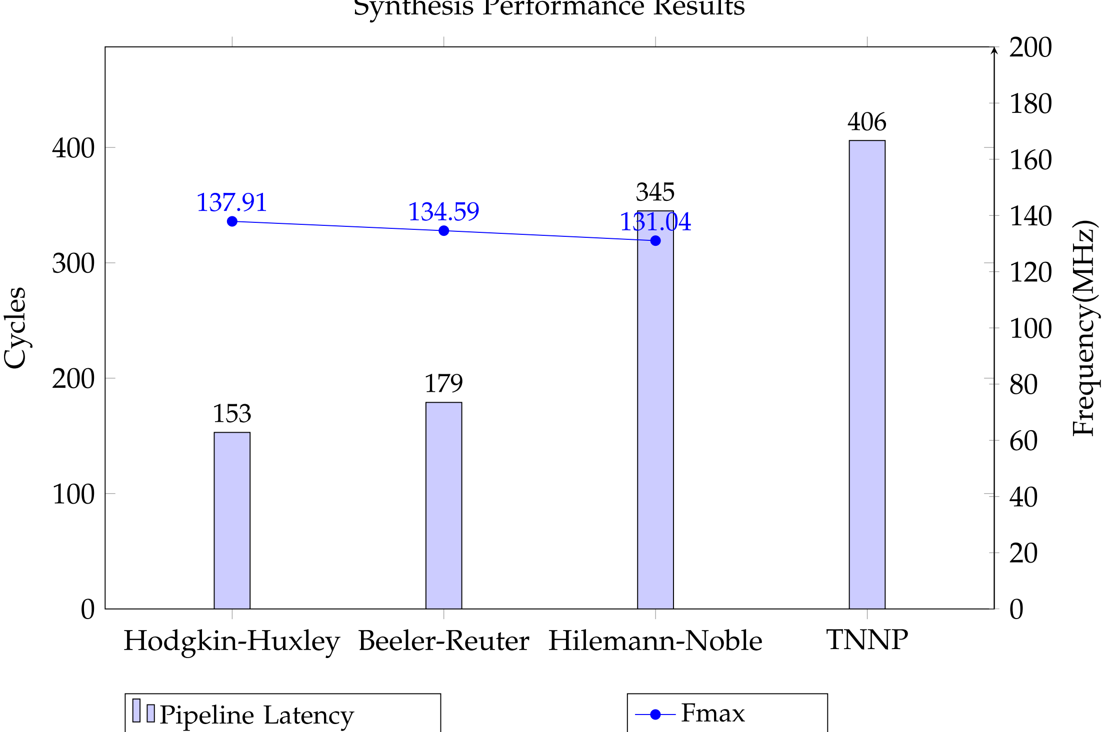


Figure 3.19: Synthesis performance results of the generated HAMs with their pipeline latencies.

ODoST. For example, multipliers are implemented using DSPs, however, these DSPs can be substituted by a pure logic implementation in order to save on DSPs; mathematical equations can be reformulated to favour resource utilisation.

3.5.3.2 *Predicted Clock Frequency*

In Figure 3.19, the latencies for the critical path of the pipeline are presented as a bar chart with a unit of cycles. The values are obtained from the model analysis phase of the auto generation by ODoST. F_{max}, represented as a line chart in the figure, refers to the predicted maximum clock frequency (MHz) from the Quartus TimeQuest timing analysis.

From Figure 3.19, the generated HAMs show good scalability with respect to frequency versus model complexity. F_{max} is limited by the performance of individual floating point cores. The predicted F_{max} reaches frequencies between 130 MHz and 140 MHz with a reasonable fall-off for complex models. According to the synthesis results, from the Hodgkin-Huxley model to the Hilemann-

Noble model, the maximum frequency drop-off is within 5%. This is a very small drop compared to the increase in complexity. With a fully pipelined design in the HAMs, the throughput approximates 1 cell/cycle during the entire computation, hence the performance is dependent on the frequency. More importantly, the relatively stable frequency across models together with the same throughput of 1 cell/cycle across all models will result in a significantly higher speed up compared to the pure software computation (which will slow down for more complex models).

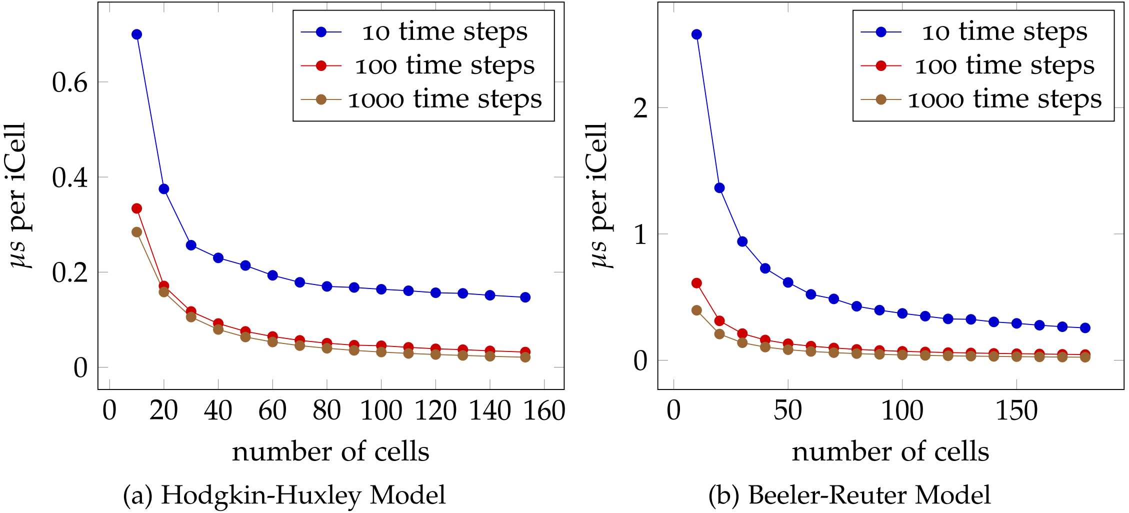
3.5.4 *Performance Results*

The performance results of the generated HAMs are presented using two metrics, the scalability and processing speed. In the scalability analysis, we measure the scalability of the design by varying the number of cells and the number of micro time steps. The purpose of this analysis is to capture the performance behaviour of the basic HAMs with a different percentage of their pipeline utilisation and a different temporal density. The number of cells processed in the HAM is varied from 10 to the maximum number of cells allowed (increasing by a step of 10 cells) to completely fill its pipeline. In practise, it will always be run with a full pipeline, the investigation of scalability is to confirm expectations. In terms of the number of micro time steps for integration of each cell, 10, 100 and 1000 iterations are tested. Varying the number of time steps can be a trade-off between speed and accuracy.

The processing speed measures the cells per second throughput for all three available HAMs and the results are compared to the CPU and GPU implementations.

3.5.4.1 *Scalability Analysis*

The scalability analysis is shown in Figure 3.20. The scalability analysis was performed on the Hodgkin-Huxley model and the Beeler-Reuter model, since their results are adequate to reflect the performance with varying model complexity, number of cells (spatial density) and number of iterations (temporal



(a) Hodgkin-Huxley Model

(b) Beeler-Reuter Model

Figure 3.20: Average execution time per iCell of the HAMs over number of cells and micro time steps.

density). In general, in large scale simulations, the number of cells to process is several orders of magnitude larger than the pipeline length. This scalability test is to confirm correct behaviour and, in practise, the pipelines are always fully filled.

With the number of micro time steps fixed, the average time for one *iCell* execution is approximately inversely related to the number of cells to be processed. The results are represented by different coloured lines in Figure 3.20. The capacity of the pipeline and data communication buffer are strictly linked. Within the capacity range, the more cells processed (increasing spatial density), the less average time per cell is spent and the better the performance. It is expected that maximum performance can only be obtained when the pipeline is fully filled and the experiments were performed to confirm this. The software module partitions and organises the workload so that each partition completely fills the pipeline. The communication overhead is included in the measurement and the partitioning into sets does not create any additional overhead.

In terms of the number of micro time steps, according to Figure 3.20, a higher temporal density result in a reduced average time per *iCell*. This is because more integration time steps lead to more intensive computation and the total ratio of computation time to the data communication time increases. The input

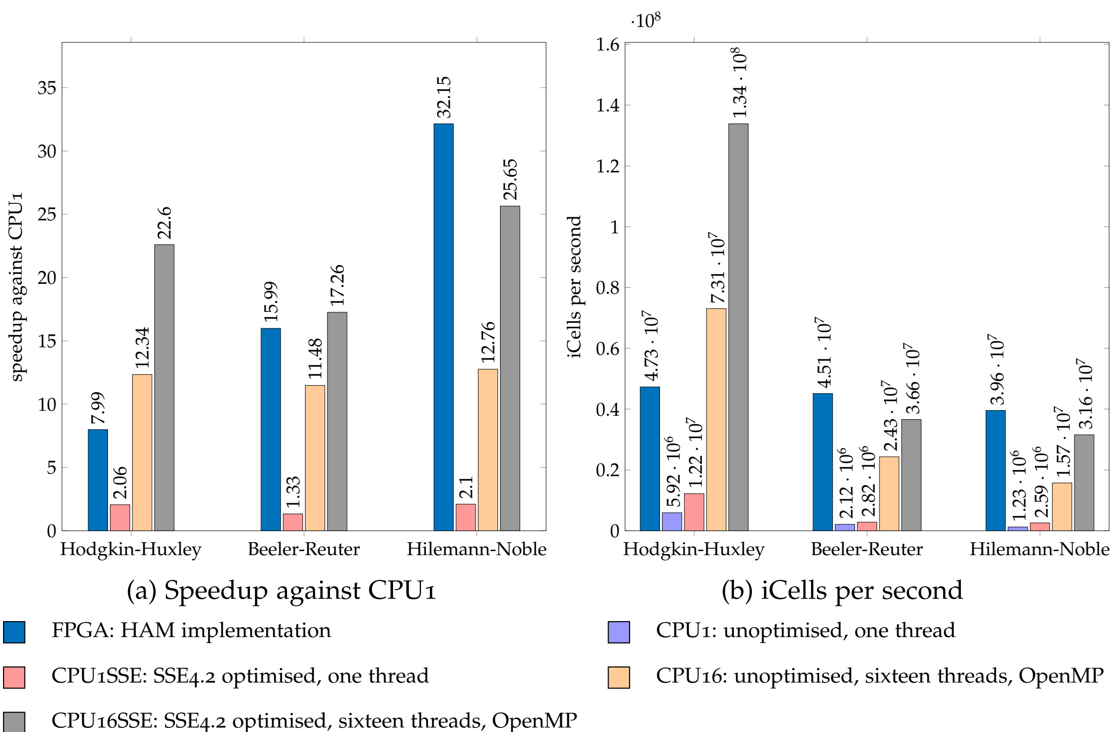
and output data per cell is shown in Table 3.2. As the number of cells per partition is fixed, with increased temporal density, the per cell time decreases at a diminishing rate. Eventually the overhead for input/output transfer becomes negligible. General biomedical simulations favour higher temporal densities due to accuracy and stability requirements, so in the rest of the experiments, a temporal density of 1000 micro time steps was used to compare with the CPU and GPU implementations. 1000 iterations or more are realistic values used in practice [86].

Comparing the *iCell* computation times across the different biomedical models, Figure 3.20 shows that, with a low number of cells, the simpler model (Hodgkin-Huxley) takes less time than the more complex model (Beeler-Reuter). However, the gap shrinks with an increasing number of processing cells as predicted from the synthesis results in Section 2.5.2. When both the models are processing with their pipelines at full capacity, they achieve a similar per *iCell* computation time.

3.5.4.2 *Processing speed*

The results of the three generated HAMs compared to the CPU implementations are shown in Figure 3.21. Since the TNNP model is too large to fit onto the test FPGA board, performance result for this model was unavailable. Two generations of Altera FPGAs have been released since the Stratix IV (Stratix V and Stratix 10) which offer more logic elements and DSP blocks and are large enough for this model [13, 10]. The processing speed is measured in terms of the number of *iCells* per second on all the test platforms. Each test case uses a biomedical simulation of 1 ms with a 1 μs micro time integration step for around 200,000 cells (the actual number of cells are a multiple of the HAM's pipeline latency closest to 200,000). The FPGA result for the Beeler-Reuter model is also compared to the equivalent GPU results in Figure 3.22. It is no surprise that the GPU implementation is much better if hand optimised, but for fairness only automatically generated implementations are compared.

Figure 3.21 shows side by side bar charts for the HAM processing speed compared to the CPU implementations. The left hand side presents the spee-



(a) Speedup against CPU1 (b) iCells per second

Figure 3.21: Processing speed of the generated HAMs compare to the CPU implementations.

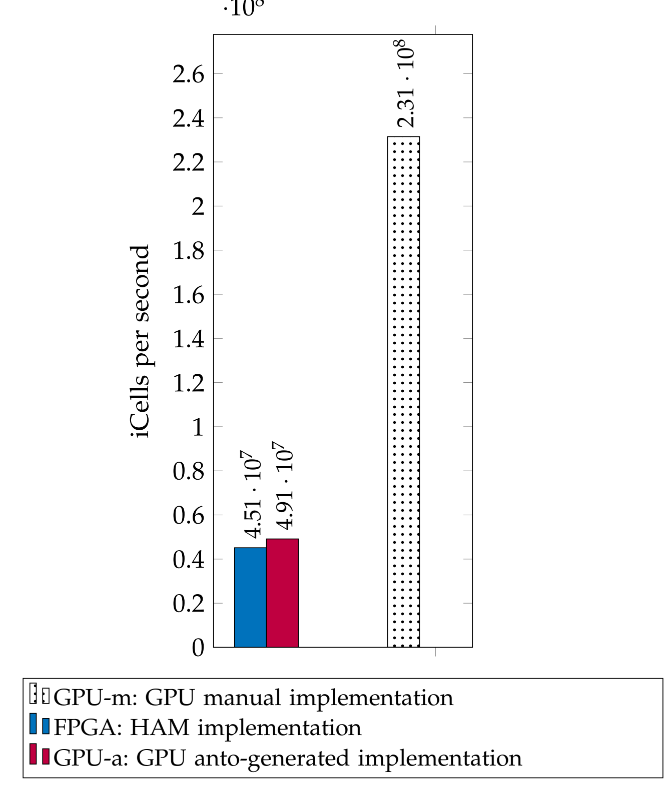


Figure 3.22: Processing speed of the HAM for the Beeler-Reuter model compared to the GPU implementations.

dup of performance relative to CPU1 and the right hand side displays the number of *iCells* per second for the implementations. The results show that the HAMs have stable performance on the FPGA over all the models tested. Complex models performed at a slightly slower speed than simple model due to reduced operating frequency and an increased data communication requirement with more inputs/outputs. However, the performance for complex models reduces significantly when compared to simple models on CPU-based implementations. It follows that the HAM for complex models would have increased speedup over the CPU implementations as CPU implementations have the same tendencies across all models. SSE optimised implementations have around twice the speedup compared to non-optimised implementations and sixteen threaded implementations have around 10 times the speedup compared to the single threaded implementations. All three HAMs demonstrate significant performance advantages over the CPU1, CPU1SSE, CPU16 implementations. For the best case from the experiments, the FPGA-based implementation has a 32x speedup compared to CPU1 and 15x to CPU1SSE. For the Beeler-Reuter and Hilemann-Noble models with a medium to large size, the HAM implementations also outperform the CPU16 and CPU16SSE implementations. GPU results are referenced from work done by Shubhranshu [90], but are only available for the Beeler-Reuter's model. According to Figure 3.22, the automated GPU implementation gains similar throughput compared to the FPGA-based implementation.

It is worthwhile to note that the above implementations across the three test platforms are all automatically generated with minimal effort of design and development. The performance of all the implementations can be improved with manual optimisation. For example, a manual SSE intrinsics implementation with third party optimised advanced mathematical functions can achieve about 2-3 times speedup compared to auto-vectorisation. The GPU implementation can be optimised after implementing shared memory, memory coalescing and constant memory [90]. It can achieve about 5 times speedup compared to unoptimised GPU implementation. For the FPGA implementations, there are

also optimisation strategies like equation optimisation, resource fitting and multiple pipelines. These optimisations are covered in Chapter 4.

3.5.5 *Power Efficiency*

Power efficiency is compared between the HAMs, the best performing CPU implementations (CPU16SSE) and the CUDA-based GPU implementation. The power requirement for the three testing platforms is shown in Table 3.4. The FPGA power usage is estimated using Altera's PowerPlay Power Analyser. The PowerPlay Power Analyser supports accurate power estimations executed at the post-fit phase of the design cycle. To have a fair comparison with a CPU and GPU, we specify the device power characteristics to maximum and junction temperature to the maximum possible degree. The estimated power requirement for the three models were 12.3 W, 15 W and 22 W respectively. The power usage increases with increased model complexity. The CPU power usage is estimated at 130 W and the GPU power usage is estimated at 238 W, both using the Thermal Design Power (TDP). The TDP of a device is the maximum amount of heat generated by the device that the cooling system is required to dissipate in a typical operation. We believe that the TDP is a reasonable estimate of the power consumption for the CPU and GPU during cell computation and integration. This is based on the fact that the repeated use of SIMD instructions usually employs the CPU at the TDP limit [56]. For the GPU, we have included the hand optimised version which is likely to work at the TDP limit. For the auto generated GPU version the power consumption estimate might be less accurate.

The power efficiency of the three models on each platform is measured by the processing speed obtained from Figure 3.21 divided by the power requirement for each device/model from Table 3.4. The resulting values in cells per second per watt are converted to the unit of *kWh* and are presented in Figure 3.23. The results show that FPGA has a better power efficiency than both CPU16SSE and GPU over all models, with nearly 4x the power efficiency of CPU16SSE on the Hodgkin-Huxley model, over 10x the power efficiency of

Testing Platform		Power Measurement (W)	Measurement Basis
Stratix IV EP4SGX530	Hodgkin-Huxley	12.3	PowerPlay Power Analyzer
	Beeler-Reuter	15	
	Hilemaan-Noble	22	
Xeon E5-4650		130	Thermal Design Power
Tesla C2070		238	Thermal Design Power

Table 3.4: Power requirements for the three testing platforms.

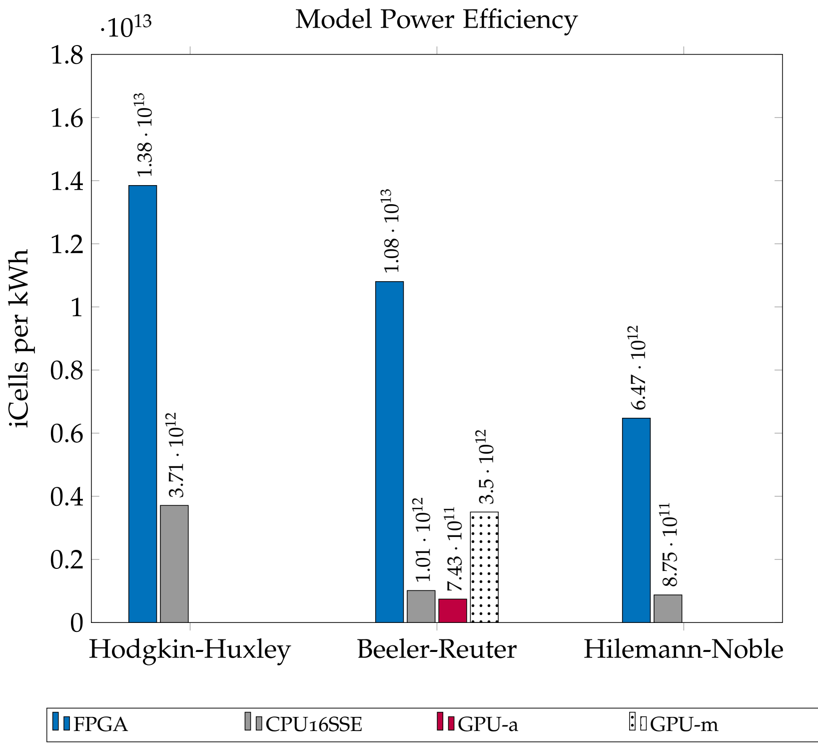


Figure 3.23: Power consumption of the generated HAMs compared to the CPU and GPU implementations.

CPU16SSE and 14.5x the efficiency of a GPU on the Beeler-Reuter Model and over 7x the power efficiency of CPU16SSE on the Hilemann-Noble model. The CPU16SSE and GPU power efficiencies are more closely matched. The automated GPU implementation may not be executing at the TDP limit since it is not fully optimised. The optimised GPU implementation [90] shown in the dotted bar is more realistic. The current auto-generated HAM has around 3x the power efficiency of the hand optimised GPU implementation and there is potential for further optimisations. So, in general, the FPGA implementations are significantly more power efficient than the relevant CPU and GPU implementations, whilst having comparable or better processing speed for complex models.

3.6 CONCLUSIONS

This chapter proposes an approach for the hardware acceleration of biomedical models. ODoST, an ODE-based Domain-specific Synthesis Tool is implemented and used to generate the software/hardware co-design of the hardware accelerator module. The design is general, flexible and applicable for a large range of biomedical models. The thesis uses CellML models and evaluates the hardware accelerator modules for three models with diverse complexity. The results show that FPGAs can provide a highly energy efficient solution with remarkable processing performance compared to both multicore processors and GPUs.

4 PERFORMANCE OPTIMISATION AND RESOURCE UTILISATION

Results from Chapter 3 shows that larger models are not suitable for hardware acceleration due to the physical capacity limit of a FPGA. In this chapter, several optimisation strategies are proposed aimed at relieving the capacity limits and further increasing the performance of hardware accelerators. The contents of this chapter have been submitted for publication as a research article, *Performance Optimisation Strategies for Automatically Generated FPGA Accelerators for Biomedical Models*, in the Journal of Concurrency and Computation: Practice and Experience.

Contributions of this chapter include: (i) proposing several strategies including compiler optimisation, resource fitting and balancing, and multiple pipelines, (ii) analysis and investigation of the effectiveness of the proposed strategies, and (iii) application of these strategies for the automatic HAM generation process provided by ODoST.

Experimental results from this chapter show that (i) optimised FPGA implementations provide significant improvements in processing performance and energy efficiency, and (ii) optimisation relieves the capacity limits of a FPGA enabling larger models to be implemented.

CHAPTER ABSTRACT

Biomedical models that are mathematically described by Ordinary Differential Equations (ODEs), are often one of the most computationally intensive parts of simulations. With high inherent parallelism, hardware acceleration based on FPGAs has potential to increase the computational performance of the ODE model integration, whilst being very power-efficient. ODoST, ODE-based Domian-specific Synthesis Tool, is a tool proposed in previous chapters to automatically generate a complete hardware/software co-design framework for computing models based on CellML. Although it provides remarkable performance and high energy efficiency when compared to CPUs and GPUs, there is still potential for optimisation. In this chapter we investigate a set of optimisation strategies including compiler optimisation, resource fitting and balancing, and multiple pipelines. They all have in common that they can be performed automatically, and can hence be integrated into the domain specific high level synthesis tool. The optimised Hardware Accelerator Modules (HAMs) generated by ODoST are evaluated on real hardware based on criteria such as their resource usage, processing speed and power consumption. The results are compared with single threaded and multicore CPUs with/without SSE optimisation and a graphics card. The results show that the proposed optimisation strategies provide significant performance improvement and result in higher energy efficient HAMs. Furthermore, the resources of the target FPGA device can be more efficiently utilised in order to fit larger biomedical models than before.

4.1 INTRODUCTION

CellML [34] is an open standard mark-up language based on XML that defines custom biomedical models based on differential algebraic equations (DAEs). CellML is used by a variety of tools to describe the DAE models, e.g., OpenCMISS, a general purpose computational library for solving field based equations with an emphasis on biomedical applications [24]. Such modelling

often uses numerical integration of ODEs to simulate dynamic systems in order for researchers to understand different physiological functions. To simulate a model with a fine mesh size, running for millions of time steps, requires a significant amount of computation which can be very time consuming, even with today's fastest CPUs [94].

In Chapter 2, a hardware accelerator with a FPGA-CPU heterogeneous architecture for biomedical models described by CellML has been designed. The HAM (Hardware Accelerator Module) provides an acceleration approach focusing on reconfigurable hardware aimed at high performance with reduced energy consumption. To reduce the effort in implementing accelerators from CellML models, an ODE-based Domain-specific Synthesis Tool (ODoST) has been designed and implemented to automatically create the HAM accelerator framework from a CellML input in Chapter 3. Three CellML models with diverse complexity have been evaluated. The results show that FPGAs can provide a highly energy efficient solution with remarkable processing performance compared to both multicore processors and GPUs.

In the previous chapter, it was shown that large models, such as the model of human ventricular tissue, do not fit into the Terasic DE4 board used for the experiments, due to the exhaustive use of resources. Small to medium models fit well into the DE4 board but there are plenty of resources remaining idle which should be used for potential performance improvement.

In this chapter, three optimisation strategies are proposed to overcome or mitigate these limitations. They are equation optimisation, resource fitting and balancing, and multiple pipelines. These strategies are used to optimise the HAMs before, and after, they are generated by ODoST. Two CellML models are used in the evaluation and results show that with multiple pipelines medium sized models can achieve significant improvement in both processing speed and power efficiency. For large models, equation optimisation and resource fitting/balancing techniques can assist the models to fit into a selected device.

This chapter is organized as follows. Related work is discussed in Section 4.2. In Section 4.3, the HAM structure and ODoST that have been proposed in the previous chapters are discussed. The three optimisation strategies are de-

scribed in Section 4.4, 4.5 and 4.6. In Section 4.7, these optimisation strategies are evaluated in experiments. The chapter is concluded in Section 4.8.

4.2 RELATED WORK

Biomedical models and simulations are used to understand normal and abnormal functions of animals and humans. Apart from CellML, other modelling languages such as the Mathematical Modelling Language (MML) [72] and the Systems Biology Markup Language (SBML) [54] have been developed for storing and interchanging biological models. Tools have also been developed to simulate models written in those modelling languages. Among them, several tools emphasise large scale, continuum simulations that require high performance computation, such as the Cancer Heart and Soft Tissue Environment (CHASTE) [79] and OpenCMISS [24]. In this thesis, an accelerator model based on CellML is designed and built which is intended to be used by OpenCMISS and other simulation packages.

Many case studies have used FPGAs to accelerate the simulation of biomedical models. Yoshimi et al. [99]'s accelerator of a fine-grained biochemical simulation achieved a 100x speedup compared to a single processor during that time. Osana et al. [76] developed a solver-based tool, ReCSiP, for biochemical simulations using Xilinx's XC2VP70-5 and reported a 50 to 80 times speedup compared to Intel's Pentium 4 processor. Thomas and Amano [93] proposed a pipelined architecture for a stochastic simulation of chemical systems and reported that their architecture was 30-100 times faster compared to a pure software simulator. de Pimentel and Tirat-Gefen [39] estimated that a real time simulation of a heart-lung system model was expected to be 90 times faster than a PC. However, their evaluation was calculated theoretically based on the performance of the multiplier on the device rather than a real implementation. Chen et al. [28] implemented a Runge-Kutta ODE solver using FPGAs and Simulink that resulted a 100x speedup compared to a 2.2 GHz desktop CPU. Most of these studies used a manual design and implementation to develop a specialised accelerator model. Manual design is impractical for biomedic-

al/mathematical simulations since it is time consuming and requires hardware development skills, often not found in those researchers with a biological background.

Due to the high requirement of development efforts and skills, many tools have been developed for implementing applications on FPGAs through High Level Synthesis (HLS) such as SPARK [49], DRFM [30], GAUT [31], LegUp [26] and polyAcc [81]. These tools are used to automatically generate hardware circuits from a high level representation, e.g., C, Matlab, Java, etc. In this thesis, a domain specific synthesis tool called ODoST is designed and implemented, which focuses on ODE-based mathematical models and aims at creating the complete datapath of a given model including the data communication and software interfacing.

Equation or compiler optimisation strategies are widely discussed in the mathematical field. These strategies include common subexpression elimination [29], partial product reduction [74] and multivariate Horner scheme optimisation [82]. These commonly used equation optimisation strategies were used as well as other strategies aimed at reducing the hardware resource usage. In this thesis, a LLVM based implementation is developed to optimise the original CellML C code to achieve optimised C code with a lower hardware resource consumption.

A few tools or strategies have been developed in order to fit large designs into target devices where high resources are utilised. Tessier and Giza [92] outlined a procedure to determine the appropriate partitioning of programmable logic and interconnection area to minimise overall device area. Liang et al. [62] formulated a module selection problem and discussed strategies to solve the problem. DeHon [40] presented a hierarchical array design to balance the interconnects and logic use. Each strategy is designed to satisfy a specified domain of problems and, in this thesis, our module is divided into operations and a resource balancing algorithm is performed on each individual operation.

4.3 HAM AND ODOST

4.3.1 *Biomedical Model Overview*

Biomedical models are often represented by a set of ODEs describing time varying variables and parameters. In this thesis, four biomedical models from CellML model repository[1] containing 300+ models are selected. Each CellML model is component based and components are represented by one or more mathematical equations or expressions. For example, the equations:

$$alpha_m = \frac{-(V+47)}{e^{-\frac{V+47}{10}} - 1} \tag{4.1}$$

$$beta_m = 40 \times e^{-0.056 \times (V+72)} \tag{4.2}$$

$$\frac{dm}{dt} = alpha_m \times (1-m) - (beta_m \times m) \tag{4.3}$$

represent the component of *"sodium current m gate"* in the Beeler-Reuter model[2], a model that describes the mammalian ventricular action potential. The model contains a total of 13 components with 26 mathematical equations/expressions. Each CellML model contains a list of state variables (e.g., V and m) that are time dependent, a list of rate constants and intermediate variables (e.g., $alpha_m$ and $beta_m$) and the rates of the states (e.g., $\frac{dm}{dt}$) at time t. For a single time step of a model integration, the values of the intermediate variables are first computed based on the state variables (and rate constants if they are required). The rate of change for the state variable is then computed which is dependent on the intermediate variables. Once the value of rate is available, a numerical integration method is used to approximate the state value at the next time step. A variety of such numerical integration algorithms exist and, in this thesis, a forward Euler's method [16] is used. This method is a simple and

1 http://www.cellml.org/model
2 http://models.physiomeproject.org/e/9a

fast numerical method that is widely used. According to Euler's method, the computation of the state variable m at time $t + \triangle t$ is represented in Eq. (4.4).

$$m_{t+\triangle t} = m_t + \triangle t \times \frac{dm}{dt} \tag{4.4}$$

In order to achieve accurate and stable results, the above process is performed using fine time steps. For example, to integrate 1 ms of the model, the time interval is divided into 1000 time steps with each time step taking 1 μs. At each time step for the *"sodium_current_m_gate"*, the computations in Eqs. (4.1 - 4.3) are performed first to obtain the rates of change and then numerical integration is performed to find the new states after 1 μs. The new state variables are then passed to the next step for the next time integration and so on. During this integration process, each point is integrated individually and independently of other points.

4.3.2 *Hardware Accelerator Module*

A Hardware Accelerator Module (HAM) architecture to accelerate biomedical models has been developed using pipelined floating point operations. The pipelined structure allows a multiple number of independent cells to be accessed cycle by cycle and hence achieve one cell operation per cycle throughput. The hardware/software co-design architecture is shown in Figure 4.1, and is composed of a host computer and a FPGA board connected through a PCIe interface. The arrows indicate the data communication flows throughout the system. The software module is used as a bridge application from the biomedical simulator such as OpenCMISS [24] which interacts with the FPGA by sending and receiving data through the PCIe interconnects.

