TECHNIQUES FOR FPGA NEURAL MODELING

A Thesis
Presented to
The Academic Faculty

by

Randall K. Weinstein

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Interdisciplinary Bioengineering Program, School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2006

TECHNIQUES FOR FPGA NEURAL MODELING

Approved by:

Professor Robert H. Lee,
Committee Chair
Biomedical Engineering
Emory University

Professor Robert H. Lee, Adviser
Biomedical Engineering
Emory University

Professor Robert J. Butera
Electrical and Computer Engineering
Georgia Institute of Technology

Professor Steven P. DeWeerth
Biomedical Engineering
Georgia Institute of Technology

Professor Vijay K. Madisetti
Electrical and Computer Engineering
Georgia Institute of Technology

Professor Eberhard O. Voit
Biomedical Engineering
Georgia Institute of Technology

Date Approved: December 2006

ACKNOWLEDGEMENTS

These past four years have been an incredibly rewarding experience for me and I am grateful to everyone who has helped and supported me throughout this process. I would like to first thank Dr. Allen Tannenbaum, my undergraduate research advisor. and Dr. Ming-Ju Ho, my colleague at Agere Systems, both of whom inspired me towards higher education.

I am thankful to Dr. Robert (Bob) Lee, my advisor, for his unwavering support for my research since he first offered me a position in his laboratory. For never letting me fail, even if I wanted to give up, and always having the confidence in me that I can achieve anything I work hard towards, I am forever grateful.

I am very thankful to my Ph.D. thesis committee for their insistence on quality and assisting me to produce a thesis that I can be proud of. I am particularly thankful to Dr. Stephen DeWeerth for continuously providing helpful and astute advice, without overreaching pressure. I want to thank Dr. Robert Butera for his leadership of the Bioengineering program and his helpful comments and advice for my thesis. To Dr. Eberhard Voit, I am grateful for ensuring a strong mathematical basis to my thesis and for your detailed comments that substantially improved this document. Finally, to Dr. Vijay Madisetti, I am thankful for you and your class which provided the initial inspiration for the hardware generation in Dynamo.

I am honored and thankful to have worked with both Sarah Jones and Nick Shapiro since the very beginning. You are both great researches, great colleagues, and great friends to me. I am looking forward to celebrating both of your graduations very soon.

I am thankful to Chris Church for his constant advice, occasional criticism, frequent coffee breaks, and for being a great sounding board for new ideas. I am looking forward to our future work together on Dynamo. I would like to thank the additional people who have worked on Dynamo, including Steve Feng, Jamie Meyers, Melissa Freedenberg, and Bejean Mosher. Dynamo is a true collaborative effort and I thank you all for your hard

work and effort towards making this tool a reality. I also want to thank and acknowledge Dr. Michael Sorensen for his helpful counsel during the last few months of this process and I anxiously look forward to our continued future work.

I am thankful to all the other past members of the Lee Group, Dr. Autumn Schumacher, William Boatin, Estelle Graas, and Phillipe Karam. Autumn, thank you in particular, for your always helpful insight and wisdom, and for being the lab nurse when weve been down.

I would like to acknowledge and thank the additional members of our group, Cassie Mitchell, Brock Wester, Matt Sowd, and Jamie Lazin. I hope we continue our all-you-can-eat sushi trips for a long time.

I am thankful to those who have helped expand my research into wet electrophysiology, including my former REU student and now graduate student, JoAnna Todd. I am particularly thankful to Dr. Kacy Cullen for his support during our collaborative efforts.

I am thankful to Dr. Michelle LaPlacas group, including Sarah, Chris, Kacy, Gustavo, Ciara, Hillary, Crystal, and Varad for always making me feel like a part of your lab group, even through I never figured out how to mix a solution.

I would like to thank Richard Blum for not only introducing me to the laboratory, but also for being my brother through this process. I am forever grateful for all of your friendship since freshman year together at Georgia Tech.

I would like to thank and acknowledge Kate Williams and Maxine McClain who would always be there for me whenever I needed sympathy and or just needed to complain.

To my parents, thank you for your unending confidence and your unconditionally love, for without that, I would have never been able to reach this milestone. To my sister Michelle, thank you for your constant support and of course for throwing all of those great parties.