On the FPGA side, there is a PCIe IP core that interacts with the PCIe connector and maps to the on-chip memory directly for the control signals, and through the DMA (Direct Memory Access) controller for data transfer. The data received from the host computer is written into on-chip memory through the DMA controller. An FPGA data flow controller is implemented as two state

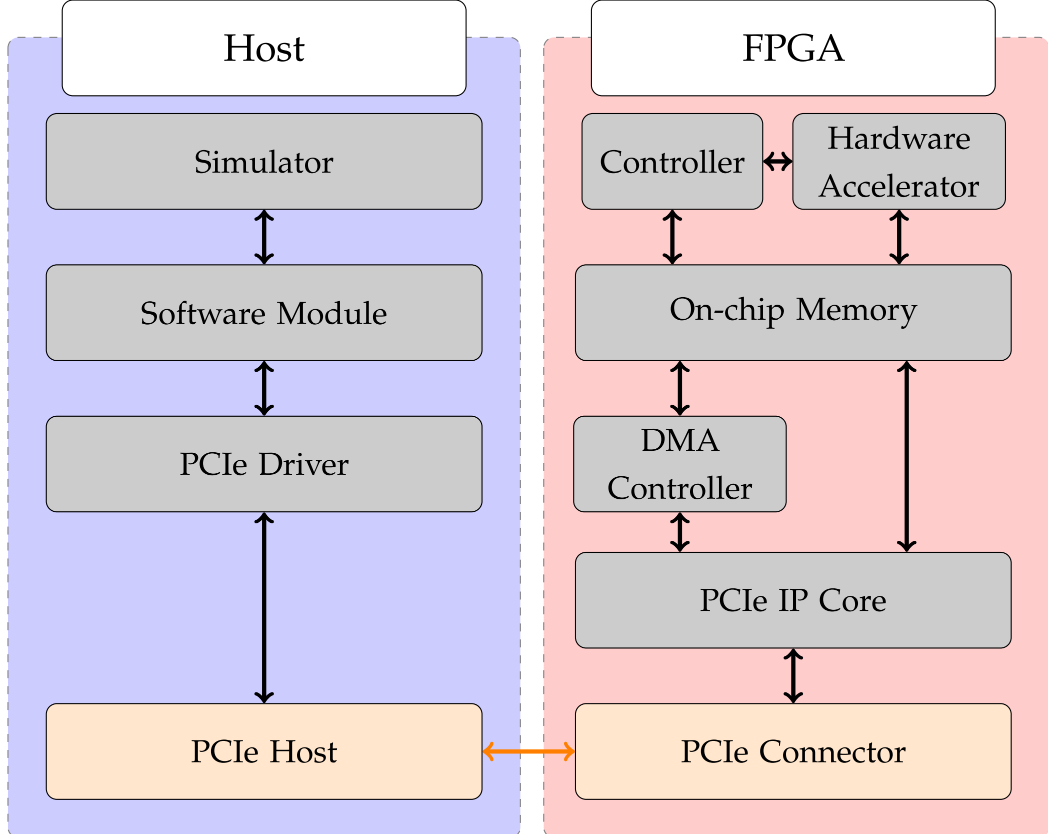


Figure 4.1: Hardware accelerator module system architecture.

machines that are used to send/receive signals to/from the host computer and interact with the CellML hardware model to control the data transfer flow.

4.3.3 *ODE-based Domain Specific Synthesis Tool*

Although it is generally agreed that hardware accelerators using FPGAs have the potential for energy efficient performance improvement, implementing such an accelerator for a given biomedical model requires an enormous effort which might offset the advantages of using FPGA. Therefore, ODoST, a domain-specific High-Level Synthesis (HLS) tool, for ODE-based biomedical simulations has been developed (refer to Chapter 3). The tool is aimed at biomedical scientists and engineers, who often have little knowledge of designing hardware accelerators targeting FPGAs.

ODoST stands for **O**DE-based **Do**main-specific **S**ynthesis **T**ool. It generates the HAM with both the software and hardware modules from an ODE-based

biomedical model. ODoST contains three phases: the analysis phase, generation phase and system integration phase. In the analysis phase, ODoST reads a biomedical input model in C99 format, converts equations from infix format to a HDL favourable data structure through postfix processing. In the generation phase, ODoST uses the Jinja2 [85] template engine to generate HDL codes and configuration scripts based on the data from the analysis phase. In the system integration phase, the configuration files are used to produce the entire hardware module based on the HDL files from the generation phase. The automatic generation process is complete as the produced software and hardware modules are ready to run on the hardware.

Evaluation on real hardware of the HAMs generated by ODoST from selected CellML models was performed and compared against CPU and GPU implementations. The results show that FPGAs can provide a highly energy efficient solution and remarkable processing performance compared to both multicore processors and GPUs. Although the HAM design already showed a significant speed up, there is still potential for further optimisation. Three optimisation strategies: compiler optimisation, resource fitting and balancing, and multiple pipelines, are proposed and discussed in the rest of this chapter.

4.4 COMPILER OPTIMISATION

Compiler optimisation is a process to minimise or maximise some attributes of a computer program. Most modern software compilers provide automatic optimisation techniques that are not provided by ODoST. Therefore, the previous evaluation is not an entirely fair comparison until C-based CellML models are optimised for hardware processing (in a similar way as software compilers do) before being processed by ODoST.

The most common goals for compiler optimisations are either to improve a program's execution speed on a software programmable processor, or to reduce its code size and thereby its memory footprint. These goals are intertwined, as a program executing less instructions is both faster and smaller, and a smaller program may execute faster due to cache effects. Architecture-

independent transformations therefore share the basic idea of eliminating redundant computations. This can be achieved by reusing the results of equivalent computations, or by evaluating constant operations at compile time.

In the targeted code optimisation for ODoST, there is a similar but not identical goal, as the aim is to optimise for lower hardware resource consumption after the high-level synthesis process. Arguably this can be similar to optimising for code size and the usefulness of well-known optimisations, the LLVM's implementations, is investigated. LLVM [61] is a compiler infrastructure written in C++ and is designed for compile-time, link-time, run-time and "idle-time" optimisation of a program written in arbitrary programming languages.

4.4.1 *Local Optimisations*

LLVM facilitates local algebraic simplifications through pattern matching and replacement, as well as constant folding and a simple form of dead code elimination.

The local transformations either reduce the number of instructions, or normalise instructions into a canonical form. These normalisations are important, because they allow the other transformations to match for fewer patterns, and create more constant folding opportunities.

Figure 4.2 shows examples of patterns that are applicable to the CellML equation systems. Eq. (4.5) moves constants to the right-hand side of commutative operations. Operations involving their respective neutral element are eliminated in Eq. (4.6). Subtractions and divisions are replaced in favour of commutative and simpler additions and multiplications with the inverse constant in Eqs. (4.7 and 4.8). Eq. (4.9) transforms a multiplication with -1 into a negation, which is encoded as $0 - x$, because the LLVM-IR (intermediate representation) does not contain a dedicated negation instruction. Eqs. (4.10 and 4.11) are used to eliminate the extra subtraction where possible. The square operation is transformed from a call to the generic power function to a single multiplication in Eq. (4.12). Eqs. (4.13 - 4.16) show the application of constant

$$c \oplus x \Rightarrow x \oplus c \tag{4.5}$$
$$x + 0, x \cdot 1 \Rightarrow x \tag{4.6}$$
$$x - c_1 \Rightarrow x + c_2 \qquad \text{with } c_2 = -c_1 \tag{4.7}$$
$$x / c_1 \Rightarrow x \cdot c_2 \qquad \text{with } c_2 = 1/c_1 \tag{4.8}$$
$$x \cdot (-1) \Rightarrow 0 - x \tag{4.9}$$
$$(0 - x) + c_1 \Rightarrow c_1 - x \tag{4.10}$$
$$(0 - x) \cdot c_1 \Rightarrow x \cdot c_2 \qquad \text{with } c_2 = -c_1 \tag{4.11}$$
$$x^2 \Rightarrow x \cdot x \tag{4.12}$$
$$c_1 \oplus c_2 \Rightarrow c_3 \qquad \text{with } c_3 = c_1 \oplus c_2 \tag{4.13}$$
$$(x \oplus c_1) \oplus c_2 \Rightarrow x \oplus c_3 \qquad \text{with } c_3 = c_1 \oplus c_2 \tag{4.14}$$
$$(x \cdot c_1 + c_2) \cdot c_3 \Rightarrow x \cdot c_4 + c_5 \qquad \text{with } c_4 = c_1 \cdot c_3 \text{ and } c_5 = c_2 \cdot c_3 \tag{4.15}$$
$$(c_1 - x \cdot c_2) \cdot c_3 \Rightarrow c_4 - x \cdot c_5 \qquad \text{with } c_4 = c_1 \cdot c_3 \text{ and } c_5 = c_2 \cdot c_3 \tag{4.16}$$

Figure 4.2: Exemplary transformations done by LLVM [61]. x denotes an unknown value, the c_i are constants, $\oplus$ is either an addition or a multiplication, $\pm$ is either an addition or subtraction.

folding where Eq. (4.15 and 4.16) are even performed on distributive expressions if the number of operations can be reduced thereby.

These equations hold for floating-point arithmetic, with the exception of Eq. (4.8), which is only safe to do if the reciprocal is accurate, and Eqs. (4.14 - 4.16). In a proposed *Unsafe* mode, they are performed unconditionally.

For all these transformations, a simple store-to-load forwarding is performed. This connects consumers of a model variable to the defining expression, circumventing the effect that the underlying pairs of array stores and loads normally break the def-use chain in the intermediate representation.

Algorithm 4.1 Algorithm to turn V^P into a minimal series of multiplications

Require: V : Value and P : int
1: X : Value
2: Powers : map(int->Value)
3: k,R : int
4: $k \leftarrow \text{LOAD}(P)$
5: $X \leftarrow Powers[0] \leftarrow V$
6: **for** $i = 1$ *to* k **do**
7: $\quad X \leftarrow Power[i] \leftarrow new\ mul\ X, X$
8: **end for**
9: $R = \{r_m, ..., r_0\} \leftarrow P - 2^k$
10: **for all** $r_j \in R$ with $r_j = 1$ **do**
11: $\quad$ X $\leftarrow$ new mul X, Powers[j]
12: **end for**
13: **return** X

4.4.2 *Common Subexpression Elimination*

Common subexpression elimination is a compiler optimisation strategy that eliminates commonly used subexpressions. It can be applied either locally in an equation, or globally, among a set of equations. For example:

$$\begin{aligned} y &= x \cdot a + b \\ z &= x \cdot a + c \end{aligned} \tag{4.17}$$

can be transformed to:

$$\begin{aligned} tmp &= x \cdot a \\ y &= tmp + b \\ z &= tmp + c \end{aligned} \tag{4.18}$$

which eliminates one multiplication.

4.4.3 *Higher-order powers*

Algorithm 4.1 is used to transform V^P for an LLVM Value V and an integer power P into a minimal sequence of multiplications.

First, V^{2^k} is constructed with $2^k \leq P \Leftrightarrow k = \lfloor ld(P) \rfloor$, and store it alongside the intermediate powers $V^{2^0}, \ldots, V^{2^{k-1}}$ in the map *Powers*. This requires k multiplications. Then, R is defined to satisfy $V^P = V^{2^k} \cdot V^R$, and construct V^R by reusing the pre-calculated values from the map *Powers* corresponding to the bits set in the binary representation of R. This requires as many multiplications as non-zero bits in R minus 1. One additional multiplication is used in the calculation i.e., $V^{30} = V^{16} \cdot V^{14}$, 4 multiplications are required by V^{16}, and 2 additional multiplications are required by $V^{14} = V^8 \cdot V^4 \cdot V^2$ while the V^8, V^4, V^2 results are reused, plus 1 multiplication for the final product. The resource consumption of a generic power function on FPGA is equivalent to around 8 multipliers. According to the calculation above, the case $P = 31$ is the first to require 8 multiplications, therefore allowing us to implement all integer powers < 31 with at most 7 multiplications.

4.4.4 *Exponential Function Simplification*

Exponential relations are common in biological processes modelled by CellML descriptions. This justifies an extra effort towards the optimisation of expressions involving the exponential function. Expressions of the form:

$$\exp(x \cdot \underline{a} + \underline{b}) \cdot \underline{c} \tag{4.19}$$

are focused on. Here constant subexpressions are underlined. The second multiplication can be folded into the existing addition by using the power laws, which is beneficial in our cost model e.g.,

$$\exp(x \cdot \underline{a} + \underline{b} + ln\underline{c}) \tag{4.20}$$

Applying this pattern without an existing addition has replaced one multiplication by an adder. This results in an overall saving of resource usage since a multiplication is generally more expensive than an addition.

Additional operations can be saved by separating the variable and constant parts of the expressions in sets of n expressions of the form:

$$\begin{aligned} &\exp(x \cdot \underline{a_1} + \underline{b_1}) \\ &\vdots \\ &\exp(x \cdot \underline{a_n} + \underline{b_n}) \end{aligned} \tag{4.21}$$

In the special case $a_1 = \cdots = a_n$, the multiplication can be reused across the expressions which reduces the number of multiplications by $n - 1$. Alternatively, splitting and reusing the variable exponentiation leads to:

$$\begin{aligned} &t_1 \leftarrow exp(x \cdot \underline{a_1}) \\ &t_1 \cdot \underline{\exp(b_1)} \\ &\vdots \\ &t_1 \cdot \underline{\exp(b_n)} \end{aligned} \tag{4.22}$$

which adds one multiplication, but eliminates $n - 1$ exponentiations and n additions.

4.4.5 *Source-to-source Optimiser*

According to the optimisation strategies discussed above, a LLVM-based source-to-source optimiser, *cellml-opt*, was developed. The optimiser generates optimised C code from the original CellML model which uses standard C that follows a fixed scheme. LLVM's C frontend clang parses the C code of the CellML model and constructs the LLVM intermediate representation (IR). *cellml-opt* reconstructs the model's C representation from the LLVM IR.

Program equations of the model map to the LLVM functions. Both C-built-in arithmetic operators (e.g., *, +) and functions defined in the C math library, such as power and exponential functions, are mapped one-to-one to the respective LLVM instructions. Program variables are accessed via pointer values constructed from the function's arguments. The particular variable and the index in an array, e.g., $STATES[2]$, can be easily determined from LLVM's special

GetElementPtr instruction used by a load or a store instruction. The input variables for each equation are treated as independent variables, made possible by the fact that the input code follows a fixed scheme where this is guaranteed. This enables the alias analysis framework of LLVM to detect that all variable accesses, excluding the ones with the same base and index, are independent. On the other hand, all variable accesses with the same base and index can be identified to a single value, greatly helping later optimisations.

The optimised output C code looks almost the same as the input code in terms of style and structure. A simple intermediate representation, "EqIR", is introduced in *cellml-opt* so that the models original equations are transformed into a list of pairs, where a pair consists of a left-hand side expression and a right-hand side expression. The EqIR is then mapped into LLVM IR. Before the resulting C code is finally emitted from the EqIR, temporary variables are added to the equation system to represent LLVM values that are reused across different expressions.

4.5 RESOURCE FITTING AND BALANCING

In a previous chapter, we used ODoST to generate HAMs using the FloPoCo generated floating point cores with DSPs. Generally, DSPs provide an order of magnitude higher performance with lower power consumption. However, DSPs are a limited resource within one FPGA and they can become a bottleneck compared to other resources in the device. In order to solve the problem of exhaustive use of DSPs, FloPoCo also provides floating point cores employing only logic elements. Apart from FloPoCo, there are numerous existing floating point cores provided by the vendors of FPGAs and other third party floating point platforms. Some floating point cores employ DSP blocks and others use pure logic. There is no right answer to the question which floating point core is the best, since it depends on the FPGA resource capacity and the particular biomedical model. With the given resources and model, an effective resource allocation algorithm can provide better resource utilisation and hence increase the computational throughput. Before such an algorithm is proposed,

Family	Stratix IV	Stratix V
Device	EP4SGX530KH40C2	EP5SGXEA7N2F45C2
Equivalent LEs	531,200	622,000
Registers	424,960	938,880
Memory Bits	21,233,664	50,000,000
DSPs	1024	768

(a) Altera FPGAs

Family	Virtex-6	Virtex-7
Device	XC6VHX565T	XC7V485T
Logic Cells	566,784	485,760
Registers	708,480	607,200
Memory Bits	32,832,000	37,080,000
DSP Slices	864	2,800

(b) Xilinx FPGAs

Table 4.1: Resource capability for selected devices.

the resources of a FPGA and the resource usage of selected floating point cores are briefly discussed.

4.5.1 *FPGA Resource Capacity*

The heterogeneous nature of modern reconfigurable devices means it is complicated to determine the capacity of a FPGA. The evaluation of the generated HAMs from previous chapters shows that the logic, registers, memory and DSPs are the four key resources consumed in the implementation, and so the capacities of these resources is focused on. Table 4.1 lists four selected high-end FPGAs from two leading FPGA vendors. The resources of different FPGA generations and vendors are organised differently, however, they follow the similar principle that the main resources are formed by logic, registers, memory and DSPs.

The basic building blocks of the Altera's Stratix series are the Adaptive Logic Modules (ALMs) that provide logic and dedicated registers. However, Stratix V devices use enhanced ALMs that contain 6% more logic and double the number of registers compared to Stratix IV ALMs. In Xilinx's Virtex series, the Con-

figurable Logic Blocks (CLBs) are the main logic resources for implementing circuits. The Altera and Xilinx FPGAs also provide DSP blocks that implement multiplication, multiply-add, multiply-accumulate (MAC), and dynamic shift functions efficiently. They can be effectively used by floating point multipliers, exponential functions, power functions and logarithms to reduce logic usage and achieve high performance. From Table 4.1, we refer to the equivalent logic elements (LEs) for the Altera FPGAs and logic cells for the Xilinx FPGAs so that they can be compared.

The Altera Stratix EP4SGX530 FPGA built in the Terasic DE4 board [91] is the target FPGA device in investigations and evaluations in this thesis. Instead of using the equivalent LEs which is used for comparision between different FPGAs, we use the Adaptive Look-Up Tables (ALUTs) in the analysis. The Altera Stratix EP4SGX530 FPGA contains 212,480 ALMs. Each ALM is composed of two ALUTs, two registers and other logic and interconnects. The ALUTs are used for either combinational or memory and the capacity of ALUTs in the EP4SGX530 FPGA is 424,960. Registers refers to the Dedicated Logic Registers (DLRs).

4.5.2 *Floating Point Cores*

As mentioned, there are numerous existing floating point cores provided by the vendors of FPGAs and other third party floating point platforms. These cores typically exploit the freedom of an FPGA by providing customisation of variable widths and of exponent and mantissa size to meet designers' specifications. They also offer IEEE standard single and double precision cores that are used in the proposed hardware accelerator. This section describes two floating point cores, Altera Floating Point Megafunctions and FloPoCo.

4.5.2.1 *Altera Floating Point Megafunctions*

Altera provides a comprehensive set of IEEE 754-compliant floating point operations as IP modules for their FPGAs [5]. The Altera floating point megafunctions support single and double precision selection and single extended con-

Function	Output Latency	ALUTs	DLRs	ALMs	DSPs	Fmax
ALTFP_ADD_SUB	7	576	345	375	-	227
ALTFP_DIV	33	1646	2074	1441	-	308
	6	207	304	212	16	358
ALTFP_MULT	5	138	148	100	4	274
ALTFP_EXP	17	631	521	448	19	275
ALTFP_LOG	21	1950	1864	1378	8	385

Table 4.2: Altera single precision Floating Point Megafunctions resource usage and frequency estimation for Stratix IV Devices.

figurable precision and can be parameterised by balancing the frequency at which the operators run and the pipeline latency of the operator hardware to fine-tune its overall performance, power and area. The typical resource usages and latencies of the typical single precision Altera floating point cores are displayed in Table 4.2 for the target FPGA. Altera Floating Point Megafunctions support round-to-nearest-even rounding mode, the default of IEEE-754-1985. They also support exception signals for underflow and overflow.

4.5.2.2 *FloPoCo*

FloPoCo, standing for Floating Point Cores, is an open source generator of arithmetic cores for FPGAs [36]. In difference to IEEE floating point representations, FloPoCo has a special floating point format with an additional two-bit prefix. The two bits are only used to signal special-case numbers, namely 00 for zero, 01 for normal numbers, 10 for infinities, and 11 for NaN. In IEEE, these exception signals are handled by the exponent and mantissa. This saves quite a lot of decoding/encoding logic. The main drawback of this format is when results have to be stored in memory as they consume two more bits. However, FPGA embedded memory can accommodate 36-bit data, so adding two bits to a 32-bit IEEE-754 format is harmless as long as data resides within the FPGA. Conversion only needs to take place when passing data to and from the host PC.

In the hardware acceleration design, floating point cores are generated individually. The resource usage and latencies of the generated single precision

Function	Output Latency	ALUTs	DLRs	ALMs	DSPs	Fmax
FPAdd	12	269	622	395	-	523
FPDiv	17	1188	1407	1116	-	308
FPMult	4	73	219	132	4	835
	5	893	524	725	-	370
FPExp	17	436	878	507	2	195
	17	816	939	755	-	237
FPLog	21	831	1210	808	18	175
	22	1434	1885	1399	2	331
FPPow	45	1808	3307	2058	31	177
	50	3884	4359	3620	5	232

Table 4.3: Resource usage and frequency estimation of FloPoCo generated single precision floating point cores for Stratix IV Devices.

compatible FloPoCo Floating Point Cores for Stratix IV devices with DSPs and without DSPs are displayed in Table 4.3. The resource usage depends on the configurations specified during the generation, especially whether to use the DSP blocks or not. *FPAdd* and *FPDiv* are only implemented with pure logic. *FPMult*, *FPExp*, *FPLog* and *FPPow* contain implementations that either favour the use of DSP blocks with hardware multipliers or pure logic. For devices with a large number of DSPs, but a lack of logic, floating point implementations with DSPs are favoured otherwise implementations with pure logic are preferred. Furthermore, multiple variants of a single operation can also be used together in a larger design, e.g., mixing pure logic implementations and DSP implementations, to achieve better resource utilisation and balance.

4.5.3 *Resource Allocation Techniques*

For biomedical models that do not fit on a given FPGA after equation optimisation, resource fitting techniques can be used to balance the logic, register, memory and DSP consumption. This process is called the resource planning process and is performed on the original or optimised C code of the CellML models. Memory in the HAM implementations is mainly used as a data buffer and RAM-type shift registers and it is unlikely to reach the memory capacity

Variation	ALUTs (%)	DLRs (%)	DSPs (%)
Altera (A)	0.0325	0.0348	0.389
FloPoCo-DSP (B)	0.0172	0.0515	0.389
FloPoCo-logic (C)	0.210	0.123	0

Table 4.4: Resources percentage usage of the three variations of floating point multiplication.

of an FPGA before the other resources. Therefore, in our resource allocation algorithm, we only consider the logic, registers and DSPs. Since the number of a certain type of operation within the original or optimised CellML model is fixed, we deal with each operation individually aiming at achieving the minimum usage of each resource, while the differences between the percentage resource usage are minimised.

4.5.3.1 *Formulating the Problem*

Multipliers are used as a case study for the underlying resource allocation techniques, but the same technique apply to the optimisation of other operators. According to the floating point multiplication cores illustrated in Table 4.2 and 4.3, there are three variants of a multiplier. We define the three variants as A - the Altera implementation, B - the FloPoCo implementation with DSPs and C - the FloPoCo implementation with the pure logic. The percentage usage of the three variants are summarised in Table 4.4. Let PL_A, PL_B and PL_C denote the percentage usage of logic for each variant of multiplier. Let PR_A, PR_B and PR_C denote the percentage usage of registers for each variant and PD_A, PD_B and PD_C denote the percentage usage of DSPs for each variant, respectively. In a FPGA design, let N_A, N_B and N_C stand for the number of multipliers implemented as variant A, B, C, respectively for each variant in a CellML model and N stand for the total number of multipliers in the model. Therefore, the following condition must be satisfied:

$$N_A + N_B + N_C = N \tag{4.23}$$

with $N_A, N_B, N_C, N \in \mathbb{N}^0$. The total usage of each resource is:

$$PL = PL_A \cdot N_A + PL_B \cdot N_B + PL_C \cdot N_C \tag{4.24}$$

$$PR = PR_A \cdot N_A + PR_B \cdot N_B + PR_C \cdot N_C \tag{4.25}$$

$$PD = PD_A \cdot N_A + PD_B \cdot N_B + PD_C \cdot N_C \tag{4.26}$$

For the FPGA resource balancing problem, best values for N_A, N_B and N_C need to be determined to minimise the maximum resource usage, P_{max}, which is the potential bottleneck. The maximum resource usage is usually minimised by increasing the usage of other resources and hence the pair-wise gap between each resource usage is minimised to achieve the resource balance purpose. The problem can be expressed as minimising P_{max} in the following expression:

$$P_{max} = \max(PL, PR, PD) \tag{4.27}$$

4.5.3.2 *Exhaustive Algorithm*

The above problem can be naively solved by an exhaustive algorithm. The algorithm enumerates all possible value combinations of N_A, N_B and N_C that adhere to Eq. (4.23), calculates P_{max} for each combination and keeps track of the values that make P_{max} the smallest. This naturally leads to the optimal solution of the problem. The complexity of the algorithm is high when the number of implementation choices, k, is high. It is the number of all possible weak compositions of N into exactly k parts, which is the following binomial coefficient:

$$O(exhaustive) = \begin{pmatrix} N+k-1 \\ k-1 \end{pmatrix} \tag{4.28}$$

For the three alternative implementations considered here the order is:

$$\frac{(N+k-1)!}{(k-1)!((N+k-1)-(k-1))!} = \frac{(N+2)!}{2N!} = \frac{(N+2)(N+1)}{2} = O(N^2) \tag{4.29}$$

Due to the size of typical problems (e.g., in the optimised TNNP model used in the evaluation, $N = 166$) and the limited number of choices, in many cases, the exhaustive algorithm is still adequate for the resource balancing problem.

4.5.3.3 *Multivariate Equations*

The minimised value of P_{max} is required. As said, one way is to minimise the pair-wise difference between each resource usage. Eqs. (4.24 - 4.26) can be reformulated into:

$$PL_A \cdot N_A + PL_B \cdot N_B + PL_C \cdot N_C \approx PR_A \cdot N_A + PR_B \cdot N_B + PR_C \cdot N_C \quad (4.30)$$

$$PR_A \cdot N_A + PR_B \cdot N_B + PR_C \cdot N_C \approx PD_A \cdot N_A + PD_B \cdot N_B + PD_C \cdot N_C \quad (4.31)$$

Eqs. (4.30 and 4.31) together with Eq. (4.23) are simply a ternary linear equation set that can be solved directly. These multivariate equations are easy to solve by hand and there are also many existing computational tools/libraries that can be used to solve these equations, e.g., Matlab [68]. However, one constraint of the equations is that N_A, N_B and N_C should be natural numbers, but the results for the equation set may end up with negative or non-integer values which are not acceptable in this analysis. If any result is negative, it can be replaced with zero and we solve the remaining linear equations. Non-integer results can be simply replaced with the nearest integers.

4.5.3.4 *Greedy Algorithm*

An alternative for this problem is to use an effective greedy algorithm. A greedy algorithm is proposed for the resource balancing problem as illustrated in Algorithm 4.2. In this greedy algorithm, six schemes are defined as illustrated in Table 4.5, to reduce the value of P_{max} in each execution. Table 4.5, used in the greedy algorithm, is created following the rules that (i) variant combinations to increment/decrement are selected to reduce the gap between the resources with maximum and minimum percentage usage, (ii) each combination should occur only once, and (iii) looping situations are avoided, i.e. in one step, NA++; NB-- and in the next step, NB++, NA--. The algorithm

Scheme	Action	Condition
$PL > PD \geqslant PR$	N_B++; N_A- -	$N_A > 0$
$PR \geqslant PL > PD$	N_A++; N_C- -	$N_C > 0$
$PR > PD \geqslant PL$	N_A++; N_B- -	$N_B > 0$
$PD \geqslant PL > PR$	N_A- - ; N_C++	$N_A > 0$
$PL > PR > PD$	N_B++; N_C- -	$N_C > 0$
$PD \geqslant PR \geqslant PL$	N_B- - ; N_C++	$N_B > 0$

Table 4.5: Schemes for reducing PT used in the greedy algorithm.

starts with the selection of the initial conditions by choosing the best values of N_A, N_B, N_C and P_{max} from a set of preselected or randomly generated conditions. In every iteration, the SELECTSCHEME method is called to choose the scheme with $newN_A$, $newN_B$ and $newN_C$ according to the order of the PL, PR and PD, given that the condition of the scheme is satisfied. $newP_{max}$ is then calculated. The $newP_{max}$ is compared with the value of the current P_{max}. If the $newP_{max}$ is smaller, it will continue to update the current P_{max} and N_A, N_B, N_C from new values and perform the next iteration check, otherwise, it reaches the local optimum and the results are returned.

The greedy algorithm is more efficient than the exhaustive algorithm. The worst case complexity is $(k-1) \cdot N$ iterations where initial condition is $N_A = N$ and then all is changed to the condition with $N_B = N$ and then to $N_C = N$ and so forth. The complexity is then at most $O(kN)$, which is much better than the exhaustive algorithm. However, the solution can stop at a local optimum and the optimal solution cannot be guaranteed. Careful selection of initial values can help the algorithm to find high quality solutions, as can be seen in the evaluation of the next section.

There are further alternatives for the resource balancing. For example, an integer linear program (ILP), that could be solved with an ILP solver like CPLEX [33]. However, the proposed algorithms already achieve quite satisfactory performance and prove the concept.

Algorithm 4.2 Greedy algorithm for resource balancing

Require: N
Ensure: $NA + NB + NC = N$ and $NA >= 0$, $NB >= 0$, $NC >= 0$

```
1:  kernel GREEDY(N)
2:      Pmax ← 100
3:      newNA, newNB, newNC, newPmax ← GETSINITIALVALUES(N)
4:      while newPmax < Pmax do
5:          NA ← newNA
6:          NB ← newNB
7:          NC ← newNC
8:          Pmax ← newPmax
9:          newNA, newNB, newNC = SELECTSCHEME(NA, NB, NC)
10:         newPmax ← CALCULATEPMAX(newNA, newNB, newNC)
11:     end whilereturn NA, NB, NC, Pmax
12: end kernel
```

4.5.3.5 *Evaluation of the Resource Balancing Techniques*

To test our techniques, the three techniques/algorithms discussed above were evaluated with 50, 100, 200, 400 and 500 multipliers to determine the proper values for N_A, N_B and N_C so that the maximum individual percentage resource usage is minimised. The results are shown in Table 4.6.

Of the three techniques, the exhaustive algorithm is easy to implement and accurate, but the execution complexity is the highest. The use of multivariate equations is low in complexity, but it is very likely that the results are negative or non-integer values and thus further equation formulation and solution is required. It is also not guaranteed to obtain optimised results and accuracy of the results are highly dependent on the selection of equations. For example, choosing three equations from Eqs. (4.23, 4.30 and 4.31) and $PL \approx PD$. The greedy algorithm has low complexity, however, it depends on the initial values and may end up with a local optimum. In the evaluation, the results of the greedy algorithm are often identical to the exact result ($N = 50$ to 400). For a large problem size ($N = 500$), the greedy algorithm does not work so well, because all the initial conditions chosen by the algorithm are larger than the board capacity (i.e., >100%) and therefore, a new random condition is chosen. In addition, the algorithm is likely to find a local optimum. Since the problem size in the resource fitting/balancing algorithm is often relatively small, the exhaustive algorithm is suggested to ensure the best results.

N	Techniques	N_A	N_B	N_C	$P_{max}(\%)$
50	Exhaustive Algorithm	0	18	32	7.03
	Multivariate Equations	-75/0	91/19	34/31	7.39
	Greedy Algorithm	0	18	32	7.03
100	Exhaustive Algorithm	0	36	64	14.06
	Multivariate Equations	-150/0	-182/37	68/63	14.39
	Greedy Algorithm	0	36	64	14.06
200	Exhaustive Algorithm	0	72	128	28.12
	Multivariate Equations	-300/0	-364/74	136/126	28.79
	Greedy Algorithm	0	72	128	28.12
400	Exhaustive Algorithm	0	144	256	56.24
	Multivariate Equations	-600/0	729/148	271/252	57.57
	Greedy Algorithm	0	144	256	56.24
500	Exhaustive Algorithm	0	180	320	70.30
	Multivariate Equations	-750/0	911/185	339/315	71.97
	Greedy Algorithm	172	13	315	71.965

Table 4.6: Evaluation results for the resource balancing example for a different numbers of multipliers (For multivariate equations method, the results for the first equations set Eqs. (4.23, 4.30 and 4.31) contain negative values where the additional equations set is required).

4.6 MULTIPLE PIPELINES

The generated hardware accelerator module is implemented with a fully pipelined architecture. This architecture approach targets high performance applications, allowing new inputs to be applied with every clock cycle. For large biomedical models that use most of the resources on a FPGA, a single pipeline is sufficient. For small to medium sized biomedical models, the HAMs with a single pipeline only use a fraction of the available resources and the remaining resources remain idle. With multiple pipelines, the performance of the HAM can be easily improved. Multiple pipelines can be implemented in two ways, either by expanding the temporal direction or by replicating in the spatial direction. We name the two methodologies, Extended Pipeline and Parallel Pipeline, respectively.

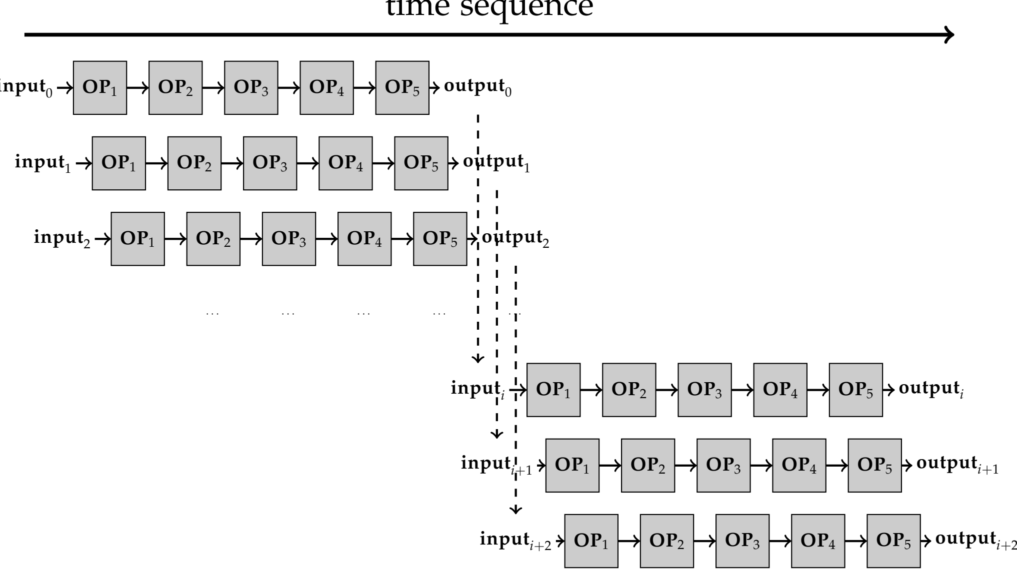


Figure 4.3: Single pipeline flow.

4.6.1 *Single Pipeline*

The single pipeline flow is illustrated in Figure 4.3. As described in Section 4.3, the operations of the complete pipeline correspond to the calculation of all the equations of the model. This calculation needs to be repeated many times for one data item, once for each micro time step. The set of pipelines shown in the figure illustrate that the same pipeline is started on each subsequent data input. Each block in a pipeline represents one operation and with every cycle a new data item enters. Once a data item has passed through the entire pipeline, the output of the last stage is fed back to the input of the pipeline for the next micro time step.

For t micro time steps of cell integration, each input data is iterated through a single pipeline for t times. To complete the entire macro time step integration of one input data set, with a p cycles/stage pipeline, it requires $p \cdot t$ cycles of computation. In the pipeline structure, the maximum number of cells allowed for computation of one data chunk depends on the number of pipeline stages, p. After the first data item completes the entire macro time step integration, it requires another $p - 1$ cycles to finish the whole data chunk (i.e., to drain the pipeline) before the outputs are available to the host. Therefore, the total time used for the computation of p cells is $(p \cdot t + p - 1)$ cycles. Compared to the

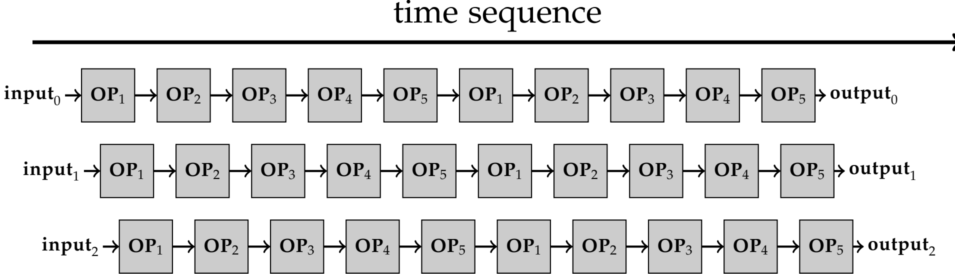


Figure 4.4: Extended pipeline flow.

non-pipelined structure where each cell is computed sequentially $(p \cdot p \cdot t)$, the speedup of the single pipeline is:

$$SpeedUp_{pipe} = \frac{ExecTime_{non-pipe}}{ExecTime_{pipe}} = \frac{p^2 \cdot t}{p \cdot t + p - 1} \tag{4.32}$$

With a 100 stage pipeline of 1000 iterations, the speedup of a pipelined computation against a non-pipelined computation is 99.9. Consequently, filling and draining the pipeline is negligible for the speedup for typical values for the pipeline size and the number of iterations.

4.6.2 *Extended Pipeline*

The numerical integration of differential equations involves repetitive calculations where the same operations are repeated with an integrated data set. The extended pipeline approach expands the pipeline in the temporal direction so that two or more identical single pipelines are joined sequentially to form a long pipeline. Figure 4.4 shows an extended pipeline that joins two single pipelines. The output of the last stage of the first pipeline is the input of the second pipeline, as can be seen in the figure, and so forth for multiple concatenated pipelines.

For a t micro time steps cell integration, each input data is iterated through a single pipeline for t times. Therefore, an extended pipeline that joins n single pipelines reduces the pipeline iterations to t/n (for simplicity assuming here that t is divisible by n). However, the length of the pipeline increases n times so it requires $p \cdot n$ cycles to complete the pipeline. The latency for one cell integ-

ration does not change. But since the pipeline length increases, the maximum number of input cells allowed for computation in one data chunk increases to $p \cdot n$. In other words, once the pipeline is full, the computation and integration for $p \cdot n$ cells is done in parallel. Therefore the total time used for the computation of $p \cdot n$ cells is $(p \cdot t + p \cdot n - 1)$ cycles. Compared to the non-pipelined structure, the speedup of the extended pipeline is then:

$$SpeedUp_{ext-pipe} = \frac{ExecTime_{non-pipe}}{ExecTime_{ext-pipe}} = \frac{p^2 \cdot t \cdot n}{p \cdot t + p \cdot n - 1} \tag{4.33}$$

For two 100 stage single pipelines with 1000 iterations that are joined into one 200 stage extended pipeline, the speedup against a non-pipelined computation is 199.6.

However, since one extended pipeline contains n cell iterations, it is required that the number of iterations to be divisible by n. In order to get the correct results, a more complicated state machine is needed in the controller to obtain the right output from the extended pipeline.

4.6.3 *Parallel Pipelines*

Alternatively, multiple pipelines can be implemented in parallel. Figure 4.5 shows such implementation with two identical parallel executing pipelines represented in different colour. The set of pipelines shown in the same colour indicates that the same pipeline is reused on subsequent input data sets. The two parallel executing pipelines are neither data dependent nor instruction dependent and can be treated as two completely isolated accelerators. They are repeatedly doing the same operations with different input data sets.

Each pipeline in the parallel pipeline structure is executing exactly the same operations as the single pipeline. Since n pipelines are executing in parallel, parallel pipelines can achieve $n\times$ speedup compared to single pipeline. Therefore, its speedup compared to the non-pipelined structure is:

$$SpeedUp_{para-pipe} = \frac{ExecTime_{non-pipe}}{ExecTime_{para-pipe}} = \frac{p^2 \cdot t \cdot n}{p \cdot t + p - 1} \tag{4.34}$$

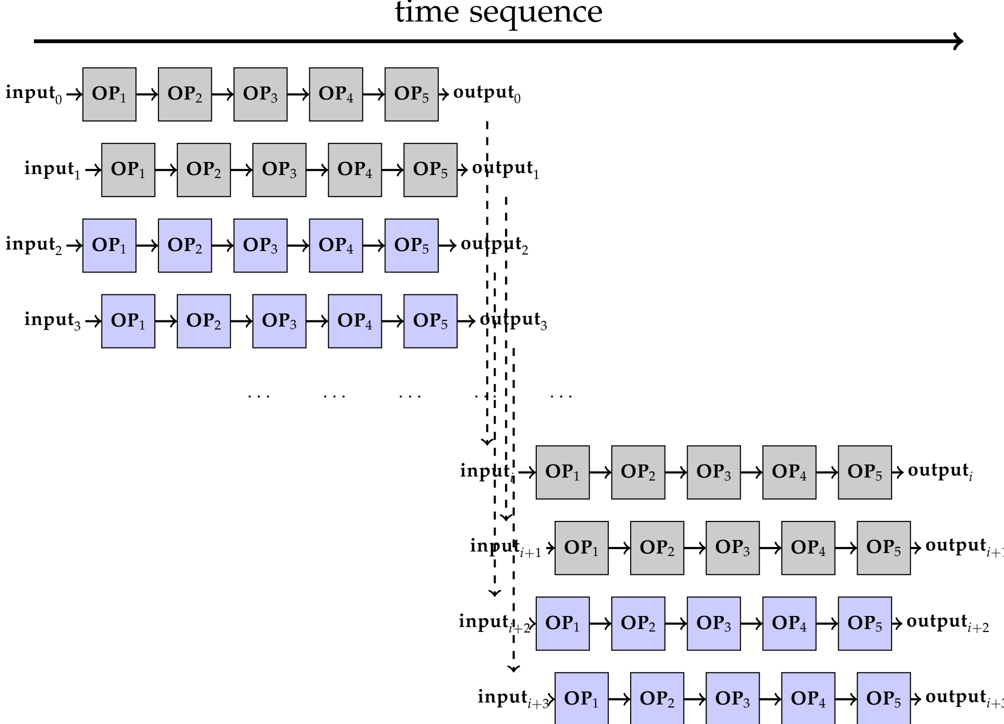


Figure 4.5: Parallel pipeline flow (Different colours represents different pipelines executing in parallel).

For two parallel pipelines each with 100 stages, the speedup with 1000 iterations against a non-pipelined computation is 199.8.

The advantages of parallel pipelines are that they are easy to implement and the same principle can be used across multiple FPGA boards. Therefore, the parallel pipeline is selected for implementation and evaluation.

4.6.4 *Implementation*

The implementation of the parallel pipeline is based on the basic HAM model that is discussed in Section 4.3. The controller and the hardware accelerator are multiplied by n and they are interconnected with the on-chip memory individually. Each controller and hardware accelerator are associated with an ID which determines the chunk of the cells in the on-chip memory that the accelerator deals with and the bits of the control signal the controller corresponds to.

The HDL codes and configurations are generated by ODoST are ready for Qsys [8] integration. The Qsys configuration is then modified by increasing

the on-chip memory size and creating multiple hardware accelerators mapped to the on-chip memory. Each controller/accelerator is operating individually and in parallel with no interaction with other controllers/accelerators. Therefore, the only change the software module needs to determine is when all the accelerators finish their work. To achieve this, individual controller signals are aggregated to a global signal and the software module reads the global signal to receive the computation completion indication.

To automate the process, the ODoST can be configured based on n, the number of pipelines. The templates including the Qsys configuration template and the software module template can be adjusted to suit the parallel pipelines as required.

4.7 EVALUATION

This section undertakes an experimental evaluation of the three proposed optimisation strategies. The experiments use the strategies on two selected biomedical models. The optimisations are used in conjunction with the previously proposed ODoST software that automatically creates the HAMs for the two models. The HAMs and optimisation technologies are assessed according to their resource usage, processing speed and power efficiency. The processing speed and power efficiency are also compared against CPU and GPU implementations of the models.

4.7.1 *Experimental Setup*

Two biomedical models were selected for the evaluation:

- Beeler-Reuter model developed by Beeler and Reuter [21] describing the membrane action potentials of mammalian ventricular myocardial fibres;
- TNNP model developed by Ten Tusscher et al. [96] describing action potentials in human ventricular tissue.

Pipelines	1	2	3
Input (bytes)	80	160	240
Output (bytes)	104	208	312
Addition	49	98	147
Subtraction	34	78	102
Multiplication	60	120	180
Division	28	56	84
Exponential Function	25	50	75
Logarithm	1	2	3
Power Function	1	2	3

Table 4.7: Operations and I/O of Beeler-Reuter models show increasing linearly with the number of pipelines.

The Beeler-Reuter model was selected because it has GPU results available for comparison. The model has low to medium complexity and the auto generated HAM fits well on the DE4 board. In fact, according to previous results, only 33% of the resources (with DSP usage being the highest) are used and the other resources remain idle (Chapter 3). Given these available resources, multiple pipelines were instantiated as discussed in Section 4.6, using the parallel pipeline approach.

The Beeler-Reuter model with two parallel pipelines HAM and three parallel pipelines HAM were evaluated in the experiments and the required operations and I/O of the model are listed in Table 4.7.

The TNNP model was selected due to its high complexity. Indeed, the model is so large that its HAM with the initial ODoST generation does not fit onto the Stratix IV EP4SGX530 board used in the evaluation (Chapter 3). So in this evaluation, the C code equations of the TNNP model are first optimised using the equations optimisation (Section 4.4) and then reformulated with the proposed resource fitting approach (Section 4.5).

Table 4.8 compares the operations and the I/O of the original C code of the model with the optimised C code. As expected, there is no change in the I/O, but there are noticable changes in the number of operations. The optimised code uses more additions, subtractions and multiplication, however it has significantly reduced the much more resource hungry division and power

Model	TNNP (original)	TNNP (optimised)
Input (bytes)	252	252
Output (bytes)	336	336
Addition	114	129
Subtraction	64	91
Multiplication	156	166
Division	129	84
Exponential Function	52	51
Logarithm	4	4
Power Function	26	2

Table 4.8: Operations and I/O of an optimised TNNP model against the original model.

function operations. Essentially, the latter has been replaced in most cases with multiplication, leading to the drop from 26 to only 2 operations. This will significantly reduce the FPGA resource requirements as shown in the next section. As before, ODoST is used to generate the HAM from the optimised C code.

The CPU test platform is an Intel Xeon E5-4650 @2.7 GHz with eight cores and 16 hardware threads [55]. This platform is selected due to its higher core counts and multi-socket capability compared to desktop-grade CPUs. It is a faster CPU than the one used for the host machine in the FPGA test platform. In addition, this system has the Intel compiler suite installed which is one of the faster compilers for x86 and supports comprehensive auto-vectorisation using Streaming SIMD Extensions (SSE). The pure software implementations are compiled with icc 14.0.2 running on a Linux 2.6.32-358 64-bit kernel. For each biomedical model, four software test cases are measured for comparison with the relevant HAM: single thread unoptimised, single thread with SSE optimisation, sixteen threads unoptimised and sixteen threads optimised with SSE.

The results of the Beeler-Reuter Model are also compared to the previous GPU results of Shubhranshu [90]. The GPU test platform that was used was an NVidia Tesla C2070 GPU with 448 Streaming processor cores and 6 GB of GDDR5 memory [73] attached to a system with an Intel Xeon X5650 @2.67 GHZ with 6 cores and 12 GB of DDR3 RAM. Shubhranshu developed an un-

optimised and automated GPU implementation and a hand optimised GPU implementation. The GPU device to host computer transfer rate configured in his experiment was 8 Gb/s.

4.7.2 *Synthesis Results*

In these experiments, we use the Quartus compiler to convert the synthesizable hardware modules, generated by ODoST, into output files for device programming. As before, a script generated by ODoST is used to automate the compilation processes using the Analysis & Synthesis, the Fitter, the Assembler, and the TimeQuest Timing Analyzer modules. The synthesis results are used here to estimate the resource consumption and clock frequency of the HAMs. For completeness, the usage of the Altera FPGA specific ALMs are also included in the resource analysis (see the resource discussion in Section 4.5).

4.7.2.1 *Resource Consumption*

The estimated resource consumption is obtained from the Quartus Fitter. The resources are divided into categories of Logic, Registers, Memory, DSPs and ALMs. The total device capacities are listed in Table 4.1. "Logic" refers to the Combinational ALUTs, "Registers" refers to the Dedicated Registers, "DSPs" refers to the DSP blocks implemented by 18x18 hardware multipliers, "Memory" refers to the memory bits and ALMs refers to the Adaptive Logic Modules.

For the Beeler-Reuter model, the resource consumptions of the HAMs with one, two and three pipelines are represented as a percentage of the total device capacity in Figure 4.6. According to the results, the resource usage grows with the number of pipelines, but not in a strictly linear fashion. Logic, Registers and Memory usage grows slower and only the DSP usage grows in a strictly proportionally manner. This can be explained by the complex relation between ALMs and the other resource categories. Some minor equation reformulation was applied for the three pipelines implementation since the ALM usage did overflow without it, which is very difficult to predict due to the multiple roles of ALMs.

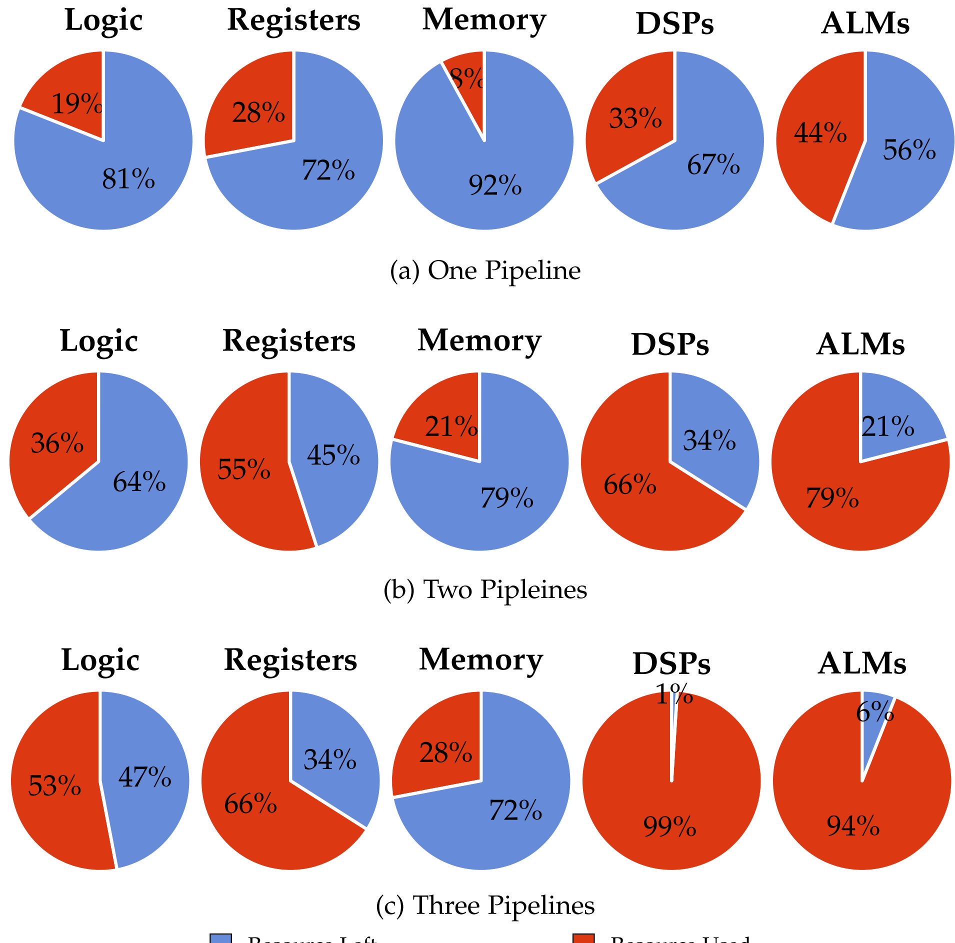


Figure 4.6: Synthesis resource usage results of the HAMs for the Beeler-Reuter model.

Model	TNNP (without balancing)	TNNP (with balancing)
ALUTs	42%	67%
DLRs	82%	88%
DSPs	88%	59%

Table 4.9: Estimated resource consumption of TNNP HAM before and after resource allocation optimisation (Estimates calculated as sums of resource usages for each operation).

For the TNNP model, the resource consumption of the non-optimised and optimised HAMs is illustrated in Figure 4.7. The non-optimised HAM does not fit onto the DE4 board since it uses up all the available DSPs and has an overflow for the ALMs. The optimised HAM for the TNNP model using the proposed equations optimisation, however, fits onto the DE4 board well.

Since the TNNP model with equation optimisation is already sufficient to execute the hardware accelerator on the DE4 board, it is not necessary to perform a further resource balancing optimisation on the model, unless the resource usage could be reduced to under half of the total resource capacity so that parallel pipeline optimisation, i.e., implementing two pipelines, can be used. The optimised TNNP model (i.e., after equations optimisation) was analysed with the exhaustive resource allocation algorithm for multipliers, divisions, exponential functions, logarithms and power functions. As shown by the resource estimation in Table 4.9, although the optimisation with resource allocation algorithm can achieve better resource balance, especially between the logic and DSPs, it is still not possible for two models to fit on the same FPGA. Therefore, resource allocation optimisation is not adopted for the TNNP model.

Interestingly, there are differences between the estimated resource usage and the resource usage in the synthesis reports of the Quartus Synthesiser. The consumption of DLRs and DSPs after synthesis are less than the estimated usage, since the Quartus Synthesiser performs a further resource optimisation step through register and memory packing [2] on the overall HAM. The ALUT consumption after synthesis is more than the estimate because the estimate does not include other logic components such as the controller, the on-chip memory, the DMA and the PCIe IP core.

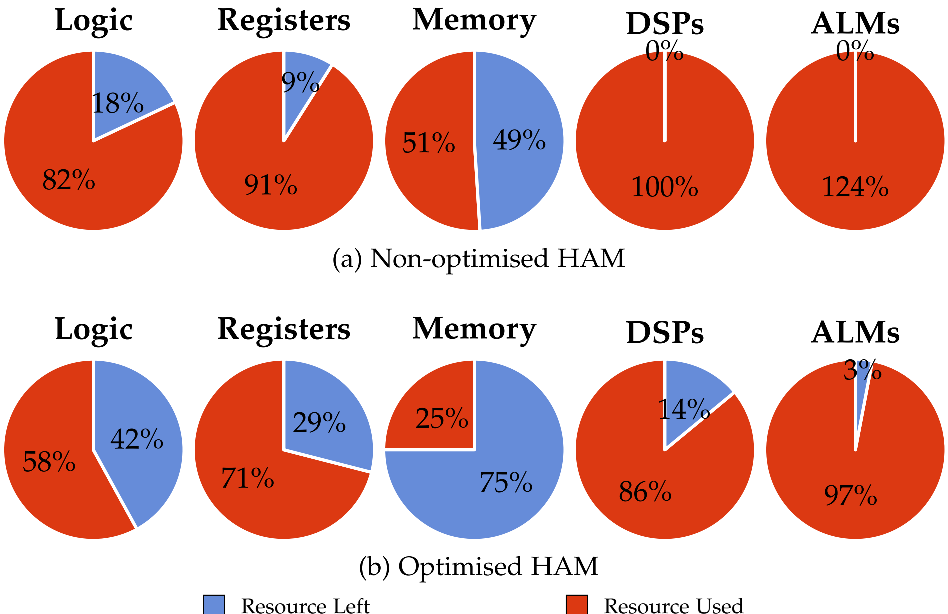


Figure 4.7: Synthesis resource usage results of the non-optimised and optimised HAMs for the TNNP model.

Number of Pipelines	F_{max}(MHz)
1	134.59
2	128.9
3	127.86

Table 4.10: Predicted clock frequencies for the HAMs of the Beeler-Reuter model.

4.7.2.2 *Predicted Clock Frequency*

The predicted maximum clock frequency F_{max} is obtained from the synthesis results performed by Quartus TimeQuest Timing Analyzer. For the design, the operating conditions are set to the slow timing model, with a voltage of 900 mV, and temperature of 85 °C. Table 4.10 displays the frequency values for the different number of pipelines for the Beeler-Reuter model.

The HAMs show good scalability with respect to frequency versus number of pipelines. The frequency used in the implementation is 125 MHz. The predicted F_{max} reaches acceptable frequencies between 125 MHz and 135 MHz with reasonable fall-off for more pipelines. The maximum frequency drop-off is around 5%. This is a very small drop compared to nearly triple of the expected performance increase. With a fully pipelined and parallel design in the HAMs, the throughput for the three pipeline implementation approximates

3 cells/cycle during the entire computation. This is a significant performance advantage compared to the throughput for the single pipeline implementation which is approximately 1 cell/cycle.

The predicted maximum clock frequency for the optimised HAM of the TNNP model is 126.31 MHz. Comparing to the models previously evaluated in Chapter 3, there is a slight fall in F_{max}. This is reasonable and reflects the complexity of the design compared to the others. Using a FPGA at the upper limit of its capacity usually leads to a drop in frequency as the placement of components cannot be fully optimised for speed by the fitter.

4.7.3 *Performance Results*

The performance of the HAMs are presented by their processing speed. For both the Beeler-Reuter model and the TNNP model, the processing speed measures throughput as the number of micro time step cell integrations per second. To simplify, we define the unit $iCells/s$ which stands for iteration cells per second. The results are compared to the CPU implementations of the two models. For the Beeler-Reuter model, the results are also compared against a GPU implementation [90].

Figure 4.8 presents the processing speed of the Beeler-Reuter model across the different implementations. Figure 4.8a presents the throughputs across the implementations in the unit of iCells per second. Figure 4.8b displays speedup against the CPU1 implementation. Each test case measures a biomedical simulation of 1 ms duration with 1 µs micro time step integration for 537,000 cells (number of pipeline stages (179) times maximum capable number of pipelines (3) times 1000). The hand optimised GPU implementation is only used here for a general comparison as all the other implementations in the evaluation are fully automated (or can be fully automated).

As shown in the figures, the two pipeline implementation achieves 1.91 speedup and the three pipeline implementation achieves 2.71 speedup compared to the single pipeline implementation, hence the results are within 10% of the theoretical optimal value. This is a reflection of the increase in the commu-

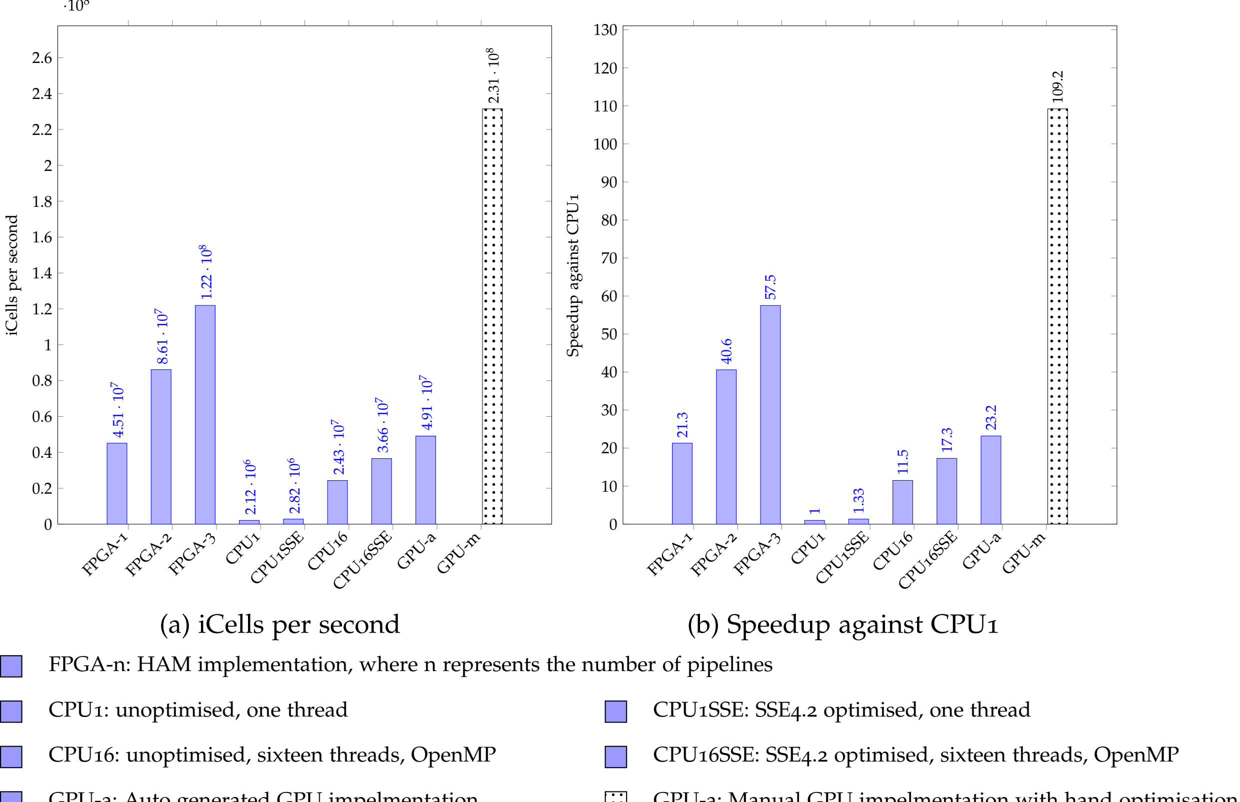


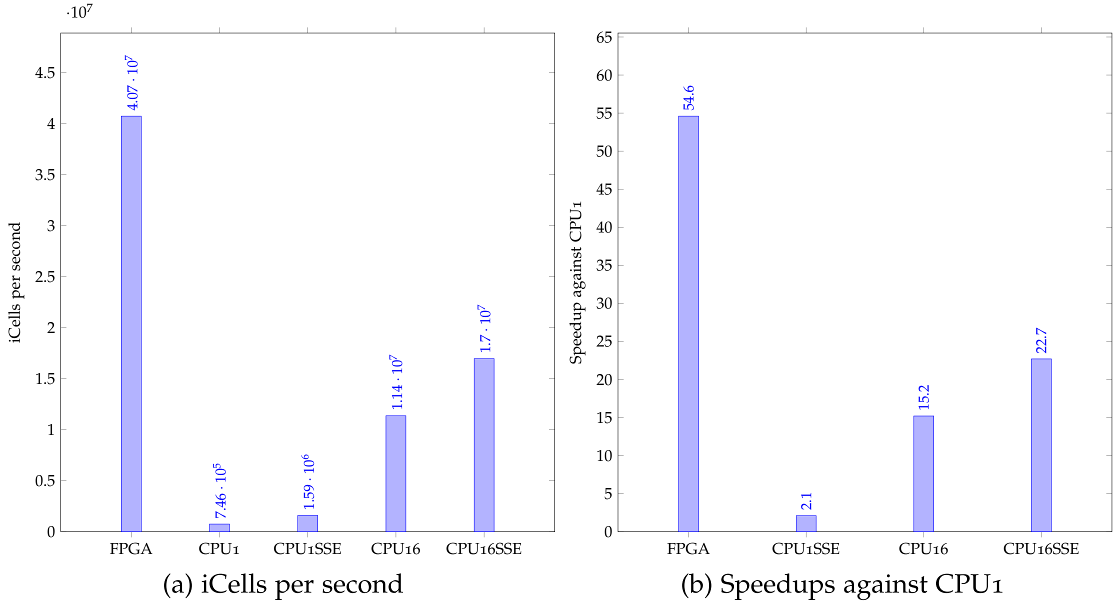
Figure 4.8: Processing speed of the HAMs compared to the CPU and GPU implementations for the Beeler-Reuter model (the bar with dotted pattern represents the hand optimised GPU implementation where all the other implementations in the evaluation are fully automated or can be fully automated).

nication overhead since, although the pipelines are executing in parallel, the data transfer is still in serial and a n-pipelines computation requires n times of data to be transferred. The three pipeline HAM implementation with the most resource utilisation displays a great performance advantage compared to the CPU implementations (57.5x speedup) and automated GPU implementation (2.5x speedup). It reaches just over half of the processing speed compared to the hand optimised GPU implementation.

Figure 4.9 presents the processing speed of the TNNP model on the FPGA and CPU platforms. Each test case measures a biomedical simulation of 1 ms duration with 1 μs micro time step integration for 364,000 cells (number of pipeline stages (364) times 1000). FPGA denotes the results for the HAM implementation with the other notations as before. Figure 4.9a presents the throughput across the implementations in the unit of iCells per second. Figure 4.9b displays speedup against the CPU1 implementation. The HAM implementation has significant performance advantage over all the CPU implementations with nearly a 55x speedup compared to the single threaded unoptimised implementation, 26x speedup compared to the single threaded implementation with SSE optimisation, 3.6x speedup compared to the sixteen threaded unoptimised implementation and 2.4x speedup compared to the sixteen threaded implementation with SSE optimisation.

4.7.4 *Power Efficiency*

For the Beeler-Reuter model power efficiency is compared between the HAMs, the best performing CPU implementation (CPU16SSE) and the CUDA-based GPU implementations. The power requirements for the three testing platforms are shown in Table 4.11. Since resources are not fully consumed in the FPGA, the FPGA power usage is estimated by Altera's PowerPlay Power Analyser. The PowerPlay Power Analyser supports accurate power estimations and is executed at the post-fit phase of the design cycle. The estimated power requirement for the three HAM implementations are 15 W, 19.2 W and 24.1 W respectively. The power usage increases with increased resource consumption.



(a) iCells per second

(b) Speedups against CPU1

Figure 4.9: Processing speed of the HAMs compare to the CPU implementations for the TNNP model.