Et je suis reconnaissant envers ma copine, Aurélie, pour ton encouragement et ta gentilesse. Tu mas aidé à atteindre mes buts. Merci beaucoup, mon amour.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Neural simulations and general dynamical system modeling consistently push the limits of available computational horsepower. This is occurring for a number of reasons: 1) models are progressing in complexity as our biological understanding increases, 2) high-level analysis tools including parameter searches and sensitivity analyses are becoming more prevalent, and 3) computational models are increasingly utilized alongside with biological preparations in a dynamic clamp configuration. General-purpose computers, as the primary target for modeling problems, are the simplest platform to implement models due to the rich variety of available tools. However, computers, limited by their generality, perform sub-optimally relative to custom hardware solutions. The goal of this thesis is to develop a new cost-effective and easy-to-use platform delivering orders of magnitude improvement in throughput over personal computers.

We suggest that FPGAs, or field programmable gate arrays, provide an outlet for dramatically enhanced performance. FPGAs are high-speed, reconfigurable devices that can implement any digital logic operation using an array of parallel computing elements. Already common in fields such as signal processing, radar, medical imaging, and consumer electronics, FPGAs have yet to gain traction in neural modeling due to their steep learning curve and lack of sufficient tools despite their high-performance capability. The overall objective of this work has been to overcome the shortfalls of FPGAs to enable adoption of FPGAs within the neural modeling community.

We embarked on an incremental process to develop an FPGA-based modeling environment. We first developed a prototype multi-compartment motoneuron model using a standard digital-design methodology. FPGAs at this point were shown to exceed software simulations by 10× to 100×. Next, we developed canonical modeling methodologies for manual generation of typical neural model topologies. We then developed a series of tools

and techniques for analog interfacing, digital protocol processing, and real-time model tuning. This thesis culminates with the development of DYNAMO, a fully-automated model compiler for the direct conversion of a model description into an FPGA implementation.

DYNAMO includes a fully-custom programming language for describing model mechanisms and construction. After processing through a minimal intermediate representation, the model equations undergo a variety of algebraic optimizations and reductions, floating-point to fixed-point conversion, timing analysis, and hardware scheduling. The DYNAMO compiler provides the core technology for a fully-integrated FPGA-based neural modeling environment.

CHAPTER I

INTRODUCTION

Our collective understanding of physiological mechanisms has been growing while our system-level understanding is still in its infancy. Improved experimental techniques have enabled experimentation with an enhanced degree-of-precision, targeting individual proteins and cellular mechanisms. At a higher-level, physiologists often study complex systems through construction of phenomenological models. A new challenge is emerging—to combine our experimentally-determined sub-cellular insight and our phenomena-based explanations of physiologic behavior into a unified modeling paradigm.

This objective has numerous challenges. First, our understanding is never complete. Models will continuously be tuned and enhanced as necessary to encompass future discoveries. Next, models are often under-specified as known behaviors are often depicted across multiple parameter sets [33, 64]. Finally, from an engineering perspective, the performance achieved by simulation technology has consistently lagged behind simulation demands. Historically, personal computers have been the standard means for model simulation spurred by ever increasing processing power and low costs. For some example applications, costly IBM (International Business Machines Corporation) supercomputers [2, 62] and computer clusters [56, 68, 30] have provided a significant performance advantage to modelers. With the exception of these few applications, high-speed modeling tools have not been able to fully penetrate the market.

Our particular question of interest involves the study of how neurons work. We study lumbar α-motoneurons due to their large size, accessibility in animal preparations, and extensive dendritic arborization. The adult-cat model, utilized for over a hundred years, heralded discoveries including the neuron, proprioceptive feedback pathways, afferent/efferent distribution in the spinal column, and *in vivo* intra-cellular recordings, among other innovations [54, 74, 75, 14]. Within the last few decades, our understanding of motoneuron

function has transformed from that of a passive integrator of synaptic input to a non-linear, complex system including persistent inward currents [71], bistability [41], slow amplification [53], and most recently amplification of fast Ia-dominated synaptic input [45].

Our laboratory has explored the behavior of motoneurons through modeling studies (Shapiro and Lee, unpublished results) and continued experimental work. The model is a multi-conductance, multicompartment topology whose implementation exceeds 200 differential equations. One particular 10-compartment model simulated one second of neural activity in three seconds using a custom-designed software simulator. While simulations across multiple protocols and various parameter sets can be time consuming, they are still tractable. For a different question—for example, the role of single-neuron emergent behavior on force production in a motor pool—the model would have to be drastically expanded. A typical cat motoneuron medial-gastrocnemius (MG) motor-pool has approximately 280 motor units [17]. Not including a muscle model, a simulation of all motor units will increase the processing time to 14 minutes per second. A typical voltage-clamp ramp protocol simulates 11 seconds of time, requiring over 2.5 hours per simulation. Excessive simulation times reduce the practicality of modeling studies.