For the triple pipelined HAM implementation, both the ALMs and the DSPs are approaching the resource capacity. A power estimation of 24.1 W is close to 25 W, the maximum power consumption allowed for a x8 PCI Express card [89]. To allow a fair comparison with the CPU and GPU, we assume the worst case for the FPGA and specify the device power characteristics to maximum and junction temperature to the maximum. The CPU power usage is estimated at 130 W and the GPU power usage is estimated at 238 W, both using the Thermal Design Power (TDP). The TDP of a device is the maximum amount of heat generated by the device that the cooling system is required to dissipate in a typical operation [80]. The TDP should be a good estimate for power consumption for the CPU during cell computation and integration, because the repeated use of SIMD instructions usually employs the CPU at the TDP limit [56]. For the GPU, the hand optimised implementation is likely to work at the TDP limit. For auto generated GPU implementation the power consumption estimate might be less accurate. For that reason, the hand optimised GPU version is also included in the comparison, even though all other implementations are mostly generated automatically.

The power efficiency of the Beeler-Reuter model on each platform is measured by the processing speed obtained from Figure 4.8 divided by the power

Testing Platform		Power Measurement (W)	Measurement Basis
Stratix IV EP4SGX530	One Pipeline	15	PowerPlay Power Analyzer
	Two Pipelines	19.2	
	Three Pipelines	24.1	
Xeon E5-4650		130	Thermal Design Power
Tesla C2070		238	Thermal Design Power

Table 4.11: Power requirement for the Beeler-Reuter model on the three testing platforms.

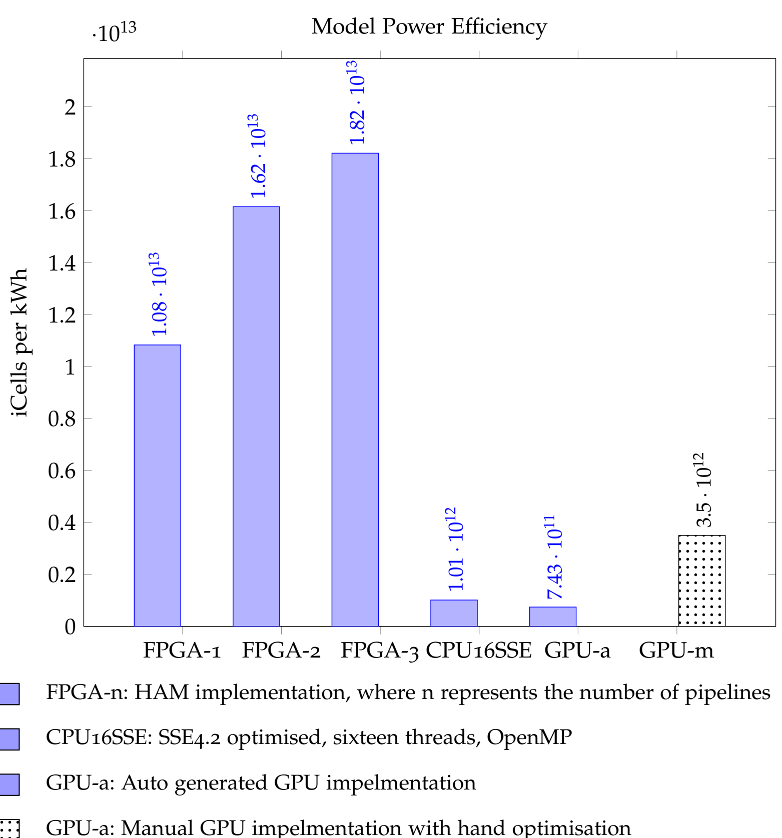


Figure 4.10: Power consumption of the HAM, CPU and GPU implementations for the Beeler-Reuter model (the bar with dotted pattern represents the hand optimised GPU implementation where all the other implementations in the evaluation are fully automated or can be fully automated).

Testing Platform	Power Measurement (W)	Measurement Basis
Stratix IV EP4SGX530	25	Maximum Power Consumption through x8 PCIe
Xeon E5-4650	130	Thermal Design Power

Table 4.12: Power requirement for the TNNP model on the two testing platforms.

requirement for each type of implementation from Table 4.11. The resulting values in iCells per watt second are converted to iCells per kWh and are presented in Figure 4.10. The results show that across the three FPGA implementations, the triple pipeline HAM implementation is the most power efficient. Although Table 4.11 shows that the power usage increases with increased resource consumption, its growth rate does not compete with the performance increase. Therefore, there is still a trend of improved power efficiency with an increasing number of pipelines, but obviously not as much as the improvement of processing speed. Compared to the CPU16SSE and GPU implementations, the FPGA implementations show significantly better power efficiency. The triple pipelined HAM implementation is 18x more power efficient than the CPU16SSE implementation and is still 5.2x more power efficient than the hand optimised GPU implementation, despite the non-automatic nature of this implementation.

For the TNNP model, power efficiency is compared between the HAM and the best performing CPU implementation (CPU16SSE). The power requirement for the two testing platforms is shown in Table 4.12. Since both the DSPs and ALMs are approaching the resource capacity of the FPGA, the FPGA power usage is specified to 25 W, the maximum power consumption allowed through a x8 PCI Express card [89]. The CPU power usage is estimated at 130 W using the TDP.

Again, the power efficiency of the TNNP model on the FPGA and CPU platforms is measured from the processing speed obtained from Figure 4.9 di-

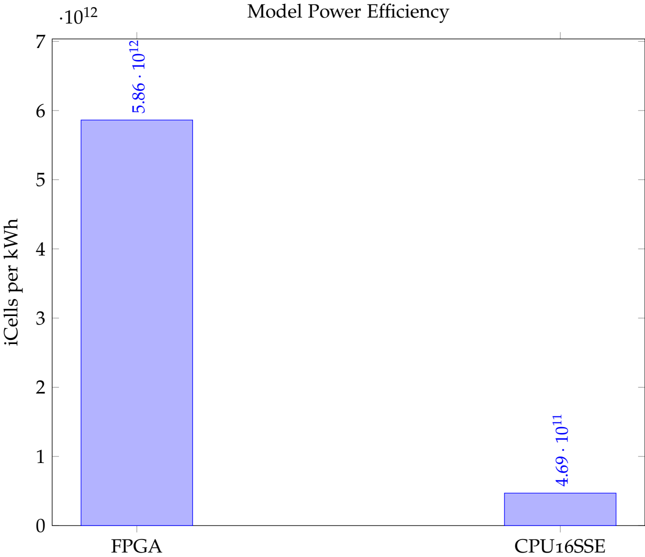


Figure 4.11: Power consumption of the HAM and CPU implementations for the TNNP model.

vided by the power requirement for each device/model from Table 4.12. The resulting values in iCells per kWh are presented in Figure 4.11. The results demonstrate that the HAM implementation is 12.5x more power efficient than the CPU16SSE implementation, whilst it also outperformed the CPU16SSE implementation by more than a factor of two.

4.8 CONCLUSIONS

This chapter proposes a set of optimisation strategies aimed at reducing resource consumption and increasing performance for the hardware acceleration modules that are generated by ODoST. The strategies are diverse and address the high-level synthesis process at different points: optimising the input, optimising the resource consumption and replicating modules for a better utilisation of the FPGAs. These strategies are all suitable for automatic high-level synthesis and integrate well into ODoST. After studying the various optimisation approaches, this chapter evaluates the optimised hardware accelerator modules for two biomedical CellML models. The results demonstrate that the optimised HAMs with parallel pipelines can provide significant improvements in processing performance and energy efficiency. Apart from the performance improvements, the optimisations are also useful to fit larger CellML models

onto a FPGA device. In the future, further optimisations have the potential to improve the performance, e.g., overlapping communication with communication and advancing the compiler and resource fitting optimisations.

5 CONCLUSIONS

Biomedical applications involving large scale simulations requiring heavy computation are generally limited by the available computational hardware and the acceptable duration of simulation. These large scale simulations usually contain portions of code that are evaluated a very large number of times and which contribute significantly to the overall computational runtime. These portions are, often in general, regular and easy to parallelise. As such, FPGAs with large amount of fine-grained parallelism, have promise for accelerating these type of simulations and can be expected to lead to higher performance, at a lower cost and with less power consumption.

However, compared to multicore processors and GPUs, FPGAs are not widely used by biomedical scientists and engineers possibly due to their lack of hardware expertise. Developing a hardware design for a given application is much more challenging than programming general purpose processors. It is all but trivial to combine general purpose processors with the reconfigurable computing capacity of the FPGAs. Furthermore, FPGAs have limited usable area, which create difficulties in implementing large size biomedical models. Hence, designs need to be optimized for size to be implementable on a FPGA with limited number of resources.

This thesis investigated and developed a hardware accelerator with a hardware/software co-design system especially designed for biomedical models. The pipeline based accelerator provides a general and feasible framework that can be applied to a range of applications that would benefit from acceleration.

Preliminary evaluation results from the manually implemented hardware accelerator module (HAM) showed good scalability and performance speedup compared to a pure software implementations. This performance improvement could have a benefit if the time and effort to implement and debug the accelerator could be significantly reduced.

Based on these early results, the thesis has advanced along two facets: 1) automatic generation of hardware accelerators from a high-level description of biomedical models, and 2) accelerator optimisation strategies to fully utilise the resources in a FPGA. The two facets are integrated together to provide a packaged solution to easily create high performance hardware accelerators for biomedical scientists or engineers without hardware design expertise.

AUTOMATIC GENERATION OF HARDWARE ACCELERATORS An ODE-based Domain-specific Synthesis Tool, ODoST, was implemented and used to generate the software/hardware co-design of the accelerator from the high-level description of a biomedical model. The design is general, flexible and capable over a large range of biomedical models. Using a set of CellML models with diverse complexity as case studies, the ODoST has generated the corresponding HAMs that have been thoroughly tested and evaluated. The results show that FPGAs can provide a highly power efficient solution with remarkable processing performance compared to both multicore processors and GPUs.

The generated HAMs, despite significant speedups, were limited in scalability by the amount of available hardware resources. Accelerators for complex biomedical models may not fit well into a FPGA device. While the ultimate solution would be to use either a larger FPGA board or multiple FPGA boards attached to the host, some alternative strategies may assist in better utilising existing resources in the target FPGA.

ACCELERATOR OPTIMISATIONS Optimisation strategies aimed at improving performance and usability of the generated HAMs have been proposed in the thesis. The strategies, including compiler optimisation, resource balancing

and parallel pipelining, address the high-level synthesis process at different points: optimising the input, optimising the resource consumptions and replicating modules for a better utilisation of the FPGAs. While these strategies are diverse in nature, they are all suitable for automatic high-level synthesis and integrate well into ODoST. The optimised HAMs are implemented and evaluated and the results demonstrate that the optimised HAMs can provide significant improvements in processing performance and power efficiency as well as relieving the capacity limits of a FPGA device to fit larger models.

Future Avenues of Research

As demonstrated in this thesis, FPGAs show great potential as hardware accelerators for biomedical modelling and simulation. The presented hardware accelerator design and the high-level synthesis tool will help to give biomedical scientists and engineers the ease of adopting FPGAs in order to obtain better performance and less power consumption. This research provides foundation for future research. Key areas for further investigation include:

- Multiple devices. The optimisation strategies discussed in the thesis enable some large models to be usable with our target FPGA. However, larger models may require much more resource capacity. One solution is upgrading to a more powerful FPGA board. An alternative solution is using multiple FPGA boards attached to the same host through different PCIe ports. CellML-based biomedical models can be divided into components and allocated to those boards. On the other hand, performance of a model can be further improved with more parallel pipelines using multiple FPGA boards. The partitioning and the interaction between the components needs to be investigated.

- Multiple models. The existing HAM with a hardware/software co-design structure is suitable for biomedical simulations with a single model. Extending the current accelerator module to support multiple CellML-based models is one of the future directions to solve some coupled problems.

- Multiple levels of precision. Single precision floating point numbers are used throughout this thesis. Single Precision is fast and area efficient but, using double precision computing for floating point arithmetic operations provides higher accuracy at the expense of more resources. Benefiting from multiple device acceleration, double precision support is one of the future areas of research.
- Overlapped communication and computation. Overlapped communication through the PCIe interconnects and computation within the FPGA was not considered in this thesis. This can be explored to propose a generic way to handle high-bandwidth data exchange.

EXAMPLE CELLML MODELS

A.1 HODGKIN-HUXLEY MODEL

A.1.1 *Mathematics*

"environment" component

This component has no equations.

"membrane" component

$$i_Stim = \begin{cases} -20 & \text{if } (time \geq 10) \land (time \leq 10.5) \\ 0 & \text{otherwise.} \end{cases}$$

$$\frac{\mathrm{d}(V)}{\mathrm{d}(time)} = \frac{-((-(i_Stim) + i_Na + i_K + i_L))}{Cm}$$

"sodium_channel" component

$$E_Na = (E_R - 115)$$

$$i_Na = g_Na * (m)^3 * h * (V - E_Na)$$

"sodium_channel_m_gate" component

$$alpha_m = \frac{0.1 * (V + 25)}{\left(e^{\frac{V+25}{10}} - 1\right)}$$

$$beta_m = 4 * e^{\frac{V}{18}}$$

$$\frac{\mathrm{d}(m)}{\mathrm{d}(time)} = (alpha_m * (1 - m) - beta_m * m)$$

"sodium_channel_h_gate" component

$$alpha_h = 0.07 * e^{\frac{V}{20}}$$

$$beta_h = \frac{1}{\left(e^{\frac{V+30}{10}} + 1\right)}$$

$$\frac{\mathrm{d}(h)}{\mathrm{d}(time)} = (alpha_h * (1 - h) - beta_h * h)$$

"potassium_channel" component

$$E_K = E_R + 12$$

$$i_K = g_K * n^4 * (V - E_K)$$

"potassium_channel_n_gate" component

$$alpha_n = \frac{0.01 * (V + 10)}{\left(e^{\frac{V+10}{10}} - 1\right)}$$

$$beta_n = 0.125 * e^{\frac{V}{80}}$$

$$\frac{\mathrm{d}(n)}{\mathrm{d}(time)} = (alpha_n * (1 - n) - beta_n * n)$$

"leakage_current" component

$$E_L = (E_R - 10.613)$$

$$i_L = g_L * (V - E_L)$$

A.1.2 *C-code Representation*

```
/*
   There are a total of 10 entries in the algebraic variable array.
   There are a total of 4 entries in each of the rate and state variable arrays.
   There are a total of 8 entries in the constant variable array.
 */
/*
 * VOI is time in component environment (millisecond).
 * STATES[0] is V in component membrane (millivolt).
 * CONSTANTS[0] is E_R in component membrane (millivolt).
 * CONSTANTS[1] is Cm in component membrane (microF_per_cm2).
 * ALGEBRAIC[4] is i_Na in component sodium_channel (microA_per_cm2).
 * ALGEBRAIC[8] is i_K in component potassium_channel (microA_per_cm2).
 * ALGEBRAIC[9] is i_L in component leakage_current (microA_per_cm2).
 * ALGEBRAIC[0] is i_Stim in component membrane (microA_per_cm2).
 * CONSTANTS[2] is g_Na in component sodium_channel (milliS_per_cm2).
 * CONSTANTS[5] is E_Na in component sodium_channel (millivolt).
 * STATES[1] is m in component sodium_channel_m_gate (dimensionless).
 * STATES[2] is h in component sodium_channel_h_gate (dimensionless).
 * ALGEBRAIC[1] is alpha_m in component sodium_channel_m_gate (per_millisecond).
 * ALGEBRAIC[5] is beta_m in component sodium_channel_m_gate (per_millisecond).
 * ALGEBRAIC[2] is alpha_h in component sodium_channel_h_gate (per_millisecond).
 * ALGEBRAIC[6] is beta_h in component sodium_channel_h_gate (per_millisecond).
 * CONSTANTS[3] is g_K in component potassium_channel (milliS_per_cm2).
 * CONSTANTS[6] is E_K in component potassium_channel (millivolt).
 * STATES[3] is n in component potassium_channel_n_gate (dimensionless).
 * ALGEBRAIC[3] is alpha_n in component potassium_channel_n_gate (per_millisecond).
 * ALGEBRAIC[7] is beta_n in component potassium_channel_n_gate (per_millisecond).
 * CONSTANTS[4] is g_L in component leakage_current (milliS_per_cm2).
 * CONSTANTS[7] is E_L in component leakage_current (millivolt).
 * RATES[0] is d/dt V in component membrane (millivolt).
 * RATES[1] is d/dt m in component sodium_channel_m_gate (dimensionless).
 * RATES[2] is d/dt h in component sodium_channel_h_gate (dimensionless).
 * RATES[3] is d/dt n in component potassium_channel_n_gate (dimensionless).
 */
void
initConsts(double* CONSTANTS, double* RATES, double *STATES)
{
STATES[0] = -75;
CONSTANTS[0] = -75;
CONSTANTS[1] = 1;
CONSTANTS[2] = 120;
STATES[1] = 0.05;
STATES[2] = 0.6;
CONSTANTS[3] = 36;
STATES[3] = 0.325;
CONSTANTS[4] = 0.3;
```

```
47 CONSTANTS[5] = CONSTANTS[0]+115.000;
48 CONSTANTS[6] = CONSTANTS[0] − 12.0000;
49 CONSTANTS[7] = CONSTANTS[0]+10.6130;
50 }
51 void
52 computeRates(double VOI, double* CONSTANTS, double* RATES, double* STATES, double*
       ALGEBRAIC)
53 {
54 ALGEBRAIC[1] = − 0.100000*(STATES[0]+50.0000)/(exp(− (STATES[0]+50.0000)/10.0000) −
       1.00000);
55 ALGEBRAIC[5] = 4.00000*exp(− (STATES[0]+75.0000)/18.0000);
56 RATES[1] = ALGEBRAIC[1]*(1.00000 − STATES[1]) − ALGEBRAIC[5]*STATES[1];
57 ALGEBRAIC[2] = 0.0700000*exp(− (STATES[0]+75.0000)/20.0000);
58 ALGEBRAIC[6] = 1.00000/(exp(− (STATES[0]+45.0000)/10.0000)+1.00000);
59 RATES[2] = ALGEBRAIC[2]*(1.00000 − STATES[2]) − ALGEBRAIC[6]*STATES[2];
60 ALGEBRAIC[3] = − 0.0100000*(STATES[0]+65.0000)/(exp(− (STATES[0]+65.0000)/10.0000) −
       1.00000);
61 ALGEBRAIC[7] = 0.125000*exp((STATES[0]+75.0000)/80.0000);
62 RATES[3] = ALGEBRAIC[3]*(1.00000 − STATES[3]) − ALGEBRAIC[7]*STATES[3];
63 ALGEBRAIC[4] = CONSTANTS[2]*pow(STATES[1], 3.00000)*STATES[2]*(STATES[0] − CONSTANTS
       [5]);
64 ALGEBRAIC[8] = CONSTANTS[3]*pow(STATES[3], 4.00000)*(STATES[0] − CONSTANTS[6]);
65 ALGEBRAIC[9] = CONSTANTS[4]*(STATES[0] − CONSTANTS[7]);
66 ALGEBRAIC[0] = (VOI>=10.0000&&VOI<=10.5000 ? 20.0000 : 0.00000);
67 RATES[0] = − (− ALGEBRAIC[0]+ALGEBRAIC[4]+ALGEBRAIC[8]+ALGEBRAIC[9])/CONSTANTS[1];
68 }
69 void
70 computeVariables(double VOI, double* CONSTANTS, double* RATES, double* STATES, double*
       ALGEBRAIC)
71 {
72 ALGEBRAIC[1] = − 0.100000*(STATES[0]+50.0000)/(exp(− (STATES[0]+50.0000)/10.0000) −
       1.00000);
73 ALGEBRAIC[5] = 4.00000*exp(− (STATES[0]+75.0000)/18.0000);
74 ALGEBRAIC[2] = 0.0700000*exp(− (STATES[0]+75.0000)/20.0000);
75 ALGEBRAIC[6] = 1.00000/(exp(− (STATES[0]+45.0000)/10.0000)+1.00000);
76 ALGEBRAIC[3] = − 0.0100000*(STATES[0]+65.0000)/(exp(− (STATES[0]+65.0000)/10.0000) −
       1.00000);
77 ALGEBRAIC[7] = 0.125000*exp((STATES[0]+75.0000)/80.0000);
78 ALGEBRAIC[4] = CONSTANTS[2]*pow(STATES[1], 3.00000)*STATES[2]*(STATES[0] − CONSTANTS
       [5]);
79 ALGEBRAIC[8] = CONSTANTS[3]*pow(STATES[3], 4.00000)*(STATES[0] − CONSTANTS[6]);
80 ALGEBRAIC[9] = CONSTANTS[4]*(STATES[0] − CONSTANTS[7]);
81 ALGEBRAIC[0] = (VOI>=10.0000&&VOI<=10.5000 ? 20.0000 : 0.00000);
82 }
```

A.2 BEELER-REUTER MODEL

A.2.1 *Mathematics*

"environment" component

This component has no equations.

"membrane" component

$$\frac{d(V)}{d(time)} = \frac{(Istim - (i_Na + i_s + i_x1 + i_K1))}{C}$$

"sodium_current" component

$$i_Na = \left(g_Na * (m)^3 * h * j + g_Nac\right) * (V - E_Na)$$

"sodium_current_m_gate" component

$$alpha_m = \frac{-(1) * (V + 47)}{\left(e^{-(0.1)*(V+47)} - 1\right)}$$

$$beta_m = 40 * e^{-(0.056)*(V+72)}$$

$$\frac{d(m)}{d(time)} = (alpha_m * (1 - m) - beta_m * m)$$

"sodium_current_h_gate" component

$$alpha_h = 0.126 * e^{-(0.25)*(V+77)}$$

$$beta_h = \frac{1.7}{\left(e^{-(0.082)*(V+22.5)} + 1\right)}$$

$$\frac{d(h)}{d(time)} = (alpha_h * (1 - h) - beta_h * h)$$

"sodium_current_j_gate" component

$$alpha_j = \frac{0.055 * e^{-(0.25)*(V+78)}}{\left(e^{-(0.2)*(V+78)} + 1\right)}$$

$$beta_j = \frac{0.3}{\left(e^{-(0.1)*(V+32)} + 1\right)}$$

$$\frac{d(j)}{d(time)} = (alpha_j * (1 - j) - beta_j * j)$$

"slow_inward_current" component

$$E_s = (-(82.3) - 13.0287 * \ln(Cai * 0.001))$$

$$i_s = g_s * d * f * (V - E_s)$$

$$\frac{d(Cai)}{d(time)} = \left(\frac{-(0.01) * i_s}{1} + 0.07 * (0.0001 - Cai)\right)$$

"slow_inward_current_d_gate" component

$$alpha_d = \frac{0.095 * e^{\frac{-((V-5))}{100}}}{\left(1 + e^{\frac{-((V-5))}{13.89}}\right)}$$

$$beta_d = \frac{0.07 * e^{\frac{-((V+44))}{59}}}{\left(1 + e^{\frac{(V+44)}{20}}\right)}$$

$$\frac{d(d)}{d(time)} = (alpha_d * (1 - d) - beta_d * d)$$

"slow_inward_current_f_gate" component

$$alpha_f = \frac{0.012 * e^{\frac{-((V+28))}{125}}}{\left(1 + e^{\frac{(V+28)}{6.67}}\right)}$$

$$beta_f = \frac{0.0065 * e^{\frac{-((V+30))}{50}}}{\left(1 + e^{\frac{-((V+30))}{5}}\right)}$$

$$\frac{d(f)}{d(time)} = (alpha_f * (1 - f) - beta_f * f)$$

"time_dependent_outward_current" component

$$i_x1 = \frac{x1 * 8 - 3 * \left(e^{0.04*(V+77)} - 1\right)}{e^{0.04*(V+35)}}$$

"time_dependent_outward_current_x1_gate" component

$$alpha_x1 = \frac{5 - 4 * e^{\frac{(V+50)}{12.1}}}{\left(1 + e^{\frac{(V+50)}{17.5}}\right)}$$

$$beta_x1 = \frac{0.0013 * e^{\frac{-((V+20))}{16.67}}}{\left(1 + e^{\frac{-((V+20))}{25}}\right)}$$

$$\frac{d(x1)}{d(time)} = (alpha_x1 * (1 - x1) - beta_x1 * x1)$$

"time_independent_outward_current" component

$$i_K1 = 0.0035 * \left(\frac{4 * \left(e^{0.04*(V+85)} - 1\right)}{\left(e^{0.08*(V+53)} + e^{0.04*(V+53)}\right)} + \frac{0.2 * (V + 23)}{\left(1 - e^{-(0.04)*(V+23)}\right)}\right)$$

"stimulus_protocol" component

$$Istim = \begin{cases} IstimAmplitude & \text{if } (time \geq IstimStart) \wedge (time \leq IstimEnd) \wedge time \\ & -IstimStart - \left(\left\lfloor \frac{time - IstimStart}{IstimPeriod} \right\rfloor IstimPeriod\right) \\ & \leqq IstimPulseDuration \\ 0 & \text{otherwise.} \end{cases}$$

A.2.2 C-code Representation

```
/*
   There are a total of 18 entries in the algebraic variable array.
   There are a total of 8 entries in each of the rate and state variable arrays.
   There are a total of 10 entries in the constant variable array.
 */
/*
 * VOI is time in component environment (ms).
 * STATES[0] is V in component membrane (mV).
 * CONSTANTS[0] is C in component membrane (uF_per_mm2).
 * ALGEBRAIC[0] is i_Na in component sodium_current (uA_per_mm2).
 * ALGEBRAIC[14] is i_s in component slow_inward_current (uA_per_mm2).
 * ALGEBRAIC[15] is i_x1 in component time_dependent_outward_current (uA_per_mm2).
 * ALGEBRAIC[16] is i_K1 in component time_independent_outward_current (uA_per_mm2).
 * ALGEBRAIC[17] is Istim in component stimulus_protocol (uA_per_mm2).
```

```
15  * CONSTANTS[1] is g_Na in component sodium_current (mS_per_mm2).
16  * CONSTANTS[2] is E_Na in component sodium_current (mV).
17  * CONSTANTS[3] is g_Nac in component sodium_current (mS_per_mm2).
18  * STATES[1] is m in component sodium_current_m_gate (dimensionless).
19  * STATES[2] is h in component sodium_current_h_gate (dimensionless).
20  * STATES[3] is j in component sodium_current_j_gate (dimensionless).
21  * ALGEBRAIC[1] is alpha_m in component sodium_current_m_gate (per_ms).
22  * ALGEBRAIC[8] is beta_m in component sodium_current_m_gate (per_ms).
23  * ALGEBRAIC[2] is alpha_h in component sodium_current_h_gate (per_ms).
24  * ALGEBRAIC[9] is beta_h in component sodium_current_h_gate (per_ms).
25  * ALGEBRAIC[3] is alpha_j in component sodium_current_j_gate (per_ms).
26  * ALGEBRAIC[10] is beta_j in component sodium_current_j_gate (per_ms).
27  * CONSTANTS[4] is g_s in component slow_inward_current (mS_per_mm2).
28  * ALGEBRAIC[7] is E_s in component slow_inward_current (mV).
29  * STATES[4] is Cai in component slow_inward_current (concentration_units).
30  * STATES[5] is d in component slow_inward_current_d_gate (dimensionless).
31  * STATES[6] is f in component slow_inward_current_f_gate (dimensionless).
32  * ALGEBRAIC[4] is alpha_d in component slow_inward_current_d_gate (per_ms).
33  * ALGEBRAIC[11] is beta_d in component slow_inward_current_d_gate (per_ms).
34  * ALGEBRAIC[5] is alpha_f in component slow_inward_current_f_gate (per_ms).
35  * ALGEBRAIC[12] is beta_f in component slow_inward_current_f_gate (per_ms).
36  * STATES[7] is x1 in component time_dependent_outward_current_x1_gate (dimensionless)
37  * ALGEBRAIC[6] is alpha_x1 in component time_dependent_outward_current_x1_gate (
      per_ms).
38  * ALGEBRAIC[13] is beta_x1 in component time_dependent_outward_current_x1_gate (
      per_ms).
39  * CONSTANTS[5] is IstimStart in component stimulus_protocol (ms).
40  * CONSTANTS[6] is IstimEnd in component stimulus_protocol (ms).
41  * CONSTANTS[7] is IstimAmplitude in component stimulus_protocol (uA_per_mm2).
42  * CONSTANTS[8] is IstimPeriod in component stimulus_protocol (ms).
43  * CONSTANTS[9] is IstimPulseDuration in component stimulus_protocol (ms).
44  * RATES[0] is d/dt V in component membrane (mV).
45  * RATES[1] is d/dt m in component sodium_current_m_gate (dimensionless).
46  * RATES[2] is d/dt h in component sodium_current_h_gate (dimensionless).
47  * RATES[3] is d/dt j in component sodium_current_j_gate (dimensionless).
48  * RATES[4] is d/dt Cai in component slow_inward_current (concentration_units).
49  * RATES[5] is d/dt d in component slow_inward_current_d_gate (dimensionless).
50  * RATES[6] is d/dt f in component slow_inward_current_f_gate (dimensionless).
51  * RATES[7] is d/dt x1 in component time_dependent_outward_current_x1_gate (
      dimensionless).
52  */
53 void
54 initConsts(double* CONSTANTS, double* RATES, double *STATES)
55 {
56 STATES[0] = -84.624;
57 CONSTANTS[0] = 0.01;
58 CONSTANTS[1] = 4e-2;
59 CONSTANTS[2] = 50;
60 CONSTANTS[3] = 3e-5;
```

```
61 STATES[1] = 0.011;
62 STATES[2] = 0.988;
63 STATES[3] = 0.975;
64 CONSTANTS[4] = 9e−4;
65 STATES[4] = 1e−4;
66 STATES[5] = 0.003;
67 STATES[6] = 0.994;
68 STATES[7] = 0.0001;
69 CONSTANTS[5] = 10;
70 CONSTANTS[6] = 50000;
71 CONSTANTS[7] = 0.5;
72 CONSTANTS[8] = 1000;
73 CONSTANTS[9] = 1;
74 }
75 void
76 computeRates(double VOI, double* CONSTANTS, double* RATES, double* STATES, double*
       ALGEBRAIC)
77 {
78 ALGEBRAIC[1] = − 1.00000*(STATES[0]+47.0000)/(exp( − 0.100000*(STATES[0]+47.0000)) −
       1.00000);
79 ALGEBRAIC[8] = 40.0000*exp( − 0.0560000*(STATES[0]+72.0000));
80 RATES[1] = ALGEBRAIC[1]*(1.00000 − STATES[1]) − ALGEBRAIC[8]*STATES[1];
81 ALGEBRAIC[2] = 0.126000*exp( − 0.250000*(STATES[0]+77.0000));
82 ALGEBRAIC[9] = 1.70000/(exp( − 0.0820000*(STATES[0]+22.5000))+1.00000);
83 RATES[2] = ALGEBRAIC[2]*(1.00000 − STATES[2]) − ALGEBRAIC[9]*STATES[2];
84 ALGEBRAIC[3] = 0.0550000*exp( − 0.250000*(STATES[0]+78.0000))/(exp( − 0.200000*(
       STATES[0]+78.0000))+1.00000);
85 ALGEBRAIC[10] = 0.300000/(exp( − 0.100000*(STATES[0]+32.0000))+1.00000);
86 RATES[3] = ALGEBRAIC[3]*(1.00000 − STATES[3]) − ALGEBRAIC[10]*STATES[3];
87 ALGEBRAIC[4] = 0.0950000*exp(− (STATES[0] − 5.00000)/100.000)/(1.00000+exp(− (STATES
       [0] − 5.00000)/13.8900));
88 ALGEBRAIC[11] = 0.0700000*exp(− (STATES[0]+44.0000)/59.0000)/(1.00000+exp((STATES
       [0]+44.0000)/20.0000));
89 RATES[5] = ALGEBRAIC[4]*(1.00000 − STATES[5]) − ALGEBRAIC[11]*STATES[5];
90 ALGEBRAIC[5] = 0.0120000*exp(− (STATES[0]+28.0000)/125.000)/(1.00000+exp((STATES
       [0]+28.0000)/6.67000));
91 ALGEBRAIC[12] = 0.00650000*exp(− (STATES[0]+30.0000)/50.0000)/(1.00000+exp(− (STATES
       [0]+30.0000)/5.00000));
92 RATES[6] = ALGEBRAIC[5]*(1.00000 − STATES[6]) − ALGEBRAIC[12]*STATES[6];
93 ALGEBRAIC[6] = 0.000500000*exp((STATES[0]+50.0000)/12.1000)/(1.00000+exp((STATES
       [0]+50.0000)/17.5000));
94 ALGEBRAIC[13] = 0.00130000*exp(− (STATES[0]+20.0000)/16.6700)/(1.00000+exp(− (STATES
       [0]+20.0000)/25.0000));
95 RATES[7] = ALGEBRAIC[6]*(1.00000 − STATES[7]) − ALGEBRAIC[13]*STATES[7];
96 ALGEBRAIC[7] = − 82.3000 − 13.0287*log( STATES[4]*0.00100000);
97 ALGEBRAIC[14] = CONSTANTS[4]*STATES[5]*STATES[6]*(STATES[0] − ALGEBRAIC[7]);
98 RATES[4] = − 0.0100000*ALGEBRAIC[14]/1.00000+ 0.0700000*(0.000100000 − STATES[4]);
99 ALGEBRAIC[0] = ( CONSTANTS[1]*pow(STATES[1], 3.00000)*STATES[2]*STATES[3]+CONSTANTS
       [3])*(STATES[0] − CONSTANTS[2]);
```

```
100 ALGEBRAIC[15] = STATES[7]*0.00800000*(exp( 0.0400000*(STATES[0]+77.0000)) − 1.00000)/
      exp( 0.0400000*(STATES[0]+35.0000));
101 ALGEBRAIC[16] = 0.00350000*( 4.00000*(exp( 0.0400000*(STATES[0]+85.0000)) − 1.00000)
      /(exp( 0.0800000*(STATES[0]+53.0000))+exp( 0.0400000*(STATES[0]+53.0000)))+
      0.200000*(STATES[0]+23.0000)/(1.00000 − exp( − 0.0400000*(STATES[0]+23.0000))));
102 ALGEBRAIC[17] = (VOI>=CONSTANTS[5]&&VOI<=CONSTANTS[6]&&VOI − CONSTANTS[5] − floor((
      VOI − CONSTANTS[5])/CONSTANTS[8])*CONSTANTS[8]<=CONSTANTS[9] ? CONSTANTS[7] :
      0.00000);
103 RATES[0] = (ALGEBRAIC[17] − ALGEBRAIC[0]+ALGEBRAIC[14]+ALGEBRAIC[15]+ALGEBRAIC[16])/
      CONSTANTS[0];
104 }
105 void
106 computeVariables(double VOI, double* CONSTANTS, double* RATES, double* STATES, double*
      ALGEBRAIC)
107 {
108 ALGEBRAIC[1] = − 1.00000*(STATES[0]+47.0000)/(exp( − 0.100000*(STATES[0]+47.0000)) −
      1.00000);
109 ALGEBRAIC[8] = 40.0000*exp( − 0.0560000*(STATES[0]+72.0000));
110 ALGEBRAIC[2] = 0.126000*exp( − 0.250000*(STATES[0]+77.0000));
111 ALGEBRAIC[9] = 1.70000/(exp( − 0.0820000*(STATES[0]+22.5000))+1.00000);
112 ALGEBRAIC[3] = 0.0550000*exp( − 0.250000*(STATES[0]+78.0000))/(exp( − 0.200000*(
      STATES[0]+78.0000))+1.00000);
113 ALGEBRAIC[10] = 0.300000/(exp( − 0.100000*(STATES[0]+32.0000))+1.00000);
114 ALGEBRAIC[4] = 0.0950000*exp(− (STATES[0] − 5.00000)/100.000)/(1.00000+exp(− (STATES
      [0] − 5.00000)/13.8900));
115 ALGEBRAIC[11] = 0.0700000*exp(− (STATES[0]+44.0000)/59.0000)/(1.00000+exp((STATES
      [0]+44.0000)/20.0000));
116 ALGEBRAIC[5] = 0.0120000*exp(− (STATES[0]+28.0000)/125.000)/(1.00000+exp((STATES
      [0]+28.0000)/6.67000));
117 ALGEBRAIC[12] = 0.00650000*exp(− (STATES[0]+30.0000)/50.0000)/(1.00000+exp(− (STATES
      [0]+30.0000)/5.00000));
118 ALGEBRAIC[6] = 0.000500000*exp((STATES[0]+50.0000)/12.1000)/(1.00000+exp((STATES
      [0]+50.0000)/17.5000));
119 ALGEBRAIC[13] = 0.00130000*exp(− (STATES[0]+20.0000)/16.6700)/(1.00000+exp(− (STATES
      [0]+20.0000)/25.0000));
120 ALGEBRAIC[7] = − 82.3000 − 13.0287*log( STATES[4]*0.00100000);
121 ALGEBRAIC[14] = CONSTANTS[4]*STATES[5]*STATES[6]*(STATES[0] − ALGEBRAIC[7]);
122 ALGEBRAIC[0] = ( CONSTANTS[1]*pow(STATES[1], 3.00000)*STATES[2]*STATES[3]+CONSTANTS
      [3])*(STATES[0] − CONSTANTS[2]);
123 ALGEBRAIC[15] = STATES[7]*0.00800000*(exp( 0.0400000*(STATES[0]+77.0000)) − 1.00000)/
      exp( 0.0400000*(STATES[0]+35.0000));
124 ALGEBRAIC[16] = 0.00350000*( 4.00000*(exp( 0.0400000*(STATES[0]+85.0000)) − 1.00000)
      /(exp( 0.0800000*(STATES[0]+53.0000))+exp( 0.0400000*(STATES[0]+53.0000)))+
      0.200000*(STATES[0]+23.0000)/(1.00000 − exp( − 0.0400000*(STATES[0]+23.0000))));
125 ALGEBRAIC[17] = (VOI>=CONSTANTS[5]&&VOI<=CONSTANTS[6]&&VOI − CONSTANTS[5] − floor((
      VOI − CONSTANTS[5])/CONSTANTS[8])*CONSTANTS[8]<=CONSTANTS[9] ? CONSTANTS[7] :
      0.00000);
126 }
```

A.3 HILEMANN-NOBLE MODEL

A.3.1 *Mathematics*

"environment" component

This component has no equations.

"membrane" component

$$RTONF = \frac{R * T}{F}$$

$$i_Stim = \begin{cases} stim_amplitude & \text{if } (time \geq stim_start) \wedge (time \leq stim_end) \wedge time \\ & -stim_start - \left(\left\lfloor \frac{time - stim_start}{stim_period} \right\rfloor stim_period \right) \\ & \leqq stim_duration \\ 0 & \text{otherwise.} \end{cases}$$

$$\frac{d(V)}{d(time)} = \frac{-((i_Stim + i_K1 + i_b_Na + i_b_Ca + i_b_K + i_NaK + i_NaCa + i_Na + i_si))}{C_m}$$

"fast_sodium_current" component

$$E_mh = RTONF * \ln \left(\frac{Na_o + 0.12 * K_c}{Na_i + 0.12 * K_i} \right)$$

$$i_Na = g_Na * (m)^3 * h * (V - E_mh)$$

"fast_sodium_current_m_gate" component

$$E0_m = (V + 41)$$

$$alpha_m = \begin{cases} 2000 & \text{if } |E0_m| < delta_m \\ \frac{200 * E0_m}{\left(1 - e^{-(0.1) * E0_m}\right)} & \text{otherwise.} \end{cases}$$

$$beta_m = 8000 * e^{-(0.056) * (V + 66)}$$

$$\frac{d(m)}{d(time)} = (alpha_m * (1 - m) - beta_m * m)$$

"fast_sodium_current_h_gate" component

$$alpha_h = 20 * e^{-(0.125)*(V+75)}$$

$$beta_h = \frac{2000}{\left(1 + 320 * e^{-(0.1)*(V+75)}\right)}$$

$$\frac{\mathrm{d}(h)}{\mathrm{d}(time)} = (alpha_h * (1 - h) - beta_h * h)$$

"sodium_potassium_pump" component

$$i_NaK = \frac{\frac{i_NaK_max*K_c}{(K_mK+K_c)} * Na_i}{(K_mNa + Na_i)}$$

"sodium_background_current" component

$$E_Na = RTONF * \ln\left(\frac{Na_o}{Na_i}\right)$$

$$i_b_Na = g_b_Na * (V - E_Na)$$

"calcium_background_current" component

$$E_Ca = 0.5 * RTONF * \ln\left(\frac{Ca_o}{Ca_i}\right)$$

$$i_b_Ca = g_b_Ca * (V - E_Ca)$$

"Na_Ca_exchanger" component

$$i_NaCa = \frac{k_NaCa*\left(e^{\frac{gamma*(n_NaCa-2)*V}{RTONF}}*(Na_i)^{n_NaCa}*Ca_o-e^{\frac{(gamma-1)*(n_NaCa-2)*V}{RTONF}}*(Na_o)^{n_NaCa}*Ca_i\right)}{\left(1+d_NaCa*\left(Ca_i*(Na_o)^{n_NaCa}+Ca_o*(Na_i)^{n_NaCa}\right)\right)*\left(1+\frac{Ca_i}{0.0069}\right)}$$

"potassium_background_current" component

$$i_b_K = g_b_K * (V - E_K)$$

"time_independent_potassium_current" component

$$E_K = RTONF * \ln\left(\frac{K_c}{K_i}\right)$$

$$i_K1 = \frac{\frac{g_K1*K_c}{(K_c+K_m_K1)} * (V - E_K)}{\left(1 + e^{\frac{((V-E_K)-10)*2}{RTONF}}\right)}$$

"second_inward_calcium_current" component

$$i_si = (i_siCa + i_siK + i_siNa)$$

$$i_siCa = \frac{\frac{4*P_si*d*CaChon*(V-50)}{RTONF}}{\left(1 - e^{\frac{-(1)*(V-50)*2}{RTONF}}\right)} * \left(Ca_i * e^{\frac{100}{RTONF}} - Ca_o * e^{\frac{-(2)*(V-50)}{RTONF}}\right)$$

$$i_siK = \frac{\frac{0.002*P_si*d*CaChon*(V-50)}{RTONF}}{\left(1 - e^{\frac{-(1)*(V-50)}{RTONF}}\right)} * \left(K_i * e^{\frac{50}{RTONF}} - K_c * e^{\frac{-(1)*(V-50)}{RTONF}}\right)$$

$$i_siNa = \frac{\frac{0.01*P_si*d*CaChon*(V-50)}{RTONF}}{\left(1 - e^{\frac{-(1)*(V-50)}{RTONF}}\right)} * \left(Na_i * e^{\frac{50}{RTONF}} - Na_o * e^{\frac{-(1)*(V-50)}{RTONF}}\right)$$

"second_inward_calcium_current_d_gate" component

$$E0_d = ((V + 24) - 5)$$

$$alpha_d = \begin{cases} 120 & \text{if } |E0_d| < delta_d \\ \frac{30*E0_d}{\left(1-e^{\frac{-(1)*E0_d}{4}}\right)} & \text{otherwise.} \end{cases}$$

$$beta_d = \begin{cases} 120 & \text{if } |E0_d| < delta_d \\ \frac{12*E0_d}{\left(e^{\frac{E0_d}{10}}-1\right)} & \text{otherwise.} \end{cases}$$

$$\frac{\mathrm{d}(d)}{\mathrm{d}(time)} = (alpha_d * (1 - d) - beta_d * d)$$

"second_inward_calcium_current_f_Ca_gate" component

$$E0_f = (V + 34)$$

$$alpha_f_Ca = \begin{cases} 25 & \text{if } |E0_f| < delta_f \\ \frac{6.25*E0_f}{\left(e^{\frac{E0_f}{4}}-1\right)} & \text{otherwise.} \end{cases}$$

$$beta_f_Ca = \frac{12}{\left(1 + e^{\frac{-(E0_f)}{4}}\right)}$$

$$\frac{d(f_Ca)}{d(time)} = ((120 * (1 - f_Ca) * CaChoff + (1 - f_Ca) * (1 - CaChoff)) * beta_f_Ca - alpha_f_Ca * f_Ca)$$

$$CaChoff = \frac{Ca_i}{(0.001 + Ca_i)}$$

$$CaChon = (1 - f_Ca) * (1 - CaChoff)$$

"sarcoplasmic_reticulum_calcium_pump" component

$$K_1 = \frac{K_cyca * K_xcs}{K_srca}$$

$$K_2 = (Ca_i + Ca_up * K_1 + K_cyca * K_xcs + K_cyca)$$

$$i_up = \left(\frac{Ca_i}{K_2} * alpha_up - \frac{Ca_up * K_1}{K_2} * beta_up\right)$$

"calcium_release" component

$$PrecFrac = ((1 - ActFrac) - ProdFrac)$$

$$VoltDep = e^{0.08*(V-40)}$$

$$RegBindSite = \left(\frac{Ca_i}{(Ca_i + 0.0005)}\right)^2$$

$$ActRate = (600 * VoltDep + 500 * RegBindSite)$$

$$InactRate = (60 + 500 * RegBindSite)$$

$$\frac{d(ActFrac)}{d(time)} = (PrecFrac * ActRate - ActFrac * InactRate)$$

$$\frac{d(ProdFrac)}{d(time)} = (ActFrac * InactRate - 0.6 * ProdFrac)$$

$$i_rel = \left(\left(\frac{ActFrac}{(ActFrac + 0.25)}\right)^2 * K_m_rel + K_leak_rate\right) * Ca_rel$$

"calcium_translocation" component

$$i_trans = (Ca_up - Ca_rel) * alpha_tr$$

"extracellular_sodium_concentration" component

This component has no equations.

"intracellular_sodium_concentration" component

$$\frac{\mathrm{d}(Na_i)}{\mathrm{d}(time)} = \frac{-(1)}{1 * V_i * F} * (i_Na + i_b_Na + i_NaK * 3 + i_NaCa * 3 + i_siNa)$$

"extracellular_calcium_concentration" component

$$\frac{\mathrm{d}(Ca_o)}{\mathrm{d}(time)} = \left((Cab - Ca_o) * K_diff - \frac{1 * (i_siCa + i_NaCa + i_b_Ca)}{2 * 1 * Ve * F}\right)$$

"extracellular_potassium_concentration" component

This component has no equations.

"intracellular_potassium_concentration" component

$$\frac{\mathrm{d}(K_i)}{\mathrm{d}(time)} = \frac{-(1)}{1 * V_i * F} * ((i_K1 + i_siK + i_b_K) - 2 * i_NaK)$$

"intracellular_calcium_concentration" component

$$V_Cell = 3.141592654 * (radius)^2 * length$$

$$V_i_ratio = (((1 - V_e_ratio) - V_up_ratio) - V_rel_ratio)$$

$$V_i = V_Cell * V_i_ratio$$

$$Ve = V_Cell * V_e_ratio$$

$$\begin{aligned} \frac{\mathrm{d}(Ca_i)}{\mathrm{d}(time)} = & ((((\frac{-1}{2 * 1 * V_i * F} * ((i_siCa + i_b_Ca) - 2 * i_NaCa) \\ & + \frac{i_rel * V_rel_ratio}{V_i_ratio}) - dCaCalmoddt) - dCaTropdt) - i_up) \end{aligned}$$

$$\frac{d(Ca_up)}{d(time)} = \left(\frac{V_i_ratio}{V_up_ratio} * i_up - i_trans\right)$$

$$\frac{d(Ca_rel)}{d(time)} = \left(\frac{V_up_ratio}{V_rel_ratio} * i_trans - i_rel\right)$$

$$\frac{d(Ca_Calmod)}{d(time)} = (alpha_Trop * Ca_i * (Calmod - Ca_Calmod) - beta_Calmod * Ca_Calmod)$$

$$\frac{d(Ca_Trop)}{d(time)} = (alpha_Trop * Ca_i * (Trop - Ca_Trop) - beta_Trop * Ca_Trop)$$

$$dCaCalmoddt = (alpha_Calmod * Ca_i * (Calmod - Ca_Calmod) - beta_Calmod * Ca_Calmod)$$

$$dCaTropdt = (alpha_Trop * Ca_i * (Trop - Ca_Trop) - beta_Trop * Ca_Trop)$$

A.3.2 *C-code Representation*

```
/*
   There are a total of 40 entries in the algebraic variable array.
   There are a total of 15 entries in each of the rate and state variable arrays.
   There are a total of 55 entries in the constant variable array.
 */
/*
 * VOI is time in component environment (second).
 * STATES[0] is V in component membrane (millivolt).
 * CONSTANTS[0] is R in component membrane (joule_per_kilomole_kelvin).
 * CONSTANTS[1] is T in component membrane (kelvin).
 * CONSTANTS[2] is F in component membrane (coulomb_per_mole).
 * CONSTANTS[49] is RTONF in component membrane (millivolt).
 * CONSTANTS[3] is C_m in component membrane (microF).
 * ALGEBRAIC[24] is i_K1 in component time_independent_potassium_current (nanoA).
 * ALGEBRAIC[18] is i_b_Na in component sodium_background_current (nanoA).
 * ALGEBRAIC[20] is i_b_Ca in component calcium_background_current (nanoA).
 * ALGEBRAIC[23] is i_b_K in component potassium_background_current (nanoA).
 * ALGEBRAIC[14] is i_NaK in component sodium_potassium_pump (nanoA).
 * ALGEBRAIC[21] is i_NaCa in component Na_Ca_exchanger (nanoA).
 * ALGEBRAIC[12] is i_Na in component fast_sodium_current (nanoA).
 * ALGEBRAIC[35] is i_si in component second_inward_calcium_current (nanoA).
```

```
22  * ALGEBRAIC[3] is i_Stim in component membrane (nanoA).
23  * CONSTANTS[4] is stim_start in component membrane (second).
24  * CONSTANTS[5] is stim_end in component membrane (second).
25  * CONSTANTS[6] is stim_period in component membrane (second).
26  * CONSTANTS[7] is stim_duration in component membrane (second).
27  * CONSTANTS[8] is stim_amplitude in component membrane (nanoA).
28  * CONSTANTS[9] is g_Na in component fast_sodium_current (microS).
29  * ALGEBRAIC[8] is E_mh in component fast_sodium_current (millivolt).
30  * CONSTANTS[10] is Na_o in component extracellular_sodium_concentration (millimolar).
31  * STATES[1] is Na_i in component intracellular_sodium_concentration (millimolar).
32  * CONSTANTS[11] is K_c in component extracellular_potassium_concentration (millimolar
      ).
33  * STATES[2] is K_i in component intracellular_potassium_concentration (millimolar).
34  * STATES[3] is m in component fast_sodium_current_m_gate (dimensionless).
35  * STATES[4] is h in component fast_sodium_current_h_gate (dimensionless).
36  * ALGEBRAIC[5] is alpha_m in component fast_sodium_current_m_gate (per_second).
37  * ALGEBRAIC[10] is beta_m in component fast_sodium_current_m_gate (per_second).
38  * CONSTANTS[12] is delta_m in component fast_sodium_current_m_gate (millivolt).
39  * ALGEBRAIC[0] is E0_m in component fast_sodium_current_m_gate (millivolt).
40  * ALGEBRAIC[1] is alpha_h in component fast_sodium_current_h_gate (per_second).
41  * ALGEBRAIC[6] is beta_h in component fast_sodium_current_h_gate (per_second).
42  * CONSTANTS[13] is i_NaK_max in component sodium_potassium_pump (nanoA).
43  * CONSTANTS[14] is K_mK in component sodium_potassium_pump (millimolar).
44  * CONSTANTS[15] is K_mNa in component sodium_potassium_pump (millimolar).
45  * ALGEBRAIC[16] is E_Na in component sodium_background_current (millivolt).
46  * CONSTANTS[16] is g_b_Na in component sodium_background_current (microS).
47  * ALGEBRAIC[19] is E_Ca in component calcium_background_current (millivolt).
48  * CONSTANTS[17] is g_b_Ca in component calcium_background_current (microS).
49  * STATES[5] is Ca_o in component extracellular_calcium_concentration (millimolar).
50  * STATES[6] is Ca_i in component intracellular_calcium_concentration (millimolar).
51  * CONSTANTS[18] is k_NaCa in component Na_Ca_exchanger (nanoA).
52  * CONSTANTS[19] is n_NaCa in component Na_Ca_exchanger (dimensionless).
53  * CONSTANTS[20] is d_NaCa in component Na_Ca_exchanger (dimensionless).
54  * CONSTANTS[21] is gamma in component Na_Ca_exchanger (dimensionless).
55  * ALGEBRAIC[22] is E_K in component time_independent_potassium_current (millivolt).
56  * CONSTANTS[22] is g_b_K in component potassium_background_current (microS).
57  * CONSTANTS[23] is g_K1 in component time_independent_potassium_current (microS).
58  * CONSTANTS[24] is K_m_K1 in component time_independent_potassium_current (millimolar
      ).
59  * ALGEBRAIC[30] is i_siCa in component second_inward_calcium_current (nanoA).
60  * ALGEBRAIC[31] is i_siK in component second_inward_calcium_current (nanoA).
61  * ALGEBRAIC[33] is i_siNa in component second_inward_calcium_current (nanoA).
62  * CONSTANTS[25] is P_si in component second_inward_calcium_current (
      nanoA_per_millimolar).
63  * STATES[7] is d in component second_inward_calcium_current_d_gate (dimensionless).
64  * STATES[8] is f_Ca in component second_inward_calcium_current_f_Ca_gate (
      dimensionless).
65  * ALGEBRAIC[29] is CaChon in component second_inward_calcium_current_f_Ca_gate (
      dimensionless).
```

```
66  * ALGEBRAIC[7] is alpha_d in component second_inward_calcium_current_d_gate (
        per_second).
67  * ALGEBRAIC[11] is beta_d in component second_inward_calcium_current_d_gate (
        per_second).
68  * CONSTANTS[26] is delta_d in component second_inward_calcium_current_d_gate (
        millivolt).
69  * ALGEBRAIC[2] is E0_d in component second_inward_calcium_current_d_gate (millivolt).
70  * ALGEBRAIC[26] is alpha_f_Ca in component second_inward_calcium_current_f_Ca_gate (
        per_second).
71  * ALGEBRAIC[27] is beta_f_Ca in component second_inward_calcium_current_f_Ca_gate (
        per_second).
72  * ALGEBRAIC[28] is CaChoff in component second_inward_calcium_current_f_Ca_gate (
        dimensionless).
73  * CONSTANTS[27] is delta_f in component second_inward_calcium_current_f_Ca_gate (
        millivolt).
74  * ALGEBRAIC[25] is E0_f in component second_inward_calcium_current_f_Ca_gate (
        millivolt).
75  * ALGEBRAIC[34] is i_up in component sarcoplasmic_reticulum_calcium_pump (
        millimolar_per_second).
76  * CONSTANTS[51] is K_1 in component sarcoplasmic_reticulum_calcium_pump (
        dimensionless).
77  * ALGEBRAIC[32] is K_2 in component sarcoplasmic_reticulum_calcium_pump (millimolar).
78  * CONSTANTS[28] is K_cyca in component sarcoplasmic_reticulum_calcium_pump (
        millimolar).
79  * CONSTANTS[29] is K_xcs in component sarcoplasmic_reticulum_calcium_pump (
        dimensionless).
80  * CONSTANTS[30] is K_srca in component sarcoplasmic_reticulum_calcium_pump (
        millimolar).
81  * CONSTANTS[31] is alpha_up in component sarcoplasmic_reticulum_calcium_pump (
        millimolar_per_second).
82  * CONSTANTS[32] is beta_up in component sarcoplasmic_reticulum_calcium_pump (
        millimolar_per_second).
83  * STATES[9] is Ca_up in component intracellular_calcium_concentration (millimolar).
84  * ALGEBRAIC[36] is i_rel in component calcium_release (millimolar_per_second).
85  * ALGEBRAIC[9] is VoltDep in component calcium_release (dimensionless).
86  * ALGEBRAIC[13] is RegBindSite in component calcium_release (dimensionless).
87  * ALGEBRAIC[15] is ActRate in component calcium_release (per_second).
88  * ALGEBRAIC[17] is InactRate in component calcium_release (per_second).
89  * CONSTANTS[33] is K_leak_rate in component calcium_release (per_second).
90  * CONSTANTS[34] is K_m_rel in component calcium_release (per_second).
91  * ALGEBRAIC[4] is PrecFrac in component calcium_release (dimensionless).
92  * STATES[10] is ActFrac in component calcium_release (dimensionless).
93  * STATES[11] is ProdFrac in component calcium_release (dimensionless).
94  * STATES[12] is Ca_rel in component intracellular_calcium_concentration (millimolar).
95  * ALGEBRAIC[37] is i_trans in component calcium_translocation (millimolar_per_second)
        .
96  * CONSTANTS[35] is alpha_tr in component calcium_translocation (per_second).
97  * CONSTANTS[54] is V_i in component intracellular_calcium_concentration (micrometre3)
        .
```

```
98   * CONSTANTS[36] is Cab in component extracellular_calcium_concentration (millimolar).
99   * CONSTANTS[37] is K_diff in component extracellular_calcium_concentration (
       per_second).
100  * CONSTANTS[53] is Ve in component intracellular_calcium_concentration (micrometre3).
101  * STATES[13] is Ca_Calmod in component intracellular_calcium_concentration (
       millimolar).
102  * STATES[14] is Ca_Trop in component intracellular_calcium_concentration (millimolar)
       .
103  * CONSTANTS[38] is Calmod in component intracellular_calcium_concentration (
       millimolar).
104  * CONSTANTS[39] is Trop in component intracellular_calcium_concentration (millimolar)
       .
105  * CONSTANTS[40] is alpha_Calmod in component intracellular_calcium_concentration (
       per_millimolar_second).
106  * CONSTANTS[41] is beta_Calmod in component intracellular_calcium_concentration (
       per_second).
107  * CONSTANTS[42] is alpha_Trop in component intracellular_calcium_concentration (
       per_millimolar_second).
108  * CONSTANTS[43] is beta_Trop in component intracellular_calcium_concentration (
       per_second).
109  * CONSTANTS[44] is radius in component intracellular_calcium_concentration (
       micrometre).
110  * CONSTANTS[45] is length in component intracellular_calcium_concentration (
       micrometre).
111  * CONSTANTS[50] is V_Cell in component intracellular_calcium_concentration (
       micrometre3).
112  * CONSTANTS[52] is V_i_ratio in component intracellular_calcium_concentration (
       dimensionless).
113  * CONSTANTS[46] is V_rel_ratio in component intracellular_calcium_concentration (
       dimensionless).
114  * CONSTANTS[47] is V_e_ratio in component intracellular_calcium_concentration (
       dimensionless).
115  * CONSTANTS[48] is V_up_ratio in component intracellular_calcium_concentration (
       dimensionless).
116  * ALGEBRAIC[38] is dCaCalmoddt in component intracellular_calcium_concentration (
       millimolar_per_second).
117  * ALGEBRAIC[39] is dCaTropdt in component intracellular_calcium_concentration (
       millimolar_per_second).
118  * RATES[0] is d/dt V in component membrane (millivolt).
119  * RATES[3] is d/dt m in component fast_sodium_current_m_gate (dimensionless).
120  * RATES[4] is d/dt h in component fast_sodium_current_h_gate (dimensionless).
121  * RATES[7] is d/dt d in component second_inward_calcium_current_d_gate (dimensionless
       ).
122  * RATES[8] is d/dt f_Ca in component second_inward_calcium_current_f_Ca_gate (
       dimensionless).
123  * RATES[10] is d/dt ActFrac in component calcium_release (dimensionless).
124  * RATES[11] is d/dt ProdFrac in component calcium_release (dimensionless).
125  * RATES[1] is d/dt Na_i in component intracellular_sodium_concentration (millimolar).
```

```
126  * RATES[5] is d/dt Ca_o in component extracellular_calcium_concentration (millimolar)
        .
127  * RATES[2] is d/dt K_i in component intracellular_potassium_concentration (millimolar
        ).
128  * RATES[6] is d/dt Ca_i in component intracellular_calcium_concentration (millimolar)
        .
129  * RATES[9] is d/dt Ca_up in component intracellular_calcium_concentration (millimolar
        ).
130  * RATES[12] is d/dt Ca_rel in component intracellular_calcium_concentration (
        millimolar).
131  * RATES[13] is d/dt Ca_Calmod in component intracellular_calcium_concentration (
        millimolar).
132  * RATES[14] is d/dt Ca_Trop in component intracellular_calcium_concentration (
        millimolar).
133  */
134 void
135 initConsts(double* CONSTANTS, double* RATES, double *STATES)
136 {
137 STATES[0] = -88;
138 CONSTANTS[0] = 8314.472;
139 CONSTANTS[1] = 310;
140 CONSTANTS[2] = 96485.3415;
141 CONSTANTS[3] = 0.006;
142 CONSTANTS[4] = 0.1;
143 CONSTANTS[5] = 10000;
144 CONSTANTS[6] = 1;
145 CONSTANTS[7] = 0.002;
146 CONSTANTS[8] = -200;
147 CONSTANTS[9] = 50;
148 CONSTANTS[10] = 140;
149 STATES[1] = 6.5;
150 CONSTANTS[11] = 4;
151 STATES[2] = 140;
152 STATES[3] = 0.076;
153 STATES[4] = 0.015;
154 CONSTANTS[12] = 1e-5;
155 CONSTANTS[13] = 14;
156 CONSTANTS[14] = 1;
157 CONSTANTS[15] = 40;
158 CONSTANTS[16] = 0.012;
159 CONSTANTS[17] = 0.005;
160 STATES[5] = 2;
161 STATES[6] = 1e-5;
162 CONSTANTS[18] = 0.01;
163 CONSTANTS[19] = 3;
164 CONSTANTS[20] = 0.0001;
165 CONSTANTS[21] = 0.5;
166 CONSTANTS[22] = 0.17;
167 CONSTANTS[23] = 1.7;
```

```
168 CONSTANTS[24] = 10;
169 CONSTANTS[25] = 5;
170 STATES[7] = 0.0011;
171 STATES[8] = 0.785;
172 CONSTANTS[26] = 0.0001;
173 CONSTANTS[27] = 0.0001;
174 CONSTANTS[28] = 0.0003;
175 CONSTANTS[29] = 0.4;
176 CONSTANTS[30] = 0.5;
177 CONSTANTS[31] = 3;
178 CONSTANTS[32] = 0.23;
179 STATES[9] = 0.3;
180 CONSTANTS[33] = 0;
181 CONSTANTS[34] = 250;
182 STATES[10] = 0;
183 STATES[11] = 0;
184 STATES[12] = 0.3;
185 CONSTANTS[35] = 50;
186 CONSTANTS[36] = 2;
187 CONSTANTS[37] = 0.0005;
188 STATES[13] = 0.0005;
189 STATES[14] = 0.0015;
190 CONSTANTS[38] = 0.02;
191 CONSTANTS[39] = 0.15;
192 CONSTANTS[40] = 100000;
193 CONSTANTS[41] = 50;
194 CONSTANTS[42] = 100000;
195 CONSTANTS[43] = 200;
196 CONSTANTS[44] = 0.08;
197 CONSTANTS[45] = 0.08;
198 CONSTANTS[46] = 0.1;
199 CONSTANTS[47] = 0.4;
200 CONSTANTS[48] = 0.01;
201 CONSTANTS[49] = CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2];
202 CONSTANTS[50] = 3.14159*pow(CONSTANTS[44], 2.00000)*CONSTANTS[45];
203 CONSTANTS[51] = CONSTANTS[28]*CONSTANTS[29]/CONSTANTS[30];
204 CONSTANTS[52] = 1.00000 - CONSTANTS[47] - CONSTANTS[48] - CONSTANTS[46];
205 CONSTANTS[53] = CONSTANTS[50]*CONSTANTS[47];
206 CONSTANTS[54] = CONSTANTS[50]*CONSTANTS[52];
207 }
208 void
209 computeRates(double VOI, double* CONSTANTS, double* RATES, double* STATES, double*
        ALGEBRAIC)
210 {
211 RATES[13] = CONSTANTS[40]*STATES[6]*(CONSTANTS[38] - STATES[13]) - CONSTANTS[41]*
        STATES[13];
212 RATES[14] = CONSTANTS[42]*STATES[6]*(CONSTANTS[39] - STATES[14]) - CONSTANTS[43]*
        STATES[14];
213 ALGEBRAIC[1] = 20.0000*exp( - 0.125000*(STATES[0]+75.0000));
```

```
214 ALGEBRAIC[6] = 2000.00/(1.00000+ 320.000*exp( - 0.100000*(STATES[0]+75.0000)));
215 RATES[4] = ALGEBRAIC[1]*(1.00000 - STATES[4]) - ALGEBRAIC[6]*STATES[4];
216 ALGEBRAIC[0] = STATES[0]+41.0000;
217 ALGEBRAIC[5] = (fabs(ALGEBRAIC[0])<CONSTANTS[12] ? 2000.00 : 200.000*ALGEBRAIC
      [0]/(1.00000 - exp( - 0.100000*ALGEBRAIC[0])));
218 ALGEBRAIC[10] = 8000.00*exp( - 0.0560000*(STATES[0]+66.0000));
219 RATES[3] = ALGEBRAIC[5]*(1.00000 - STATES[3]) - ALGEBRAIC[10]*STATES[3];
220 ALGEBRAIC[2] = STATES[0]+24.0000 - 5.00000;
221 ALGEBRAIC[7] = (fabs(ALGEBRAIC[2])<CONSTANTS[26] ? 120.000 : 30.0000*ALGEBRAIC
      [2]/(1.00000 - exp( - 1.00000*ALGEBRAIC[2]/4.00000)));
222 ALGEBRAIC[11] = (fabs(ALGEBRAIC[2])<CONSTANTS[26] ? 120.000 : 12.0000*ALGEBRAIC[2]/(
      exp(ALGEBRAIC[2]/10.0000) - 1.00000));
223 RATES[7] = ALGEBRAIC[7]*(1.00000 - STATES[7]) - ALGEBRAIC[11]*STATES[7];
224 ALGEBRAIC[9] = exp( 0.0800000*(STATES[0] - 40.0000));
225 ALGEBRAIC[13] = pow(STATES[6]/(STATES[6]+0.000500000), 2.00000);
226 ALGEBRAIC[15] = 600.000*ALGEBRAIC[9]+ 500.000*ALGEBRAIC[13];
227 ALGEBRAIC[17] = 60.0000+ 500.000*ALGEBRAIC[13];
228 ALGEBRAIC[4] = 1.00000 - STATES[10] - STATES[11];
229 RATES[10] = ALGEBRAIC[4]*ALGEBRAIC[15] - STATES[10]*ALGEBRAIC[17];
230 RATES[11] = STATES[10]*ALGEBRAIC[17] - 0.600000*STATES[11];
231 ALGEBRAIC[25] = STATES[0]+34.0000;
232 ALGEBRAIC[26] = (fabs(ALGEBRAIC[25])<CONSTANTS[27] ? 25.0000 : 6.25000*ALGEBRAIC
      [25]/(exp(ALGEBRAIC[25]/4.00000) - 1.00000));
233 ALGEBRAIC[27] = 12.0000/(1.00000+exp(- ALGEBRAIC[25]/4.00000));
234 ALGEBRAIC[28] = STATES[6]/(0.00100000+STATES[6]);
235 RATES[8] = ( 120.000*(1.00000 - STATES[8])*ALGEBRAIC[28]+ (1.00000 - STATES[8])
      *(1.00000 - ALGEBRAIC[28]))*ALGEBRAIC[27] - ALGEBRAIC[26]*STATES[8];
236 ALGEBRAIC[19] = 0.500000*CONSTANTS[49]*log(STATES[5]/STATES[6]);
237 ALGEBRAIC[20] = CONSTANTS[17]*(STATES[0] - ALGEBRAIC[19]);
238 ALGEBRAIC[21] = CONSTANTS[18]*( exp( CONSTANTS[21]*(CONSTANTS[19] - 2.00000)*STATES
      [0]/CONSTANTS[49])*pow(STATES[1], CONSTANTS[19])*STATES[5] - exp( (CONSTANTS[21]
      - 1.00000)*(CONSTANTS[19] - 2.00000)*STATES[0]/CONSTANTS[49])*pow(CONSTANTS[10],
      CONSTANTS[19])*STATES[6])/( (1.00000+ CONSTANTS[20]*( STATES[6]*pow(CONSTANTS[10],
       CONSTANTS[19])+ STATES[5]*pow(STATES[1], CONSTANTS[19])))*(1.00000+STATES
      [6]/0.00690000));
239 ALGEBRAIC[29] = (1.00000 - STATES[8])*(1.00000 - ALGEBRAIC[28]);
240 ALGEBRAIC[30] = 4.00000*CONSTANTS[25]*STATES[7]*ALGEBRAIC[29]*(STATES[0] - 50.0000)/
      CONSTANTS[49]/(1.00000 - exp( - 1.00000*(STATES[0] - 50.0000)*2.00000/CONSTANTS
      [49]))*( STATES[6]*exp(100.000/CONSTANTS[49]) - STATES[5]*exp( - 2.00000*(STATES
      [0] - 50.0000)/CONSTANTS[49]));
241 RATES[5] = (CONSTANTS[36] - STATES[5])*CONSTANTS[37] - 1.00000*(ALGEBRAIC[30]+
      ALGEBRAIC[21]+ALGEBRAIC[20])/( 2.00000*1.00000*CONSTANTS[53]*CONSTANTS[2]);
242 ALGEBRAIC[22] = CONSTANTS[49]*log(CONSTANTS[11]/STATES[2]);
243 ALGEBRAIC[24] = CONSTANTS[23]*CONSTANTS[11]/(CONSTANTS[11]+CONSTANTS[24])*(STATES[0]
       - ALGEBRAIC[22])/(1.00000+exp( (STATES[0] - ALGEBRAIC[22] - 10.0000)*2.00000/
      CONSTANTS[49]));
244 ALGEBRAIC[23] = CONSTANTS[22]*(STATES[0] - ALGEBRAIC[22]);
245 ALGEBRAIC[14] = CONSTANTS[13]*CONSTANTS[11]/(CONSTANTS[14]+CONSTANTS[11])*STATES
      [1]/(CONSTANTS[15]+STATES[1]);
```

```
246 ALGEBRAIC[31] = 0.00200000*CONSTANTS[25]*STATES[7]*ALGEBRAIC[29]*(STATES[0] −
        50.0000)/CONSTANTS[49]/(1.00000 − exp( − 1.00000*(STATES[0] − 50.0000)/CONSTANTS
        [49]))*( STATES[2]*exp(50.0000/CONSTANTS[49]) − CONSTANTS[11]*exp( − 1.00000*(
        STATES[0] − 50.0000)/CONSTANTS[49]));
247 RATES[2] = − 1.00000/( 1.00000*CONSTANTS[54]*CONSTANTS[2])*(ALGEBRAIC[24]+ALGEBRAIC
        [31]+ALGEBRAIC[23] − 2.00000*ALGEBRAIC[14]);
248 ALGEBRAIC[16] = CONSTANTS[49]*log(CONSTANTS[10]/STATES[1]);
249 ALGEBRAIC[18] = CONSTANTS[16]*(STATES[0] − ALGEBRAIC[16]);
250 ALGEBRAIC[8] = CONSTANTS[49]*log((CONSTANTS[10]+ 0.120000*CONSTANTS[11])/(STATES[1]+
        0.120000*STATES[2]));
251 ALGEBRAIC[12] = CONSTANTS[9]*pow(STATES[3], 3.00000)*STATES[4]*(STATES[0] − ALGEBRAIC
        [8]);
252 ALGEBRAIC[33] = 0.0100000*CONSTANTS[25]*STATES[7]*ALGEBRAIC[29]*(STATES[0] −
        50.0000)/CONSTANTS[49]/(1.00000 − exp( − 1.00000*(STATES[0] − 50.0000)/CONSTANTS
        [49]))*( STATES[1]*exp(50.0000/CONSTANTS[49]) − CONSTANTS[10]*exp( − 1.00000*(
        STATES[0] − 50.0000)/CONSTANTS[49]));
253 RATES[1] = − 1.00000/( 1.00000*CONSTANTS[54]*CONSTANTS[2])*(ALGEBRAIC[12]+ALGEBRAIC
        [18]+ ALGEBRAIC[14]*3.00000+ ALGEBRAIC[21]*3.00000+ALGEBRAIC[33]);
254 ALGEBRAIC[35] = ALGEBRAIC[30]+ALGEBRAIC[31]+ALGEBRAIC[33];
255 ALGEBRAIC[3] = (VOI>=CONSTANTS[4]&&VOI<=CONSTANTS[5]&&VOI − CONSTANTS[4] − floor((VOI
         − CONSTANTS[4])/CONSTANTS[6])*CONSTANTS[6]<=CONSTANTS[7] ? CONSTANTS[8] :
        0.00000);
256 RATES[0] = − (ALGEBRAIC[3]+ALGEBRAIC[24]+ALGEBRAIC[18]+ALGEBRAIC[20]+ALGEBRAIC[23]+
        ALGEBRAIC[14]+ALGEBRAIC[21]+ALGEBRAIC[12]+ALGEBRAIC[35])/CONSTANTS[3];
257 ALGEBRAIC[32] = STATES[6]+ STATES[9]*CONSTANTS[51]+ CONSTANTS[28]*CONSTANTS[29]+
        CONSTANTS[28];
258 ALGEBRAIC[34] = STATES[6]/ALGEBRAIC[32]*CONSTANTS[31] − STATES[9]*CONSTANTS[51]/
        ALGEBRAIC[32]*CONSTANTS[32];
259 ALGEBRAIC[37] = (STATES[9] − STATES[12])*CONSTANTS[35];
260 RATES[9] = CONSTANTS[52]/CONSTANTS[48]*ALGEBRAIC[34] − ALGEBRAIC[37];
261 ALGEBRAIC[36] = ( pow(STATES[10]/(STATES[10]+0.250000), 2.00000)*CONSTANTS[34]+
        CONSTANTS[33])*STATES[12];
262 RATES[12] = CONSTANTS[48]/CONSTANTS[46]*ALGEBRAIC[37] − ALGEBRAIC[36];
263 ALGEBRAIC[38] = CONSTANTS[40]*STATES[6]*(CONSTANTS[38] − STATES[13]) − CONSTANTS
        [41]*STATES[13];
264 ALGEBRAIC[39] = CONSTANTS[42]*STATES[6]*(CONSTANTS[39] − STATES[14]) − CONSTANTS
        [43]*STATES[14];
265 RATES[6] = − 1.00000/( 2.00000*1.00000*CONSTANTS[54]*CONSTANTS[2])*(ALGEBRAIC[30]+
        ALGEBRAIC[20] − 2.00000*ALGEBRAIC[21])+ ALGEBRAIC[36]*CONSTANTS[46]/CONSTANTS[52]
         − ALGEBRAIC[38] − ALGEBRAIC[39] − ALGEBRAIC[34];
266 }
267 void
268 computeVariables(double VOI, double* CONSTANTS, double* RATES, double* STATES, double*
         ALGEBRAIC)
269 {
270 ALGEBRAIC[1] = 20.0000*exp( − 0.125000*(STATES[0]+75.0000));
271 ALGEBRAIC[6] = 2000.00/(1.00000+ 320.000*exp( − 0.100000*(STATES[0]+75.0000)));
272 ALGEBRAIC[0] = STATES[0]+41.0000;
```

```
273 ALGEBRAIC[5] = (fabs(ALGEBRAIC[0])<CONSTANTS[12] ? 2000.00 : 200.000*ALGEBRAIC
        [0]/(1.00000 - exp( - 0.100000*ALGEBRAIC[0])));
274 ALGEBRAIC[10] = 8000.00*exp( - 0.0560000*(STATES[0]+66.0000));
275 ALGEBRAIC[2] = STATES[0]+24.0000 - 5.00000;
276 ALGEBRAIC[7] = (fabs(ALGEBRAIC[2])<CONSTANTS[26] ? 120.000 : 30.0000*ALGEBRAIC
        [2]/(1.00000 - exp( - 1.00000*ALGEBRAIC[2]/4.00000)));
277 ALGEBRAIC[11] = (fabs(ALGEBRAIC[2])<CONSTANTS[26] ? 120.000 : 12.0000*ALGEBRAIC[2]/(
        exp(ALGEBRAIC[2]/10.0000) - 1.00000));
278 ALGEBRAIC[9] = exp( 0.0800000*(STATES[0] - 40.0000));
279 ALGEBRAIC[13] = pow(STATES[6]/(STATES[6]+0.000500000), 2.00000);
280 ALGEBRAIC[15] = 600.000*ALGEBRAIC[9]+ 500.000*ALGEBRAIC[13];
281 ALGEBRAIC[17] = 60.0000+ 500.000*ALGEBRAIC[13];
282 ALGEBRAIC[4] = 1.00000 - STATES[10] - STATES[11];
283 ALGEBRAIC[25] = STATES[0]+34.0000;
284 ALGEBRAIC[26] = (fabs(ALGEBRAIC[25])<CONSTANTS[27] ? 25.0000 : 6.25000*ALGEBRAIC
        [25]/(exp(ALGEBRAIC[25]/4.00000) - 1.00000));
285 ALGEBRAIC[27] = 12.0000/(1.00000+exp(- ALGEBRAIC[25]/4.00000));
286 ALGEBRAIC[28] = STATES[6]/(0.00100000+STATES[6]);
287 ALGEBRAIC[19] = 0.500000*CONSTANTS[49]*log(STATES[5]/STATES[6]);
288 ALGEBRAIC[20] = CONSTANTS[17]*(STATES[0] - ALGEBRAIC[19]);
289 ALGEBRAIC[21] = CONSTANTS[18]*( exp( CONSTANTS[21]*(CONSTANTS[19] - 2.00000)*STATES
        [0]/CONSTANTS[49])*pow(STATES[1], CONSTANTS[19])*STATES[5] - exp( (CONSTANTS[21]
        - 1.00000)*(CONSTANTS[19] - 2.00000)*STATES[0]/CONSTANTS[49])*pow(CONSTANTS[10],
        CONSTANTS[19])*STATES[6])/( (1.00000+ CONSTANTS[20]*( STATES[6]*pow(CONSTANTS[10],
         CONSTANTS[19])+ STATES[5]*pow(STATES[1], CONSTANTS[19])))*(1.00000+STATES
        [6]/0.00690000));
290 ALGEBRAIC[29] = (1.00000 - STATES[8])*(1.00000 - ALGEBRAIC[28]);
291 ALGEBRAIC[30] = 4.00000*CONSTANTS[25]*STATES[7]*ALGEBRAIC[29]*(STATES[0] - 50.0000)/
        CONSTANTS[49]/(1.00000 - exp( - 1.00000*(STATES[0] - 50.0000)*2.00000/CONSTANTS
        [49]))*( STATES[6]*exp(100.000/CONSTANTS[49]) - STATES[5]*exp( - 2.00000*(STATES
        [0] - 50.0000)/CONSTANTS[49]));
292 ALGEBRAIC[22] = CONSTANTS[49]*log(CONSTANTS[11]/STATES[2]);
293 ALGEBRAIC[24] = CONSTANTS[23]*CONSTANTS[11]/(CONSTANTS[11]+CONSTANTS[24])*(STATES[0]
         - ALGEBRAIC[22])/(1.00000+exp( (STATES[0] - ALGEBRAIC[22] - 10.0000)*2.00000/
        CONSTANTS[49]));
294 ALGEBRAIC[23] = CONSTANTS[22]*(STATES[0] - ALGEBRAIC[22]);
295 ALGEBRAIC[14] = CONSTANTS[13]*CONSTANTS[11]/(CONSTANTS[14]+CONSTANTS[11])*STATES
        [1]/(CONSTANTS[15]+STATES[1]);
296 ALGEBRAIC[31] = 0.00200000*CONSTANTS[25]*STATES[7]*ALGEBRAIC[29]*(STATES[0] -
        50.0000)/CONSTANTS[49]/(1.00000 - exp( - 1.00000*(STATES[0] - 50.0000)/CONSTANTS
        [49]))*( STATES[2]*exp(50.0000/CONSTANTS[49]) - CONSTANTS[11]*exp( - 1.00000*(
        STATES[0] - 50.0000)/CONSTANTS[49]));
297 ALGEBRAIC[16] = CONSTANTS[49]*log(CONSTANTS[10]/STATES[1]);
298 ALGEBRAIC[18] = CONSTANTS[16]*(STATES[0] - ALGEBRAIC[16]);
299 ALGEBRAIC[8] = CONSTANTS[49]*log((CONSTANTS[10]+ 0.120000*CONSTANTS[11])/(STATES[1]+
        0.120000*STATES[2]));
300 ALGEBRAIC[12] = CONSTANTS[9]*pow(STATES[3], 3.00000)*STATES[4]*(STATES[0] - ALGEBRAIC
        [8]);
```

```
301 ALGEBRAIC[33] = 0.0100000*CONSTANTS[25]*STATES[7]*ALGEBRAIC[29]*(STATES[0] −
      50.0000)/CONSTANTS[49]/(1.00000 − exp( − 1.00000*(STATES[0] − 50.0000)/CONSTANTS
      [49]))*( STATES[1]*exp(50.0000/CONSTANTS[49]) − CONSTANTS[10]*exp( − 1.00000*(
      STATES[0] − 50.0000)/CONSTANTS[49]));
302 ALGEBRAIC[35] = ALGEBRAIC[30]+ALGEBRAIC[31]+ALGEBRAIC[33];
303 ALGEBRAIC[3] = (VOI>=CONSTANTS[4]&&VOI<=CONSTANTS[5]&&VOI − CONSTANTS[4] − floor((VOI
       − CONSTANTS[4])/CONSTANTS[6])*CONSTANTS[6]<=CONSTANTS[7] ? CONSTANTS[8] :
      0.00000);
304 ALGEBRAIC[32] = STATES[6]+ STATES[9]*CONSTANTS[51]+ CONSTANTS[28]*CONSTANTS[29]+
      CONSTANTS[28];
305 ALGEBRAIC[34] = STATES[6]/ALGEBRAIC[32]*CONSTANTS[31] − STATES[9]*CONSTANTS[51]/
      ALGEBRAIC[32]*CONSTANTS[32];
306 ALGEBRAIC[37] = (STATES[9] − STATES[12])*CONSTANTS[35];
307 ALGEBRAIC[36] = ( pow(STATES[10]/(STATES[10]+0.250000), 2.00000)*CONSTANTS[34]+
      CONSTANTS[33])*STATES[12];
308 ALGEBRAIC[38] = CONSTANTS[40]*STATES[6]*(CONSTANTS[38] − STATES[13]) − CONSTANTS
      [41]*STATES[13];
309 ALGEBRAIC[39] = CONSTANTS[42]*STATES[6]*(CONSTANTS[39] − STATES[14]) − CONSTANTS
      [43]*STATES[14];
310 }
```

A.4 TNNP MODEL

A.4.1 *Mathematics*

"environment" component

This component has no equations.

"membrane" component

$$\frac{\mathrm{d}(V)}{\mathrm{d}(time)} = \frac{-(1)}{1} * (i_K1 + i_to + i_Kr + i_Ks + i_CaL + i_NaK + i_Na + i_b_Na + i_NaCa + i_b_Ca + i_p_K + i_p_Ca + i_stim)$$

"reversal_potentials" component

$$E_Na = \frac{R * T}{F} * \ln\left(\frac{Na_o}{Na_i}\right)$$

$$E_K = \frac{R * T}{F} * \ln\left(\frac{K_o}{K_i}\right)$$

$$E_Ks = \frac{R * T}{F} * \ln\left(\frac{K_o + P_kna * Na_o}{K_i + P_kna * Na_i}\right)$$

$$E_Ca = \frac{0.5 * R * T}{F} * \ln\left(\frac{Ca_o}{Ca_i}\right)$$

"inward_rectifier_potassium_current" component

$$alpha_K1 = \frac{0.1}{\left(1 + e^{0.06*((V - E_K) - 200)}\right)}$$

$$beta_K1 = \frac{\left(3 * e^{0.0002*((V - E_K) + 100)} + e^{0.1*((V - E_K) - 10)}\right)}{\left(1 + e^{-(0.5)*(V - E_K)}\right)}$$

$$xK1_inf = \frac{alpha_K1}{(alpha_K1 + beta_K1)}$$

$$i_K1 = g_K1 * xK1_inf * \sqrt{\frac{K_o}{5.4}} * (V - E_K)$$

"rapid_time_dependent_potassium_current" component

$$i_Kr = g_Kr * \sqrt{\frac{K_o}{5.4}} * Xr1 * Xr2 * (V - E_K)$$

"rapid_time_dependent_potassium_current_Xr1_gate" component

$$xr1_inf = \frac{1}{\left(1 + e^{\frac{(-(26) - V)}{7}}\right)}$$

$$alpha_xr1 = \frac{450}{\left(1 + e^{\frac{(-(45) - V)}{10}}\right)}$$

$$beta_xr1 = \frac{6}{\left(1 + e^{\frac{(V + 30)}{11.5}}\right)}$$

$$tau_xr1 = 1 * alpha_xr1 * beta_xr1$$

$$\frac{d(Xr1)}{d(time)} = \frac{(xr1_inf - Xr1)}{tau_xr1}$$

"rapid_time_dependent_potassium_current_Xr2_gate" component

$$xr2_inf = \frac{1}{\left(1 + e^{\frac{(V + 88)}{24}}\right)}$$

$$alpha_xr2 = \frac{3}{\left(1 + e^{\frac{(-(60)-V)}{20}}\right)}$$

$$beta_xr2 = \frac{1.12}{\left(1 + e^{\frac{(V-60)}{20}}\right)}$$

$$tau_xr2 = 1 * alpha_xr2 * beta_xr2$$

$$\frac{\mathrm{d}(Xr2)}{\mathrm{d}(time)} = \frac{(xr2_inf - Xr2)}{tau_xr2}$$

"slow_time_dependent_potassium_current" component

$$i_Ks = g_Ks * (Xs)^2 * (V - E_Ks)$$

"slow_time_dependent_potassium_current_Xs_gate" component

$$xs_inf = \frac{1}{\left(1 + e^{\frac{(-(5)-V)}{14}}\right)}$$

$$alpha_xs = \frac{1100}{\sqrt{\left(1 + e^{\frac{(-(10)-V)}{6}}\right)}}$$

$$beta_xs = \frac{1}{\left(1 + e^{\frac{(V-60)}{20}}\right)}$$

$$tau_xs = 1 * alpha_xs * beta_xs$$

$$\frac{\mathrm{d}(Xs)}{\mathrm{d}(time)} = \frac{(xs_inf - Xs)}{tau_xs}$$

"fast_sodium_current" component

$$i_Na = g_Na * (m)^3 * h * j * (V - E_Na)$$

"fast_sodium_current_m_gate" component

$$m_inf = \frac{1}{\left(\left(1 + e^{\frac{(-(56.86)-V)}{9.03}}\right)\right)^2}$$

$$alpha_m = \frac{1}{\left(1 + e^{\frac{(-(60)-V)}{5}}\right)}$$

$$beta_m = \left(\frac{0.1}{\left(1 + e^{\frac{(V+35)}{5}}\right)} + \frac{0.1}{\left(1 + e^{\frac{(V-50)}{200}}\right)} \right)$$

$$tau_m = 1 * alpha_m * beta_m$$

$$\frac{\mathrm{d}(m)}{\mathrm{d}(time)} = \frac{(m_inf - m)}{tau_m}$$

"fast_sodium_current_h_gate" component

$$h_inf = \frac{1}{\left(\left(1 + e^{\frac{(V+71.55)}{7.43}}\right)\right)^2}$$

$$alpha_h = \begin{cases} 0.057 * e^{\frac{-((V+80))}{6.8}} & \text{if } V < -(40) \\ 0 & \text{otherwise.} \end{cases}$$

$$beta_h = \begin{cases} \left(2.7 * e^{0.079*V} + 310000 * e^{0.3485*V}\right) & \text{if } V < -(40) \\ \frac{0.77}{0.13*\left(1+e^{\frac{(V+10.66)}{-(11.1)}}\right)} & \text{otherwise.} \end{cases}$$

$$tau_h = \frac{1}{(alpha_h + beta_h)}$$

$$\frac{\mathrm{d}(h)}{\mathrm{d}(time)} = \frac{(h_inf - h)}{tau_h}$$

"fast_sodium_current_j_gate" component

$$j_inf = \frac{1}{\left(\left(1 + e^{\frac{(V+71.55)}{7.43}}\right)\right)^2}$$

$$alpha_j = \begin{cases} \frac{\left(-(25428)*e^{0.2444*V} - 6.948 - 6*e^{-(0.04391)*V}\right)*(V+37.78)}{\frac{1}{\left(1+e^{0.311*(V+79.23)}\right)}} & \text{if } V < -(40) \\ 0 & \text{otherwise.} \end{cases}$$

$$beta_j = \begin{cases} \frac{0.02424*e^{-(0.01052)*V}}{\left(1+e^{-(0.1378)*(V+40.14)}\right)} & \text{if } V < -(40) \\ \frac{0.6*e^{0.057*V}}{\left(1+e^{-(0.1)*(V+32)}\right)} & \text{otherwise.} \end{cases}$$

$$tau_j = \frac{1}{(alpha_j + beta_j)}$$

$$\frac{d(j)}{d(time)} = \frac{(j_inf - j)}{tau_j}$$

"sodium_background_current" component

$$i_b_Na = g_bna * (V - E_Na)$$

"L_type_Ca_current" component

$$i_CaL = \frac{\frac{g_CaL*d*f*fCa*4*V*(F)^2}{R*T} * \left(Ca_i * e^{\frac{2*V*F}{R*T}} - 0.341 * Ca_o\right)}{\left(e^{\frac{2*V*F}{R*T}} - 1\right)}$$

"L_type_Ca_current_d_gate" component

$$d_inf = \frac{1}{\left(1 + e^{\frac{(-(5)-V)}{7.5}}\right)}$$

$$alpha_d = \left(\frac{1.4}{\left(1 + e^{\frac{(-(35)-V)}{13}}\right)} + 0.25\right)$$

$$beta_d = \frac{1.4}{\left(1 + e^{\frac{(V+5)}{5}}\right)}$$

$$gamma_d = \frac{1}{\left(1 + e^{\frac{(50-V)}{20}}\right)}$$

$$tau_d = (1 * alpha_d * beta_d + gamma_d)$$

$$\frac{d(d)}{d(time)} = \frac{(d_inf - d)}{tau_d}$$

"L_type_Ca_current_f_gate" component

$$f_inf = \frac{1}{\left(1 + e^{\frac{(V+20)}{7}}\right)}$$

$$tau_f = \left(1125 * e^{\frac{-(((V+27))^2)}{240}} + 80 + \frac{165}{\left(1 + e^{\frac{(25-V)}{10}}\right)}\right)$$

$$\frac{d(f)}{d(time)} = \frac{(f_inf - f)}{tau_f}$$

"L_type_Ca_current_fCa_gate" component

$$alpha_fCa = \frac{1}{\left(1 + \left(\frac{Ca_i}{0.000325}\right)^8\right)}$$

$$beta_fCa = \frac{0.1}{\left(1 + e^{\frac{(Ca_i - 0.0005)}{0.0001}}\right)}$$

$$gama_fCa = \frac{0.2}{\left(1 + e^{\frac{(Ca_i - 0.00075)}{0.0008}}\right)}$$

$$fCa_inf = \frac{(alpha_fCa + beta_fCa + gama_fCa + 0.23)}{1.46}$$

$$tau_fCa = 2$$

$$d_fCa = \frac{(fCa_inf - fCa)}{tau_fCa}$$

$$\frac{d(fCa)}{d(time)} = \begin{cases} 0 & \text{if } (0.01 * d_fCa > 0) \wedge (V > -(60)) \\ d_fCa & \text{otherwise.} \end{cases}$$

"calcium_background_current" component

$$i_b_Ca = g_bca * (V - E_Ca)$$

"transient_outward_current" component

$$i_to = g_to * r * s * (V - E_K)$$

"transient_outward_current_s_gate" component

$$s_inf = \frac{1}{\left(1 + e^{\frac{(V+28)}{5}}\right)}$$

$$tau_s = \left(1000 * e^{\frac{-(((V+67))^2)}{1000}} + 8\right)$$

$$\frac{d(s)}{d(time)} = \frac{(s_inf - s)}{tau_s}$$

"transient_outward_current_r_gate" component

$$r_inf = \frac{1}{\left(1 + e^{\frac{(20-V)}{6}}\right)}$$

$$tau_r = \left(9.5 * e^{\frac{-(((V+40))^2)}{1800}} + 0.8\right)$$

$$\frac{\mathrm{d}(r)}{\mathrm{d}(time)} = \frac{(r_inf - r)}{tau_r}$$

"sodium_potassium_pump_current" component

$$i_NaK = \frac{\frac{\frac{P_NaK*K_o}{(K_o+K_mk)}*Na_i}{(Na_i+K_mNa)}}{\left(1 + 0.1245 * e^{\frac{-(0.1)*V*F}{R*T}} + 0.0353 * e^{\frac{-(V)*F}{R*T}}\right)}$$

"sodium_calcium_exchanger_current" component

$$i_NaCa = \frac{K_NaCa * \left(e^{\frac{gamma*V*F}{R*T}} * (Na_i)^3 * Ca_o - e^{\frac{(gamma-1)*V*F}{R*T}} * (Na_o)^3 * Ca_i * alpha\right)}{\left((Km_Nai)^3 + (Na_o)^3\right) * (Km_Ca + Ca_o) * \left(1 + K_sat * e^{\frac{(gamma-1)*V*F}{R*T}}\right)}$$

"calcium_pump_current" component

$$i_p_Ca = \frac{g_pCa * Ca_i}{(Ca_i + K_pCa)}$$

"potassium_pump_current" component

$$i_p_K = \frac{g_pK * (V - E_K)}{\left(1 + e^{\frac{(25-V)}{5.98}}\right)}$$

"calcium_dynamics" component

$$i_rel = \left(\frac{a_rel * (Ca_SR)^2}{\left((b_rel)^2 + (Ca_SR)^2\right)} + c_rel\right) * d * g$$

$$i_up = \frac{Vmax_up}{\left(1 + \frac{(K_up)^2}{(Ca_i)^2}\right)}$$

$$i_leak = V_leak * (Ca_SR - Ca_i)$$

$$g_inf = \begin{cases} \frac{1}{\left(1+\left(\frac{Ca_i}{0.00035}\right)^{6}\right)} & \text{if } Ca_i < 0.00035 \\ \frac{1}{\left(1+\left(\frac{Ca_i}{0.00035}\right)^{16}\right)} & \text{otherwise.} \end{cases}$$

$$d_g = \frac{(g_inf - g)}{tau_g}$$

$$\frac{\mathrm{d}(g)}{\mathrm{d}(time)} = \begin{cases} 0 & \text{if } (0.01 * d_g > 0) \wedge (V > -(60)) \\ d_g & \text{otherwise.} \end{cases}$$

$$Ca_i_bufc = \frac{1}{\left(1 + \frac{Buf_c * K_buf_c}{((Ca_i + K_buf_c))^{2}}\right)}$$

$$Ca_sr_bufsr = \frac{1}{\left(1 + \frac{Buf_sr * K_buf_sr}{((Ca_SR + K_buf_sr))^{2}}\right)}$$

$$\frac{\mathrm{d}(Ca_i)}{\mathrm{d}(time)} = Ca_i_bufc * (((i_leak - i_up) + i_rel) - \frac{1 * ((i_CaL + i_b_Ca + i_p_Ca) - 2 * i_NaCa)}{2 * 1 * V_c * F} * Cm)$$

$$\frac{\mathrm{d}(Ca_SR)}{\mathrm{d}(time)} = \frac{Ca_sr_bufsr * V_c}{V_sr} * (i_up - (i_rel + i_leak))$$

"sodium_dynamics" component

$$\frac{\mathrm{d}(Na_i)}{\mathrm{d}(time)} = \frac{-(1) * (i_Na + i_b_Na + 3 * i_NaK + 3 * i_NaCa)}{1 * V_c * F} * Cm$$

"potassium_dynamics" component

$$\frac{\mathrm{d}(K_i)}{\mathrm{d}(time)} = \frac{-(1) * ((i_K1 + i_to + i_Kr + i_Ks + i_p_K + i_stim) - 2 * i_NaK)}{1 * V_c * F} * Cm$$

"stimulus_protocol" component

$$i_stim = \begin{cases} i_stimAmplitude & \text{if } (time \geq i_stimStart) \wedge (time \leq i_stimEnd) \wedge time \\ & -i_stimStart - \left(\left\lfloor \frac{time - i_stimStart}{i_stimPeriod} \right\rfloor i_stimPeriod \right) \\ & \leqq i_stimPulseDuration \\ 0 & \text{otherwise.} \end{cases}$$

A.4.2 *C-code Representation*

```
/*
   There are a total of 67 entries in the algebraic variable array.
   There are a total of 17 entries in each of the rate and state variable arrays.
   There are a total of 47 entries in the constant variable array.
 */
/*
 * VOI is time in component environment (millisecond).
 * STATES[0] is V in component membrane (millivolt).
 * CONSTANTS[0] is R in component membrane (joule_per_mole_kelvin).
 * CONSTANTS[1] is T in component membrane (kelvin).
 * CONSTANTS[2] is F in component membrane (coulomb_per_millimole).
 * CONSTANTS[3] is Cm in component membrane (microF).
 * CONSTANTS[4] is V_c in component membrane (micrometre3).
 * ALGEBRAIC[49] is i_K1 in component inward_rectifier_potassium_current (
     picoA_per_picoF).
 * ALGEBRAIC[56] is i_to in component transient_outward_current (picoA_per_picoF).
 * ALGEBRAIC[50] is i_Kr in component rapid_time_dependent_potassium_current (
     picoA_per_picoF).
 * ALGEBRAIC[51] is i_Ks in component slow_time_dependent_potassium_current (
     picoA_per_picoF).
 * ALGEBRAIC[54] is i_CaL in component L_type_Ca_current (picoA_per_picoF).
 * ALGEBRAIC[57] is i_NaK in component sodium_potassium_pump_current (picoA_per_picoF)
     .
 * ALGEBRAIC[52] is i_Na in component fast_sodium_current (picoA_per_picoF).
 * ALGEBRAIC[53] is i_b_Na in component sodium_background_current (picoA_per_picoF).
 * ALGEBRAIC[58] is i_NaCa in component sodium_calcium_exchanger_current (
     picoA_per_picoF).
 * ALGEBRAIC[55] is i_b_Ca in component calcium_background_current (picoA_per_picoF).
 * ALGEBRAIC[60] is i_p_K in component potassium_pump_current (picoA_per_picoF).
 * ALGEBRAIC[59] is i_p_Ca in component calcium_pump_current (picoA_per_picoF).
 * ALGEBRAIC[62] is i_stim in component stimulus_protocol (picoA_per_picoF).
 * ALGEBRAIC[0] is E_Na in component reversal_potentials (millivolt).
 * ALGEBRAIC[13] is E_K in component reversal_potentials (millivolt).
```

```
29  * ALGEBRAIC[26] is E_Ks in component reversal_potentials (millivolt).
30  * ALGEBRAIC[35] is E_Ca in component reversal_potentials (millivolt).
31  * CONSTANTS[5] is P_kna in component reversal_potentials (dimensionless).
32  * CONSTANTS[6] is K_o in component potassium_dynamics (millimolar).
33  * CONSTANTS[7] is Na_o in component sodium_dynamics (millimolar).
34  * STATES[1] is K_i in component potassium_dynamics (millimolar).
35  * STATES[2] is Na_i in component sodium_dynamics (millimolar).
36  * CONSTANTS[8] is Ca_o in component calcium_dynamics (millimolar).
37  * STATES[3] is Ca_i in component calcium_dynamics (millimolar).
38  * CONSTANTS[9] is g_K1 in component inward_rectifier_potassium_current (
      nanoS_per_picoF).
39  * ALGEBRAIC[48] is xK1_inf in component inward_rectifier_potassium_current (
      dimensionless).
40  * ALGEBRAIC[44] is alpha_K1 in component inward_rectifier_potassium_current (
      dimensionless).
41  * ALGEBRAIC[47] is beta_K1 in component inward_rectifier_potassium_current (
      dimensionless).
42  * CONSTANTS[10] is g_Kr in component rapid_time_dependent_potassium_current (
      nanoS_per_picoF).
43  * STATES[4] is Xr1 in component rapid_time_dependent_potassium_current_Xr1_gate (
      dimensionless).
44  * STATES[5] is Xr2 in component rapid_time_dependent_potassium_current_Xr2_gate (
      dimensionless).
45  * ALGEBRAIC[1] is xr1_inf in component
      rapid_time_dependent_potassium_current_Xr1_gate (dimensionless).
46  * ALGEBRAIC[14] is alpha_xr1 in component
      rapid_time_dependent_potassium_current_Xr1_gate (dimensionless).
47  * ALGEBRAIC[27] is beta_xr1 in component
      rapid_time_dependent_potassium_current_Xr1_gate (dimensionless).
48  * ALGEBRAIC[36] is tau_xr1 in component
      rapid_time_dependent_potassium_current_Xr1_gate (millisecond).
49  * ALGEBRAIC[2] is xr2_inf in component
      rapid_time_dependent_potassium_current_Xr2_gate (dimensionless).
50  * ALGEBRAIC[15] is alpha_xr2 in component
      rapid_time_dependent_potassium_current_Xr2_gate (dimensionless).
51  * ALGEBRAIC[28] is beta_xr2 in component
      rapid_time_dependent_potassium_current_Xr2_gate (dimensionless).
52  * ALGEBRAIC[37] is tau_xr2 in component
      rapid_time_dependent_potassium_current_Xr2_gate (millisecond).
53  * CONSTANTS[11] is g_Ks in component slow_time_dependent_potassium_current (
      nanoS_per_picoF).
54  * STATES[6] is Xs in component slow_time_dependent_potassium_current_Xs_gate (
      dimensionless).
55  * ALGEBRAIC[3] is xs_inf in component slow_time_dependent_potassium_current_Xs_gate (
      dimensionless).
56  * ALGEBRAIC[16] is alpha_xs in component
      slow_time_dependent_potassium_current_Xs_gate (dimensionless).
57  * ALGEBRAIC[29] is beta_xs in component slow_time_dependent_potassium_current_Xs_gate
       (dimensionless).
```

```
58  * ALGEBRAIC[38] is tau_xs in component slow_time_dependent_potassium_current_Xs_gate
        (millisecond).
59  * CONSTANTS[12] is g_Na in component fast_sodium_current (nanoS_per_picoF).
60  * STATES[7] is m in component fast_sodium_current_m_gate (dimensionless).
61  * STATES[8] is h in component fast_sodium_current_h_gate (dimensionless).
62  * STATES[9] is j in component fast_sodium_current_j_gate (dimensionless).
63  * ALGEBRAIC[4] is m_inf in component fast_sodium_current_m_gate (dimensionless).
64  * ALGEBRAIC[17] is alpha_m in component fast_sodium_current_m_gate (dimensionless).
65  * ALGEBRAIC[30] is beta_m in component fast_sodium_current_m_gate (dimensionless).
66  * ALGEBRAIC[39] is tau_m in component fast_sodium_current_m_gate (millisecond).
67  * ALGEBRAIC[5] is h_inf in component fast_sodium_current_h_gate (dimensionless).
68  * ALGEBRAIC[18] is alpha_h in component fast_sodium_current_h_gate (per_millisecond).
69  * ALGEBRAIC[31] is beta_h in component fast_sodium_current_h_gate (per_millisecond).
70  * ALGEBRAIC[40] is tau_h in component fast_sodium_current_h_gate (millisecond).
71  * ALGEBRAIC[6] is j_inf in component fast_sodium_current_j_gate (dimensionless).
72  * ALGEBRAIC[19] is alpha_j in component fast_sodium_current_j_gate (per_millisecond).
73  * ALGEBRAIC[32] is beta_j in component fast_sodium_current_j_gate (per_millisecond).
74  * ALGEBRAIC[41] is tau_j in component fast_sodium_current_j_gate (millisecond).
75  * CONSTANTS[13] is g_bna in component sodium_background_current (nanoS_per_picoF).
76  * CONSTANTS[14] is g_CaL in component L_type_Ca_current (litre_per_farad_second).
77  * STATES[10] is d in component L_type_Ca_current_d_gate (dimensionless).
78  * STATES[11] is f in component L_type_Ca_current_f_gate (dimensionless).
79  * STATES[12] is fCa in component L_type_Ca_current_fCa_gate (dimensionless).
80  * ALGEBRAIC[7] is d_inf in component L_type_Ca_current_d_gate (dimensionless).
81  * ALGEBRAIC[20] is alpha_d in component L_type_Ca_current_d_gate (dimensionless).
82  * ALGEBRAIC[33] is beta_d in component L_type_Ca_current_d_gate (dimensionless).
83  * ALGEBRAIC[42] is gamma_d in component L_type_Ca_current_d_gate (millisecond).
84  * ALGEBRAIC[45] is tau_d in component L_type_Ca_current_d_gate (millisecond).
85  * ALGEBRAIC[8] is f_inf in component L_type_Ca_current_f_gate (dimensionless).
86  * ALGEBRAIC[21] is tau_f in component L_type_Ca_current_f_gate (millisecond).
87  * ALGEBRAIC[9] is alpha_fCa in component L_type_Ca_current_fCa_gate (dimensionless).
88  * ALGEBRAIC[22] is beta_fCa in component L_type_Ca_current_fCa_gate (dimensionless).
89  * ALGEBRAIC[34] is gama_fCa in component L_type_Ca_current_fCa_gate (dimensionless).
90  * ALGEBRAIC[43] is fCa_inf in component L_type_Ca_current_fCa_gate (dimensionless).
91  * CONSTANTS[46] is tau_fCa in component L_type_Ca_current_fCa_gate (millisecond).
92  * ALGEBRAIC[46] is d_fCa in component L_type_Ca_current_fCa_gate (per_millisecond).
93  * CONSTANTS[15] is g_bca in component calcium_background_current (nanoS_per_picoF).
94  * CONSTANTS[16] is g_to in component transient_outward_current (nanoS_per_picoF).
95  * STATES[13] is s in component transient_outward_current_s_gate (dimensionless).
96  * STATES[14] is r in component transient_outward_current_r_gate (dimensionless).
97  * ALGEBRAIC[10] is s_inf in component transient_outward_current_s_gate (dimensionless
        ).
98  * ALGEBRAIC[23] is tau_s in component transient_outward_current_s_gate (millisecond).
99  * ALGEBRAIC[11] is r_inf in component transient_outward_current_r_gate (dimensionless
        ).
100 * ALGEBRAIC[24] is tau_r in component transient_outward_current_r_gate (millisecond).
101 * CONSTANTS[17] is P_NaK in component sodium_potassium_pump_current (picoA_per_picoF)
        .
102 * CONSTANTS[18] is K_mk in component sodium_potassium_pump_current (millimolar).
```

```
103  * CONSTANTS[19] is K_mNa in component sodium_potassium_pump_current (millimolar).
104  * CONSTANTS[20] is K_NaCa in component sodium_calcium_exchanger_current (
         picoA_per_picoF).
105  * CONSTANTS[21] is K_sat in component sodium_calcium_exchanger_current (dimensionless
         ).
106  * CONSTANTS[22] is alpha in component sodium_calcium_exchanger_current (dimensionless
         ).
107  * CONSTANTS[23] is gamma in component sodium_calcium_exchanger_current (dimensionless
         ).
108  * CONSTANTS[24] is Km_Ca in component sodium_calcium_exchanger_current (millimolar).
109  * CONSTANTS[25] is Km_Nai in component sodium_calcium_exchanger_current (millimolar).
110  * CONSTANTS[26] is g_pCa in component calcium_pump_current (picoA_per_picoF).
111  * CONSTANTS[27] is K_pCa in component calcium_pump_current (millimolar).
112  * CONSTANTS[28] is g_pK in component potassium_pump_current (nanoS_per_picoF).
113  * STATES[15] is Ca_SR in component calcium_dynamics (millimolar).
114  * ALGEBRAIC[61] is i_rel in component calcium_dynamics (millimolar_per_millisecond).
115  * ALGEBRAIC[63] is i_up in component calcium_dynamics (millimolar_per_millisecond).
116  * ALGEBRAIC[64] is i_leak in component calcium_dynamics (millimolar_per_millisecond).
117  * STATES[16] is g in component calcium_dynamics (dimensionless).
118  * CONSTANTS[29] is tau_g in component calcium_dynamics (millisecond).
119  * ALGEBRAIC[12] is g_inf in component calcium_dynamics (dimensionless).
120  * CONSTANTS[30] is a_rel in component calcium_dynamics (millimolar_per_millisecond).
121  * CONSTANTS[31] is b_rel in component calcium_dynamics (millimolar).
122  * CONSTANTS[32] is c_rel in component calcium_dynamics (millimolar_per_millisecond).
123  * CONSTANTS[33] is K_up in component calcium_dynamics (millimolar).
124  * CONSTANTS[34] is V_leak in component calcium_dynamics (per_millisecond).
125  * CONSTANTS[35] is Vmax_up in component calcium_dynamics (millimolar_per_millisecond)
         .
126  * ALGEBRAIC[65] is Ca_i_bufc in component calcium_dynamics (dimensionless).
127  * ALGEBRAIC[66] is Ca_sr_bufsr in component calcium_dynamics (dimensionless).
128  * CONSTANTS[36] is Buf_c in component calcium_dynamics (millimolar).
129  * CONSTANTS[37] is K_buf_c in component calcium_dynamics (millimolar).
130  * CONSTANTS[38] is Buf_sr in component calcium_dynamics (millimolar).
131  * CONSTANTS[39] is K_buf_sr in component calcium_dynamics (millimolar).
132  * CONSTANTS[40] is V_sr in component calcium_dynamics (micrometre3).
133  * ALGEBRAIC[25] is d_g in component calcium_dynamics (per_millisecond).
134  * CONSTANTS[41] is i_stimStart in component stimulus_protocol (millisecond).
135  * CONSTANTS[42] is i_stimEnd in component stimulus_protocol (millisecond).
136  * CONSTANTS[43] is i_stimAmplitude in component stimulus_protocol (picoA_per_picoF).
137  * CONSTANTS[44] is i_stimPeriod in component stimulus_protocol (millisecond).
138  * CONSTANTS[45] is i_stimPulseDuration in component stimulus_protocol (millisecond).
139  * RATES[0] is d/dt V in component membrane (millivolt).
140  * RATES[4] is d/dt Xr1 in component rapid_time_dependent_potassium_current_Xr1_gate (
         dimensionless).
141  * RATES[5] is d/dt Xr2 in component rapid_time_dependent_potassium_current_Xr2_gate (
         dimensionless).
142  * RATES[6] is d/dt Xs in component slow_time_dependent_potassium_current_Xs_gate (
         dimensionless).
143  * RATES[7] is d/dt m in component fast_sodium_current_m_gate (dimensionless).
```

```
144  * RATES[8] is d/dt h in component fast_sodium_current_h_gate (dimensionless).
145  * RATES[9] is d/dt j in component fast_sodium_current_j_gate (dimensionless).
146  * RATES[10] is d/dt d in component L_type_Ca_current_d_gate (dimensionless).
147  * RATES[11] is d/dt f in component L_type_Ca_current_f_gate (dimensionless).
148  * RATES[12] is d/dt fCa in component L_type_Ca_current_fCa_gate (dimensionless).
149  * RATES[13] is d/dt s in component transient_outward_current_s_gate (dimensionless).
150  * RATES[14] is d/dt r in component transient_outward_current_r_gate (dimensionless).
151  * RATES[16] is d/dt g in component calcium_dynamics (dimensionless).
152  * RATES[3] is d/dt Ca_i in component calcium_dynamics (millimolar).
153  * RATES[15] is d/dt Ca_SR in component calcium_dynamics (millimolar).
154  * RATES[2] is d/dt Na_i in component sodium_dynamics (millimolar).
155  * RATES[1] is d/dt K_i in component potassium_dynamics (millimolar).
156  */
157 void
158 initConsts(double* CONSTANTS, double* RATES, double *STATES)
159 {
160 STATES[0] = -86.2;
161 CONSTANTS[0] = 8314.472;
162 CONSTANTS[1] = 310;
163 CONSTANTS[2] = 96485.3415;
164 CONSTANTS[3] = 0.185;
165 CONSTANTS[4] = 0.016404;
166 CONSTANTS[5] = 0.03;
167 CONSTANTS[6] = 5.4;
168 CONSTANTS[7] = 140;
169 STATES[1] = 138.3;
170 STATES[2] = 11.6;
171 CONSTANTS[8] = 2;
172 STATES[3] = 0.0002;
173 CONSTANTS[9] = 5.405;
174 CONSTANTS[10] = 0.096;
175 STATES[4] = 0;
176 STATES[5] = 1;
177 CONSTANTS[11] = 0.245;
178 STATES[6] = 0;
179 CONSTANTS[12] = 14.838;
180 STATES[7] = 0;
181 STATES[8] = 0.75;
182 STATES[9] = 0.75;
183 CONSTANTS[13] = 0.00029;
184 CONSTANTS[14] = 0.000175;
185 STATES[10] = 0;
186 STATES[11] = 1;
187 STATES[12] = 1;
188 CONSTANTS[15] = 0.000592;
189 CONSTANTS[16] = 0.073;
190 STATES[13] = 1;
191 STATES[14] = 0;
192 CONSTANTS[17] = 1.362;
```

```
193 CONSTANTS[18] = 1;
194 CONSTANTS[19] = 40;
195 CONSTANTS[20] = 1000;
196 CONSTANTS[21] = 0.1;
197 CONSTANTS[22] = 2.5;
198 CONSTANTS[23] = 0.35;
199 CONSTANTS[24] = 1.38;
200 CONSTANTS[25] = 87.5;
201 CONSTANTS[26] = 0.825;
202 CONSTANTS[27] = 0.0005;
203 CONSTANTS[28] = 0.0146;
204 STATES[15] = 0.2;
205 STATES[16] = 1;
206 CONSTANTS[29] = 2;
207 CONSTANTS[30] = 0.016464;
208 CONSTANTS[31] = 0.25;
209 CONSTANTS[32] = 0.008232;
210 CONSTANTS[33] = 0.00025;
211 CONSTANTS[34] = 8e-5;
212 CONSTANTS[35] = 0.000425;
213 CONSTANTS[36] = 0.15;
214 CONSTANTS[37] = 0.001;
215 CONSTANTS[38] = 10;
216 CONSTANTS[39] = 0.3;
217 CONSTANTS[40] = 0.001094;
218 CONSTANTS[41] = 100;
219 CONSTANTS[42] = 50000;
220 CONSTANTS[43] = -52;
221 CONSTANTS[44] = 1000;
222 CONSTANTS[45] = 1;
223 CONSTANTS[46] = 2.00000;
224 }
225 void
226 computeRates(double VOI, double* CONSTANTS, double* RATES, double* STATES, double*
        ALGEBRAIC)
227 {
228 ALGEBRAIC[8] = 1.00000/(1.00000+exp((STATES[0]+20.0000)/7.00000));
229 ALGEBRAIC[21] =  1125.00*exp(- pow(STATES[0]+27.0000, 2.00000)/240.000)
        +80.0000+165.000/(1.00000+exp((25.0000 - STATES[0])/10.0000));
230 RATES[11] = (ALGEBRAIC[8] - STATES[11])/ALGEBRAIC[21];
231 ALGEBRAIC[10] = 1.00000/(1.00000+exp((STATES[0]+28.0000)/5.00000));
232 ALGEBRAIC[23] =  1000.00*exp(- pow(STATES[0]+67.0000, 2.00000)/1000.00)+8.00000;
233 RATES[13] = (ALGEBRAIC[10] - STATES[13])/ALGEBRAIC[23];
234 ALGEBRAIC[11] = 1.00000/(1.00000+exp((20.0000 - STATES[0])/6.00000));
235 ALGEBRAIC[24] =  9.50000*exp(- pow(STATES[0]+40.0000, 2.00000)/1800.00)+0.800000;
236 RATES[14] = (ALGEBRAIC[11] - STATES[14])/ALGEBRAIC[24];
237 ALGEBRAIC[12] = (STATES[3]<0.000350000 ? 1.00000/(1.00000+pow(STATES[3]/0.000350000,
        6.00000)) : 1.00000/(1.00000+pow(STATES[3]/0.000350000, 16.0000)));
238 ALGEBRAIC[25] = (ALGEBRAIC[12] - STATES[16])/CONSTANTS[29];
```

```
239 RATES[16] = ( 0.0100000*ALGEBRAIC[25]>0.00000&&STATES[0]>- 60.0000 ? 0.00000 :
        ALGEBRAIC[25]);
240 ALGEBRAIC[1] = 1.00000/(1.00000+exp((- 26.0000 - STATES[0])/7.00000));
241 ALGEBRAIC[14] = 450.000/(1.00000+exp((- 45.0000 - STATES[0])/10.0000));
242 ALGEBRAIC[27] = 6.00000/(1.00000+exp((STATES[0]+30.0000)/11.5000));
243 ALGEBRAIC[36] =  1.00000*ALGEBRAIC[14]*ALGEBRAIC[27];
244 RATES[4] = (ALGEBRAIC[1] - STATES[4])/ALGEBRAIC[36];
245 ALGEBRAIC[2] = 1.00000/(1.00000+exp((STATES[0]+88.0000)/24.0000));
246 ALGEBRAIC[15] = 3.00000/(1.00000+exp((- 60.0000 - STATES[0])/20.0000));
247 ALGEBRAIC[28] = 1.12000/(1.00000+exp((STATES[0] - 60.0000)/20.0000));
248 ALGEBRAIC[37] =  1.00000*ALGEBRAIC[15]*ALGEBRAIC[28];
249 RATES[5] = (ALGEBRAIC[2] - STATES[5])/ALGEBRAIC[37];
250 ALGEBRAIC[3] = 1.00000/(1.00000+exp((- 5.00000 - STATES[0])/14.0000));
251 ALGEBRAIC[16] = 1100.00/ pow((1.00000+exp((- 10.0000 - STATES[0])/6.00000)), 1.0 / 2);
252 ALGEBRAIC[29] = 1.00000/(1.00000+exp((STATES[0] - 60.0000)/20.0000));
253 ALGEBRAIC[38] =  1.00000*ALGEBRAIC[16]*ALGEBRAIC[29];
254 RATES[6] = (ALGEBRAIC[3] - STATES[6])/ALGEBRAIC[38];
255 ALGEBRAIC[4] = 1.00000/pow(1.00000+exp((- 56.8600 - STATES[0])/9.03000), 2.00000);
256 ALGEBRAIC[17] = 1.00000/(1.00000+exp((- 60.0000 - STATES[0])/5.00000));
257 ALGEBRAIC[30] = 0.100000/(1.00000+exp((STATES[0]+35.0000)/5.00000))+0.100000/(1.00000+
        exp((STATES[0] - 50.0000)/200.000));
258 ALGEBRAIC[39] =  1.00000*ALGEBRAIC[17]*ALGEBRAIC[30];
259 RATES[7] = (ALGEBRAIC[4] - STATES[7])/ALGEBRAIC[39];
260 ALGEBRAIC[5] = 1.00000/pow(1.00000+exp((STATES[0]+71.5500)/7.43000), 2.00000);
261 ALGEBRAIC[18] = (STATES[0]<- 40.0000 ?  0.0570000*exp(- (STATES[0]+80.0000)/6.80000) :
         0.00000);
262 ALGEBRAIC[31] = (STATES[0]<- 40.0000 ?  2.70000*exp( 0.0790000*STATES[0])+ 310000.*exp
        ( 0.348500*STATES[0]) : 0.770000/ 0.130000*(1.00000+exp((STATES[0]+10.6600)/-
        11.1000)));
263 ALGEBRAIC[40] = 1.00000/(ALGEBRAIC[18]+ALGEBRAIC[31]);
264 RATES[8] = (ALGEBRAIC[5] - STATES[8])/ALGEBRAIC[40];
265 ALGEBRAIC[6] = 1.00000/pow(1.00000+exp((STATES[0]+71.5500)/7.43000), 2.00000);
266 ALGEBRAIC[19] = (STATES[0]<- 40.0000 ?  ( - 25428.0*exp( 0.244400*STATES[0]) -
        6.94800e-06*exp( - 0.0439100*STATES[0]))*(STATES[0]+37.7800)/1.00000/(1.00000+exp(
         0.311000*(STATES[0]+79.2300))) : 0.00000);
267 ALGEBRAIC[32] = (STATES[0]<- 40.0000 ?  0.0242400*exp( - 0.0105200*STATES[0])
        /(1.00000+exp( - 0.137800*(STATES[0]+40.1400))) :  0.600000*exp( 0.0570000*STATES
        [0])/(1.00000+exp( - 0.100000*(STATES[0]+32.0000))));
268 ALGEBRAIC[41] = 1.00000/(ALGEBRAIC[19]+ALGEBRAIC[32]);
269 RATES[9] = (ALGEBRAIC[6] - STATES[9])/ALGEBRAIC[41];
270 ALGEBRAIC[7] = 1.00000/(1.00000+exp((- 5.00000 - STATES[0])/7.50000));
271 ALGEBRAIC[20] = 1.40000/(1.00000+exp((- 35.0000 - STATES[0])/13.0000))+0.250000;
272 ALGEBRAIC[33] = 1.40000/(1.00000+exp((STATES[0]+5.00000)/5.00000));
273 ALGEBRAIC[42] = 1.00000/(1.00000+exp((50.0000 - STATES[0])/20.0000));
274 ALGEBRAIC[45] =  1.00000*ALGEBRAIC[20]*ALGEBRAIC[33]+ALGEBRAIC[42];
275 RATES[10] = (ALGEBRAIC[7] - STATES[10])/ALGEBRAIC[45];
276 ALGEBRAIC[9] = 1.00000/(1.00000+pow(STATES[3]/0.000325000, 8.00000));
277 ALGEBRAIC[22] = 0.100000/(1.00000+exp((STATES[3] - 0.000500000)/0.000100000));
278 ALGEBRAIC[34] = 0.200000/(1.00000+exp((STATES[3] - 0.000750000)/0.000800000));
```

```
279 ALGEBRAIC[43] = (ALGEBRAIC[9]+ALGEBRAIC[22]+ALGEBRAIC[34]+0.230000)/1.46000;
280 ALGEBRAIC[46] = (ALGEBRAIC[43] - STATES[12])/CONSTANTS[46];
281 RATES[12] = ( 0.0100000*ALGEBRAIC[46]>0.00000&&STATES[0]>- 60.0000 ? 0.00000 :
        ALGEBRAIC[46]);
282 ALGEBRAIC[57] = CONSTANTS[17]*CONSTANTS[6]/(CONSTANTS[6]+CONSTANTS[18])*STATES[2]/(
        STATES[2]+CONSTANTS[19])/(1.00000+ 0.124500*exp( - 0.100000*STATES[0]*CONSTANTS
        [2]/( CONSTANTS[0]*CONSTANTS[1]))+ 0.0353000*exp( - STATES[0]*CONSTANTS[2]/(
        CONSTANTS[0]*CONSTANTS[1])));
283 ALGEBRAIC[0] = CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2]*log(CONSTANTS[7]/STATES[2]);
284 ALGEBRAIC[52] = CONSTANTS[12]*pow(STATES[7], 3.00000)*STATES[8]*STATES[9]*(STATES[0]
        - ALGEBRAIC[0]);
285 ALGEBRAIC[53] = CONSTANTS[13]*(STATES[0] - ALGEBRAIC[0]);
286 ALGEBRAIC[58] = CONSTANTS[20]*( exp( CONSTANTS[23]*STATES[0]*CONSTANTS[2]/( CONSTANTS
        [0]*CONSTANTS[1]))*pow(STATES[2], 3.00000)*CONSTANTS[8] - exp( (CONSTANTS[23] -
        1.00000)*STATES[0]*CONSTANTS[2]/( CONSTANTS[0]*CONSTANTS[1]))*pow(CONSTANTS[7],
        3.00000)*STATES[3]*CONSTANTS[22])/( (pow(CONSTANTS[25], 3.00000)+pow(CONSTANTS[7],
         3.00000))*(CONSTANTS[24]+CONSTANTS[8])*(1.00000+ CONSTANTS[21]*exp( (CONSTANTS
        [23] - 1.00000)*STATES[0]*CONSTANTS[2]/( CONSTANTS[0]*CONSTANTS[1]))));
287 RATES[2] = - 1.00000*(ALGEBRAIC[52]+ALGEBRAIC[53]+ 3.00000*ALGEBRAIC[57]+ 3.00000*
        ALGEBRAIC[58])/( 1.00000*CONSTANTS[4]*CONSTANTS[2])*CONSTANTS[3];
288 ALGEBRAIC[13] = CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2]*log(CONSTANTS[6]/STATES[1]);
289 ALGEBRAIC[44] = 0.100000/(1.00000+exp( 0.0600000*(STATES[0] - ALGEBRAIC[13] - 200.000)
        ));
290 ALGEBRAIC[47] = ( 3.00000*exp( 0.000200000*(STATES[0] - ALGEBRAIC[13]+100.000))+exp(
        0.100000*(STATES[0] - ALGEBRAIC[13] - 10.0000)))/(1.00000+exp( - 0.500000*(STATES
        [0] - ALGEBRAIC[13])));
291 ALGEBRAIC[48] = ALGEBRAIC[44]/(ALGEBRAIC[44]+ALGEBRAIC[47]);
292 ALGEBRAIC[49] = CONSTANTS[9]*ALGEBRAIC[48]* pow(CONSTANTS[6]/5.40000, 1.0 / 2)*(
        STATES[0] - ALGEBRAIC[13]);
293 ALGEBRAIC[56] = CONSTANTS[16]*STATES[14]*STATES[13]*(STATES[0] - ALGEBRAIC[13]);
294 ALGEBRAIC[50] = CONSTANTS[10]* pow(CONSTANTS[6]/5.40000, 1.0 / 2)*STATES[4]*STATES
        [5]*(STATES[0] - ALGEBRAIC[13]);
295 ALGEBRAIC[26] = CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2]*log((CONSTANTS[6]+ CONSTANTS
        [5]*CONSTANTS[7])/(STATES[1]+ CONSTANTS[5]*STATES[2]));
296 ALGEBRAIC[51] = CONSTANTS[11]*pow(STATES[6], 2.00000)*(STATES[0] - ALGEBRAIC[26]);
297 ALGEBRAIC[54] = CONSTANTS[14]*STATES[10]*STATES[11]*STATES[12]*4.00000*STATES[0]*pow
        (CONSTANTS[2], 2.00000)/( CONSTANTS[0]*CONSTANTS[1])*( STATES[3]*exp( 2.00000*
        STATES[0]*CONSTANTS[2]/( CONSTANTS[0]*CONSTANTS[1])) - 0.341000*CONSTANTS[8])/(
        exp( 2.00000*STATES[0]*CONSTANTS[2]/( CONSTANTS[0]*CONSTANTS[1])) - 1.00000);
298 ALGEBRAIC[35] = 0.500000*CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2]*log(CONSTANTS[8]/
        STATES[3]);
299 ALGEBRAIC[55] = CONSTANTS[15]*(STATES[0] - ALGEBRAIC[35]);
300 ALGEBRAIC[60] = CONSTANTS[28]*(STATES[0] - ALGEBRAIC[13])/(1.00000+exp((25.0000 -
        STATES[0])/5.98000));
301 ALGEBRAIC[59] = CONSTANTS[26]*STATES[3]/(STATES[3]+CONSTANTS[27]);
302 ALGEBRAIC[62] = (VOI>=CONSTANTS[41]&&VOI<=CONSTANTS[42]&&VOI - CONSTANTS[41] - floor
        ((VOI - CONSTANTS[41])/CONSTANTS[44])*CONSTANTS[44]<=CONSTANTS[45] ? CONSTANTS[43]
         : 0.00000);
```

```
303 RATES[0] = - 1.00000/1.00000*(ALGEBRAIC[49]+ALGEBRAIC[56]+ALGEBRAIC[50]+ALGEBRAIC
        [51]+ALGEBRAIC[54]+ALGEBRAIC[57]+ALGEBRAIC[52]+ALGEBRAIC[53]+ALGEBRAIC[58]+
        ALGEBRAIC[55]+ALGEBRAIC[60]+ALGEBRAIC[59]+ALGEBRAIC[62]);
304 RATES[1] = - 1.00000*(ALGEBRAIC[49]+ALGEBRAIC[56]+ALGEBRAIC[50]+ALGEBRAIC[51]+
        ALGEBRAIC[60]+ALGEBRAIC[62] - 2.00000*ALGEBRAIC[57])/( 1.00000*CONSTANTS[4]*
        CONSTANTS[2])*CONSTANTS[3];
305 ALGEBRAIC[61] = ( CONSTANTS[30]*pow(STATES[15], 2.00000)/(pow(CONSTANTS[31], 2.00000)
        +pow(STATES[15], 2.00000))+CONSTANTS[32])*STATES[10]*STATES[16];
306 ALGEBRAIC[63] = CONSTANTS[35]/(1.00000+pow(CONSTANTS[33], 2.00000)/pow(STATES[3],
        2.00000));
307 ALGEBRAIC[64] = CONSTANTS[34]*(STATES[15] - STATES[3]);
308 ALGEBRAIC[65] = 1.00000/(1.00000+ CONSTANTS[36]*CONSTANTS[37]/pow(STATES[3]+CONSTANTS
        [37], 2.00000));
309 RATES[3] = ALGEBRAIC[65]*(ALGEBRAIC[64] - ALGEBRAIC[63]+ALGEBRAIC[61] - 1.00000*(
        ALGEBRAIC[54]+ALGEBRAIC[55]+ALGEBRAIC[59] - 2.00000*ALGEBRAIC[58])/(
        2.00000*1.00000*CONSTANTS[4]*CONSTANTS[2])*CONSTANTS[3]);
310 ALGEBRAIC[66] = 1.00000/(1.00000+ CONSTANTS[38]*CONSTANTS[39]/pow(STATES[15]+CONSTANTS
        [39], 2.00000));
311 RATES[15] = ALGEBRAIC[66]*CONSTANTS[4]/CONSTANTS[40]*(ALGEBRAIC[63] - ALGEBRAIC[61]+
        ALGEBRAIC[64]);
312 }
313 void
314 computeVariables(double VOI, double* CONSTANTS, double* RATES, double* STATES, double*
         ALGEBRAIC)
315 {
316 ALGEBRAIC[8] = 1.00000/(1.00000+exp((STATES[0]+20.0000)/7.00000));
317 ALGEBRAIC[21] = 1125.00*exp(- pow(STATES[0]+27.0000, 2.00000)/240.000)
        +80.0000+165.000/(1.00000+exp((25.0000 - STATES[0])/10.0000));
318 ALGEBRAIC[10] = 1.00000/(1.00000+exp((STATES[0]+28.0000)/5.00000));
319 ALGEBRAIC[23] = 1000.00*exp(- pow(STATES[0]+67.0000, 2.00000)/1000.00)+8.00000;
320 ALGEBRAIC[11] = 1.00000/(1.00000+exp((20.0000 - STATES[0])/6.00000));
321 ALGEBRAIC[24] = 9.50000*exp(- pow(STATES[0]+40.0000, 2.00000)/1800.00)+0.800000;
322 ALGEBRAIC[12] = (STATES[3]<0.000350000 ? 1.00000/(1.00000+pow(STATES[3]/0.000350000,
        6.00000)) : 1.00000/(1.00000+pow(STATES[3]/0.000350000, 16.0000)));
323 ALGEBRAIC[25] = (ALGEBRAIC[12] - STATES[16])/CONSTANTS[29];
324 ALGEBRAIC[1] = 1.00000/(1.00000+exp((- 26.0000 - STATES[0])/7.00000));
325 ALGEBRAIC[14] = 450.000/(1.00000+exp((- 45.0000 - STATES[0])/10.0000));
326 ALGEBRAIC[27] = 6.00000/(1.00000+exp((STATES[0]+30.0000)/11.5000));
327 ALGEBRAIC[36] = 1.00000*ALGEBRAIC[14]*ALGEBRAIC[27];
328 ALGEBRAIC[2] = 1.00000/(1.00000+exp((STATES[0]+88.0000)/24.0000));
329 ALGEBRAIC[15] = 3.00000/(1.00000+exp((- 60.0000 - STATES[0])/20.0000));
330 ALGEBRAIC[28] = 1.12000/(1.00000+exp((STATES[0] - 60.0000)/20.0000));
331 ALGEBRAIC[37] = 1.00000*ALGEBRAIC[15]*ALGEBRAIC[28];
332 ALGEBRAIC[3] = 1.00000/(1.00000+exp((- 5.00000 - STATES[0])/14.0000));
333 ALGEBRAIC[16] = 1100.00/ pow((1.00000+exp((- 10.0000 - STATES[0])/6.00000)), 1.0 / 2);
334 ALGEBRAIC[29] = 1.00000/(1.00000+exp((STATES[0] - 60.0000)/20.0000));
335 ALGEBRAIC[38] = 1.00000*ALGEBRAIC[16]*ALGEBRAIC[29];
336 ALGEBRAIC[4] = 1.00000/pow(1.00000+exp((- 56.8600 - STATES[0])/9.03000), 2.00000);
337 ALGEBRAIC[17] = 1.00000/(1.00000+exp((- 60.0000 - STATES[0])/5.00000));
```

```
338 ALGEBRAIC[30] = 0.100000/(1.00000+exp((STATES[0]+35.0000)/5.00000))+0.100000/(1.00000+
      exp((STATES[0] - 50.0000)/200.000));
339 ALGEBRAIC[39] =  1.00000*ALGEBRAIC[17]*ALGEBRAIC[30];
340 ALGEBRAIC[5] = 1.00000/pow(1.00000+exp((STATES[0]+71.5500)/7.43000), 2.00000);
341 ALGEBRAIC[18] = (STATES[0]<- 40.0000 ?  0.0570000*exp(- (STATES[0]+80.0000)/6.80000) :
       0.00000);
342 ALGEBRAIC[31] = (STATES[0]<- 40.0000 ?  2.70000*exp( 0.0790000*STATES[0])+ 310000.*exp
      ( 0.348500*STATES[0]) : 0.770000/ 0.130000*(1.00000+exp((STATES[0]+10.6600)/-
      11.1000)));
343 ALGEBRAIC[40] = 1.00000/(ALGEBRAIC[18]+ALGEBRAIC[31]);
344 ALGEBRAIC[6] = 1.00000/pow(1.00000+exp((STATES[0]+71.5500)/7.43000), 2.00000);
345 ALGEBRAIC[19] = (STATES[0]<- 40.0000 ?  ( - 25428.0*exp( 0.244400*STATES[0]) -
      6.94800e-06*exp( - 0.0439100*STATES[0]))*(STATES[0]+37.7800)/1.00000/(1.00000+exp(
       0.311000*(STATES[0]+79.2300))) : 0.00000);
346 ALGEBRAIC[32] = (STATES[0]<- 40.0000 ?  0.0242400*exp( - 0.0105200*STATES[0])
      /(1.00000+exp( - 0.137800*(STATES[0]+40.1400))) :  0.600000*exp( 0.0570000*STATES
      [0])/(1.00000+exp( - 0.100000*(STATES[0]+32.0000))));
347 ALGEBRAIC[41] = 1.00000/(ALGEBRAIC[19]+ALGEBRAIC[32]);
348 ALGEBRAIC[7] = 1.00000/(1.00000+exp((- 5.00000 - STATES[0])/7.50000));
349 ALGEBRAIC[20] = 1.40000/(1.00000+exp((- 35.0000 - STATES[0])/13.0000))+0.250000;
350 ALGEBRAIC[33] = 1.40000/(1.00000+exp((STATES[0]+5.00000)/5.00000));
351 ALGEBRAIC[42] = 1.00000/(1.00000+exp((50.0000 - STATES[0])/20.0000));
352 ALGEBRAIC[45] =  1.00000*ALGEBRAIC[20]*ALGEBRAIC[33]+ALGEBRAIC[42];
353 ALGEBRAIC[9] = 1.00000/(1.00000+pow(STATES[3]/0.000325000, 8.00000));
354 ALGEBRAIC[22] = 0.100000/(1.00000+exp((STATES[3] - 0.000500000)/0.000100000));
355 ALGEBRAIC[34] = 0.200000/(1.00000+exp((STATES[3] - 0.000750000)/0.000800000));
356 ALGEBRAIC[43] = (ALGEBRAIC[9]+ALGEBRAIC[22]+ALGEBRAIC[34]+0.230000)/1.46000;
357 ALGEBRAIC[46] = (ALGEBRAIC[43] - STATES[12])/CONSTANTS[46];
358 ALGEBRAIC[57] =   CONSTANTS[17]*CONSTANTS[6]/(CONSTANTS[6]+CONSTANTS[18])*STATES[2]/(
      STATES[2]+CONSTANTS[19])/(1.00000+ 0.124500*exp( - 0.100000*STATES[0]*CONSTANTS
      [2]/( CONSTANTS[0]*CONSTANTS[1]))+ 0.0353000*exp( - STATES[0]*CONSTANTS[2]/(
      CONSTANTS[0]*CONSTANTS[1])));
359 ALGEBRAIC[0] =   CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2]*log(CONSTANTS[7]/STATES[2]);
360 ALGEBRAIC[52] =  CONSTANTS[12]*pow(STATES[7], 3.00000)*STATES[8]*STATES[9]*(STATES[0]
      - ALGEBRAIC[0]);
361 ALGEBRAIC[53] =  CONSTANTS[13]*(STATES[0] - ALGEBRAIC[0]);
362 ALGEBRAIC[58] =  CONSTANTS[20]*( exp( CONSTANTS[23]*STATES[0]*CONSTANTS[2]/( CONSTANTS
      [0]*CONSTANTS[1]))*pow(STATES[2], 3.00000)*CONSTANTS[8] -  exp( (CONSTANTS[23] -
      1.00000)*STATES[0]*CONSTANTS[2]/( CONSTANTS[0]*CONSTANTS[1]))*pow(CONSTANTS[7],
      3.00000)*STATES[3]*CONSTANTS[22])/( (pow(CONSTANTS[25], 3.00000)+pow(CONSTANTS[7],
       3.00000))*(CONSTANTS[24]+CONSTANTS[8])*(1.00000+ CONSTANTS[21]*exp( (CONSTANTS
      [23] - 1.00000)*STATES[0]*CONSTANTS[2]/( CONSTANTS[0]*CONSTANTS[1]))));
363 ALGEBRAIC[13] =   CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2]*log(CONSTANTS[6]/STATES[1]);
364 ALGEBRAIC[44] = 0.100000/(1.00000+exp( 0.0600000*(STATES[0] - ALGEBRAIC[13] - 200.000)
      ));
365 ALGEBRAIC[47] = ( 3.00000*exp( 0.000200000*(STATES[0] - ALGEBRAIC[13]+100.000))+exp(
      0.100000*(STATES[0] - ALGEBRAIC[13] - 10.0000)))/(1.00000+exp( - 0.500000*(STATES
      [0] - ALGEBRAIC[13])));
366 ALGEBRAIC[48] = ALGEBRAIC[44]/(ALGEBRAIC[44]+ALGEBRAIC[47]);
```

```
367 ALGEBRAIC[49] = CONSTANTS[9]*ALGEBRAIC[48]* pow(CONSTANTS[6]/5.40000, 1.0 / 2)*(
        STATES[0] - ALGEBRAIC[13]);
368 ALGEBRAIC[56] = CONSTANTS[16]*STATES[14]*STATES[13]*(STATES[0] - ALGEBRAIC[13]);
369 ALGEBRAIC[50] = CONSTANTS[10]* pow(CONSTANTS[6]/5.40000, 1.0 / 2)*STATES[4]*STATES
        [5]*(STATES[0] - ALGEBRAIC[13]);
370 ALGEBRAIC[26] = CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2]*log((CONSTANTS[6]+ CONSTANTS
        [5]*CONSTANTS[7])/(STATES[1]+ CONSTANTS[5]*STATES[2]));
371 ALGEBRAIC[51] = CONSTANTS[11]*pow(STATES[6], 2.00000)*(STATES[0] - ALGEBRAIC[26]);
372 ALGEBRAIC[54] = CONSTANTS[14]*STATES[10]*STATES[11]*STATES[12]*4.00000*STATES[0]*pow
        (CONSTANTS[2], 2.00000)/( CONSTANTS[0]*CONSTANTS[1])*( STATES[3]*exp( 2.00000*
        STATES[0]*CONSTANTS[2]/( CONSTANTS[0]*CONSTANTS[1])) - 0.341000*CONSTANTS[8])/(
        exp( 2.00000*STATES[0]*CONSTANTS[2]/( CONSTANTS[0]*CONSTANTS[1])) - 1.00000);
373 ALGEBRAIC[35] = 0.500000*CONSTANTS[0]*CONSTANTS[1]/CONSTANTS[2]*log(CONSTANTS[8]/
        STATES[3]);
374 ALGEBRAIC[55] = CONSTANTS[15]*(STATES[0] - ALGEBRAIC[35]);
375 ALGEBRAIC[60] = CONSTANTS[28]*(STATES[0] - ALGEBRAIC[13])/(1.00000+exp((25.0000 -
        STATES[0])/5.98000));
376 ALGEBRAIC[59] = CONSTANTS[26]*STATES[3]/(STATES[3]+CONSTANTS[27]);
377 ALGEBRAIC[62] = (VOI>=CONSTANTS[41]&&VOI<=CONSTANTS[42]&&VOI - CONSTANTS[41] - floor
        ((VOI - CONSTANTS[41])/CONSTANTS[44])*CONSTANTS[44]<=CONSTANTS[45] ? CONSTANTS[43]
         : 0.00000);
378 ALGEBRAIC[61] = ( CONSTANTS[30]*pow(STATES[15], 2.00000)/(pow(CONSTANTS[31], 2.00000)
        +pow(STATES[15], 2.00000))+CONSTANTS[32])*STATES[10]*STATES[16];
379 ALGEBRAIC[63] = CONSTANTS[35]/(1.00000+pow(CONSTANTS[33], 2.00000)/pow(STATES[3],
        2.00000));
380 ALGEBRAIC[64] = CONSTANTS[34]*(STATES[15] - STATES[3]);
381 ALGEBRAIC[65] = 1.00000/(1.00000+ CONSTANTS[36]*CONSTANTS[37]/pow(STATES[3]+CONSTANTS
        [37], 2.00000));
382 ALGEBRAIC[66] = 1.00000/(1.00000+ CONSTANTS[38]*CONSTANTS[39]/pow(STATES[15]+CONSTANTS
        [39], 2.00000));
383 }
```

BIBLIOGRAPHY

[1] Alachiotis, N., and Stamatakis, A. Efficient floating-point logarithm unit for FPGAs. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), IEEE International Symposium on* (2010), IEEE, pp. 1–8. (Cited on page 34.)

[2] Altera Corporation. Differences in logic utilization between quartus ii & synplify report files. Tech. rep., November 2002. (Cited on page 137.)

[3] Altera Corporation. *DSP Builder Handbook*, 2012. (Cited on page 34.)

[4] Altera Corporation. Using PCI Express on DE4 Boards. Tech. rep., October 2012. (Cited on page 44.)

[5] Altera Corporation. Floating-Point IP Cores User Guide. Tech. rep., December 2014. (Cited on pages 22, 34, and 119.)

[6] Altera Corporation. IP Compiler for PCI Express User Guide. Tech. rep., August 2014. (Cited on pages 19 and 20.)

[7] Altera Corporation. Avalon interface specification. (Cited on page 19.)

[8] Altera Corporation. Qsys - Altera's System Integration Tool. http://www.altera.com/products/design-software/fpga-design/quartus-ii/quartus-ii-subscription-edition/qts-qsys.html, 2015. (Cited on pages 85 and 131.)

[9] Altera Corporation. Quartus II and ModelSim Design Flow. http://www.altera.com/products/design-software/model---simulation/modelsim-altera-software.html, 2015. (Cited on page 16.)

[10] Altera Corporation. Stratix 10 Overview. http://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html, 2015. (Cited on page 97.)

[11] Altera Corporation. Stratix IV FPGA ALM Logic Structure's 8-Input Fracturable LUT. http://www.altera.com/products/fpga/features/stxiv-alm-logic-structure.html, 2015. (Cited on pages 13 and 14.)

[12] Altera Corporation. Stratix IV FPGA family architecture. http://www.altera.com/products/fpga/stratix-series/stratix-iv/features.html, 2015. (Cited on page 14.)

[13] Altera Corporation. Stratix V Overview. http://www.altera.com/products/fpga/stratix-series/stratix-v/overview.html, 2015. (Cited on page 97.)

[14] Anderson, D., Shanley, T., and Budruk, R. *PCI express system architecture*. Addison-Wesley Professional, 2004. (Cited on page 17.)

[15] Ashenden, P. J. *The designer's guide to VHDL*, vol. 3. Morgan Kaufmann, 2010. (Cited on page 15.)

[16] Atkinson, K. E. *An introduction to numerical analysis*. John Wiley & Sons, 2008. (Cited on page 108.)

[17] Ausbrooks, R., Buswell, S., Carlisle, D., Chavchanidze, G., Dalmas, S., Devitt, S., Diaz, A., Dooley, S., Hunter, R., Ion, P., et al. Mathematical markup language (mathml) version 3.0. w3c candidate recommendation of 15 december 2009. *World Wide Web Consortium 13* (2009). (Cited on page 2.)

[18] Barroso, L. A. The price of performance. *Queue 3*, 7 (2005), 48–53. (Cited on page 1.)

[19] Bassingthwaighte, J., Butterworth, E., Jardine, B., Raymond, G., and Neal, M. JSim, an open-source modeling system for data analysis and reproducibility in research (733.1). *The FASEB Journal 28*, 1 Supplement (Apr. 2014), 733.1. (Cited on page 56.)

[20] Beeler, G. W., and Reuter, H. Reconstruction of the action potential of ventricular myocardial fibres. https://models.physiomeproject.org/e/1. CellML Author: Catherine Lloyd. (Cited on page 6.)

[21] Beeler, G. W., and Reuter, H. Reconstruction of the action potential of ventricular myocardial fibres. *The Journal of Physiology 268*, 1 (June 1977), 177–210. (Cited on pages 5, 89, and 132.)

[22] Betkaoui, B., Thomas, D., and Luk, W. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *2010 International Conference on Field-Programmable Technology (FPT)* (Dec. 2010), pp. 94–101. (Cited on page 57.)

[23] Birk, M., Zapf, M., Balzer, M., Ruiter, N., and Becker, J. A comprehensive comparison of GPU-and FPGA-based acceleration of reflection image reconstruction for 3d ultrasound computer tomography. *Journal of real-time image processing* 9, 1 (2014), 159–170. (Cited on page 55.)

[24] Bradley, C., Bowery, A., Britten, R., Budelmann, V., Camara, O., Christie, R., Cookson, A., Frangi, A. F., Gamage, T. B., Heidlauf, T., and others. OpenCMISS: a multi-physics & multi-scale computational infrastructure for the VPH/physiome project. *Progress in biophysics and molecular biology 107*, 1 (2011), 32–47. (Cited on pages 2, 9, 10, 32, 55, 56, 104, 106, and 109.)

[25] Brickley, D., and Guha, R. V. Resource description framework (rdf) schema specification 1.0: W3c candidate recommendation 27 march 2000. (Cited on page 2.)

[26] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the*

19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (New York, NY, USA, 2011), FPGA '11, ACM, pp. 33–36. (Cited on pages 58 and 107.)

[27] Chen, D., and Singh, D. Invited paper: Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering. In *2012 22nd International Conference on Field Programmable Logic and Applications (FPL)* (Aug. 2012), pp. 5–12. (Cited on page 57.)

[28] Chen, H., Sun, S., Aliprantis, D., and Zambreno, J. Dynamic simulation of electric machines on FPGA boards. In *Electric Machines and Drives Conference, 2009. IEMDC '09. IEEE International* (May 2009), pp. 1523–1528. (Cited on pages 57 and 106.)

[29] Cocke, J. Global common subexpression elimination. *ACM Sigplan Notices 5*, 7 (1970), 20–24. (Cited on page 107.)

[30] Cong, J., Fan, Y., Han, G., Jiang, W., and Zhang, Z. Platform-based behavior-level and system-level synthesis. In *SOC Conference, 2006 IEEE International* (Sept. 2006), pp. 199–202. (Cited on pages 58 and 107.)

[31] Coussy, P., Chavet, C., Bomel, P., Heller, D., Senn, E., and Martin, E. GAUT: A high-level synthesis tool for DSP applications. In *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, Jan. 2008, pp. 147–169. (Cited on pages 58 and 107.)

[32] Coussy, P., and Morawiec, A. *High-level synthesis*. Springer, 2010. (Cited on page 24.)

[33] CPLEX, I. I. V12.1: User's manual for cplex. *International Business Machines Corporation 46*, 53 (2009), 157. (Cited on page 125.)

[34] Cuellar, A. A., Lloyd, C. M., Nielsen, P. F., Bullivant, D. P., Nickerson, D. P., and Hunter, P. J. An overview of CellML 1.1, a biological model description language. *Simulation 79*, 12 (2003), 740–747. (Cited on pages 2, 3, 32, 55, 56, and 104.)

[35] Cullinan, C., Wyant, C., Frattesi, T., and Huang, X. Computing performance benchmarks among CPU, GPU, and FPGA. *Internet: www. wpi. edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking Final* (2013). (Cited on page 55.)

[36] de Dinechin, F., and others. *FloPoCo, a generator of arithmetic cores for FPGAs*. 2010. (Cited on pages 24, 34, 62, and 120.)

[37] de Dinechin, F., and Pasca, B. Flopoco-floating-point cores. (Cited on page 22.)

[38] De Dinechin, F., and Pasca, B. Floating-point exponential functions for DSP-enabled FPGAs. In *Field-Programmable Technology (FPT), 2010 International Conference on* (2010), IEEE, pp. 110–117. (Cited on page 34.)

[39] de Pimentel, J., and Tirat-Gefen, Y. Hardware acceleration for real time simulation of physiological systems. In *28th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2006. EMBS '06* (Aug. 2006), pp. 218–223. (Cited on pages 57 and 106.)

[40] DeHon, A. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% lut utilization). In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays* (1999), ACM, pp. 69–78. (Cited on page 107.)

[41] DeRose, S., Maler, E., Orchard, D., and Walsh, N. XML Linking Language (XLink). http://www.w3.org/TR/xlink/, 2010. (Cited on page 2.)

[42] Dick, G., and Ceriel, H. Parsing Techniques, a Practical Guide. Tech. rep., 1990. (Cited on pages 74 and 75.)

[43] Dijkstra, E. W. *ALGOL-60 translation*. Mathematisch Centrum, 1961. (Cited on page 75.)

[44] Fan, X., Weber, W.-D., and Barroso, L. A. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News* (2007), vol. 35, ACM, pp. 13–23. (Cited on page 1.)

[45] Fowers, J., Brown, G., Cooke, P., and Stitt, G. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays* (2012), ACM, pp. 47–56. (Cited on page 35.)

[46] Garny, A. *OpenCell*. 2010. (Cited on page 56.)

[47] Ghosh, A., Paul, S., and Bhunia, S. Energy-Efficient Application Mapping in FPGA through Computation in Embedded Memory Blocks. In *VLSI Design (VLSID), 2012 25th International Conference on* (2012), IEEE, pp. 424–429. (Cited on page 34.)

[48] Goth, G. Entering a parallel universe. *Communications of the ACM 52*, 9 (2009), 15–17. (Cited on page 1.)

[49] Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *16th International Conference on VLSI Design, 2003. Proceedings* (Jan. 2003), pp. 461–466. (Cited on pages 58 and 107.)

[50] Hilgemann, D. W., and Noble, D. Excitation-contraction coupling and extracellular calcium transients in rabbit atrium: reconstruction of basic cellular mechanisms. https://models.physiomeproject.org/exposure/49f298f54f3e916fca650c8e76d82e46. CellML Author: Catherine Lloyd. (Cited on page 7.)

[51] Hilgemann, D. W., and Noble, D. Excitation-contraction coupling and extracellular calcium transients in rabbit atrium: reconstruction of basic cellular mechanisms. *Proceedings of the Royal Society of London. Series B, Containing Papers of a Biological Character. Royal Society (Great Britain) 230*, 1259 (Mar. 1987), 163–205. (Cited on pages 6 and 89.)

[52] Hodgkin, A. L., and Huxley, A. F. A quantitative description of membrane current and its application to conductance and excitation in nerve. http://models.physiomeproject.org/exposure/

3b7252fdf6ff56356382f42d2824ecae. CellML Author: Catherine Lloyd. (Cited on page 5.)

[53] Hodgkin, A. L., and Huxley, A. F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology 117*, 4 (Aug. 1952), 500–544. (Cited on pages 4, 35, and 89.)

[54] Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., et al. The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models. *Bioinformatics 19*, 4 (2003), 524–531. (Cited on pages 56 and 106.)

[55] Intel Corporation. Intel Xeon Processor E5-4650 (20M Cache, 2.70 GHz, 8.00 GT/s Intel QPI). http://ark.intel.com/products/64622/Intel-Xeon-Processor-E5-4650-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI. (Cited on pages 91 and 134.)

[56] Intel Corporation. Optimizing Performance with Intel Advanced Vector Extensions. Tech. rep., September 2014. (Cited on pages 100 and 142.)

[57] Jones, D. H., Powell, A., Bouganis, C., and Cheung, P. Y. GPU versus FPGA for high productivity computing. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on* (2010), IEEE, pp. 119–124. (Cited on page 55.)

[58] Kapre, N., and DeHon, A. Accelerating the SPICE Circuit Simulator Using an FPGA: A Case Study. In *[High-Performance Computing Using FPGAs*. (Cited on page 34.)

[59] Kestur, S., Davis, J., and Williams, O. BLAS comparison on FPGA, CPU and GPU. In *2010 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (July 2010), pp. 288–293. (Cited on pages 35 and 57.)

[60] Langhammer, M., and VanCourt, T. FPGA floating point datapath compiler. In *Field Programmable Custom Computing Machines, 2009.*

FCCM'09. 17th IEEE Symposium on (2009), IEEE, pp. 259–262. (Cited on page 34.)

[61] Lattner, C., and Adve, V. LLVM language reference manual, 2006. (Cited on pages 112 and 113.)

[62] Liang, X., Vetter, J. S., Smith, M. C., and Bland, A. S. Balancing FPGA Resource Utilities. In *ERSA* (2005), pp. 156–162. (Cited on page 107.)

[63] Llamocca, D., Carranza, C., and Pattichis, M. Separable FIR filtering in FPGA and GPU implementations: Energy, performance, and accuracy considerations. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on* (2011), IEEE, pp. 363–368. (Cited on page 35.)

[64] Lloyd, C. M., Lawson, J. R., Hunter, P. J., and Nielsen, P. F. The CellML model repository. *Bioinformatics 24*, 18 (2008), 2122–2123. (Cited on page 4.)

[65] Mathworks. *MATLAB and Simulink*. 2014. (Cited on page 56.)

[66] Meredith, M. A look inside behavioral synthesis. *EEdesign. com* (2004), 04–08. (Cited on page 25.)

[67] Miller, A. K., Marsh, J., Reeve, A., Garny, A., Britten, R., Halstead, M., Cooper, J., Nickerson, D. P., and Nielsen, P. F. An overview of the CellML API and its implementation. *BMC bioinformatics 11*, 1 (2010), 178. (Cited on pages 9 and 11.)

[68] Moore, H. *MATLAB for Engineers*. Prentice Hall Press, 2014. (Cited on page 124.)

[69] Nallamuthu, A., Smith, M. C., Hampton, S., Agarwal, P. K., and Alam, S. R. Energy efficient biomolecular simulations with FPGA-based reconfigurable computing. In *Proceedings of the 7th ACM international conference on Computing frontiers* (2010), ACM, pp. 83–84. (Cited on page 34.)

[70] National instruments. *NI LabView*. 2014. (Cited on page 56.)

[71] Nickerson, D. P., Ladd, D., Hussan, J. R., Safaei, S., Suresh, V., Hunter, P. J., and Bradley, C. P. Using CellML with OpenCMISS to simulate multi-scale physiology. *Frontiers in bioengineering and biotechnology 2* (2014). (Cited on page 11.)

[72] NSR Physiome Project. *Mathematical markup language*. 2012. (Cited on pages 56 and 106.)

[73] NVIDIA UK. *Tesla C2050 / C2070 GPU Computing Processor*. 2011. (Cited on pages 91 and 134.)

[74] Oklobdzija, V. G., Villeger, D., and Liu, S. S. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *Computers, IEEE Transactions on 45*, 3 (1996), 294–306. (Cited on page 107.)

[75] Okuyama, T., Okita, M., Abe, T., Asai, Y., Kitano, H., Nomura, T., and Hagihara, K. Accelerating ODE-based simulation of general and heterogeneous biophysical models using a GPU. *IEEE Transactions on Parallel and Distributed Systems 25*, 8 (Aug. 2014), 1966–1975. (Cited on page 57.)

[76] Osana, Y., Yoshimi, M., Iwaoka, Y., Kojima, T., Nishikawa, Y., Funahashi, A., Hiroi, N., Shibata, Y., Iwanaga, N., Kitano, H., and Amano, H. ReCSiP: An FPGA-based general-purpose biochemical simulator. *Electronics and Communications in Japan (Part II: Electronics) 90*, 7 (July 2007), 1–10. (Cited on pages 56 and 106.)

[77] Ousterhout, J. K. *Tcl: An embeddable command language*. Citeseer, 1989. (Cited on page 85.)

[78] Petitet, A., Whaley, R. C., Dongarra, J., and Cleary, A. *High performance LINPACK*. 2012. (Cited on pages 35 and 58.)

[79] Pitt-Francis, J., Pathmanathan, P., Bernabeu, M. O., Bordas, R., Cooper, J., Fletcher, A. G., Mirams, G. R., Murray, P., Osborne, J. M., Walter, A., et al. Chaste: a test-driven approach to software develop-

ment for biological modelling. *Computer Physics Communications 180*, 12 (2009), 2452–2471. (Cited on pages 56 and 106.)

[80] Processor, D. Power and thermal management in the intel® core tm. *Intel® Centrino® Duo Mobile Technology 10*, 2 (2006), 109. (Cited on page 142.)

[81] Ravikesh Chandra. *Novel Approaches to Automatic Hardware Acceleration of High-Level Software*. 2013. (Cited on pages 58 and 107.)

[82] Reiter, T. Optimising code generation with haggies. *Computer Physics Communications 181*, 7 (2010), 1301–1331. (Cited on page 107.)

[83] Resasco, D. C., Gao, F., Morgan, F., Novak, I. L., Schaff, J. C., and Slepchenko, B. M. Virtual cell: computational tools for modeling in cell biology. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine 4*, 2 (2012), 129–140. (Cited on page 56.)

[84] Roesler, E., and Nelson, B. Novel optimizations for hardware floating-point units in a modern FPGA architecture. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Springer, 2002, pp. 637–646. (Cited on page 34.)

[85] Ronacher, A. *Jinja2 (The Python Template Engine)*. 2011. (Cited on pages 80 and 111.)

[86] Rush, S., and Larsen, H. A practical algorithm for solving dynamic membrane equations. *IEEE Transactions on Biomedical Engineering BME-25*, 4 (July 1978), 389–392. (Cited on page 97.)

[87] Sadoghi, M., Singh, H., and Jacobsen, H.-A. Towards highly parallel event processing through reconfigurable hardware. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware* (2011), ACM, pp. 27–32. (Cited on page 11.)

[88] Salcic, Z., Cao, J., and Nguang, S. K. A floating-point FPGA-based self-tuning regulator. *Industrial Electronics, IEEE Transactions on 53*, 2 (2006), 693–704. (Cited on page 34.)

[89] Schoenborn, Z. Board design guidelines for pci express architecture. In *PCI-SIG APAC Developers Conference* (2004). (Cited on pages 142 and 144.)

[90] Shubhranshu. Integrating SED-ML and CellML models on the GPU. Tech. rep., 2011. (Cited on pages 57, 91, 99, 102, 134, and 139.)

[91] Terasic Technologies. Altera DE4 development and education board. http://de4.terasic.com/, 2012. (Cited on pages 91 and 119.)

[92] Tessier, R., and Giza, H. Balancing logic utilization and area efficiency in FPGAs. In *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*. Springer, 2000, pp. 535–544. (Cited on page 107.)

[93] Thomas, D., and Amano, H. A fully pipelined FPGA architecture for stochastic simulation of chemical systems. In *2013 23rd International Conference on Field Programmable Logic and Applications (FPL)* (Sept. 2013), pp. 1–7. (Cited on pages 56 and 106.)

[94] TOP500. *TOP500 Supercomputer*. 2014. (Cited on pages 2, 55, and 105.)

[95] Tusscher, K. H. W. J. t., Noble, D., Noble, P. J., and Panfilov, A. V. A model for human ventricular tissue. https://models.physiomeproject.org/exposure/bf8e84daa606d751b2ada6355e2333c1. CellML Author: Catherine Lloyd. (Cited on page 9.)

[96] Tusscher, K. H. W. J. t., Noble, D., Noble, P. J., and Panfilov, A. V. A model for human ventricular tissue. *American Journal of Physiology - Heart and Circulatory Physiology 286*, 4 (Apr. 2004), H1573–H1589. (Cited on pages 7, 89, and 132.)

[97] Umbehr, J. *Supercomputer Creates Most Advanced Heart Model*. 2008. (Cited on page 57.)

[98] Wolfram. *Wolfram Mathematica: Definitive System for Modern Technical Computing*. 2014. (Cited on page 56.)

[99] Yoshimi, M., Osana, Y., Fukushima, T., and Amano, H. Stochastic simulation for biochemical reactions on FPGA. In *Field Programmable Logic*

and Application, J. Becker, M. Platzner, and S. Vernalde, Eds., no. 3203 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2004, pp. 105–114. (Cited on pages 56 and 106.)

[100] Yu, T., Bradley, C., and Sinnen, O. Hardware acceleration of biomedical models with OpenCMISS and CellML. In *Field-Programmable Technology (FPT), 2013 International Conference on* (2013), IEEE, pp. 370–373. (Cited on pages 29 and 31.)