The role of observed single-cell phenomena in a larger physiologic system motivated research into high-throughput modeling alternatives. The Graas *et al.* comparison of personal computers, digital signal processors (DSPs), and field-programmable gate arrays (FPGAs) exposed a wide performance-gap between FPGAs and other low-cost simulation technologies [36, 37]. FPGAs, as reprogrammable devices capable of arbitrary computation using an array of parallel digital logic primitives, are shown to provide performance improvements up to 75× over general-purpose computers. This study confirmed the performance advantage of FPGAs, but highlighted the difficulty in utilizing these devices for neural modeling.

Neural modelers use a variety of platforms for their simulations. Each simulation platform has clear trade-offs (see Figure 1 for a comparison). General-purpose computers, the most common neural modeling platform, has a wide variety of software modeling tools written for it and has been validated as a capable simulation platform in thousands of publications. It has never been the highest performing simulation platform as that honor

	Speed	Rate	Size	Ease	Cost	Accuracy
Computer	✘	✘	-	✓✓	✓✓	✓
Cluster	✓✓	✘✘	✓	-	✘	✓
Supercomputer	✓✓	-	✓✓	✘	✘✘	✓
Analog Circuit	✓✓	✓✓	-	✘✘	✘✘	✘
FPGA	✓✓	✓✓	✘✘	✘✘	✓	-

Figure 1: This platform comparison table highlights the trade-offs and limitiations inherent in each possible simulation platform. Here, *speed* refers to overall simulation throughput, *rate* is a measure of the latency of the simulation platform, *size* refers to limits in the overall complexity either based on the computation or memory limitations, *ease* combines multiple factors including the available of modeling tools and required custom programming, *cost* refers to the full system cost including design revisions, software, etc., and *accuracy* is a measure of numerical accuracy.

has generally been reserved for computer clusters and supercomputers. Computer clusters and supercomputers typically require custom software solutions and expensive hardware to make neural modeling viable. Therefore, the ease of developing a model on a standard computer is generally preferred over the raw performance found on other platforms.

Some researchers have proposed the use of analog VLSI devices to simulate neurons. Implementing neurons directly in an analog circuit on a silicon device can enable very high performance and would be ideal for neural interfacing applications. However, these devices are exceedingly difficult to develop even for analog design engineers. As a result, many costly revisions are often required until the device performs as expected.

FPGAs share the high performance characteristics of computer clusters, supercomputers, and analog circuits along with the interfacing capabilities of analog VLSI. FPGAs disadvantages are three fold. First, limited resources (memory and parallel algebraic operations) available on an FPGA make large model development difficult. Second, FPGAs are difficult to program and require special training in digital design. As a result, FPGAs historically have been exclusively within the domain of electrical engineers for vertical market applications including radar, medical imaging, military equipment, and cellular base

stations. Finally, FPGAs perform optimally when calculating using a fixed-point number system over a floating-point number system. Floating-point is a mainstay of other computer platforms and provides very high relative accuracy even when numbers change by orders of magnitude. Fixed-point, on the other hand, provides high absolute accuracy and is not well suited for changes in values over orders of magnitude. This is a concern since neural modelers have naturally come to expect numerically accurate simulations. We initiated research into neural model development on FPGAs to overcome these accuracy concerns as well as the size and ease-of-use concerns raised in Figure 1.

This thesis began with an initial high-performance neural models manually designed based on the "art" of digital system design. This effort culminated in an example ten-compartment motoneuron model that is described in Chapter 3. The development of this model exposed the need for reusable and canonical model constructions. Research was refocused around simple models to explore various methods of building small and fast models as well as populations of models (see Chapter 4). This research provided a basis for a prototype set of tools to aid in model construction. As part of a collaboration with Michael S. Reid, we built a 40-neuron population from a simplified Pre-Bötzinger Complex model. This model included 1600 synapses interconnecting each neuron. Model construction, simulation, and interaction via this assisted design flow is elaborated in Chapter 5.

In parallel to these efforts, work was underway to build a fully-automated environment from neural model construction to simulation. We dubbed this research and development effort DYNAMO, and the result is a software compiler incorporating lessons learned from the manual design of neural models. The DYNAMO compiler required considerable research into generalizing the construction of models, as inputs into this system were not guaranteed to take on any particular structure. Assumptions made in our manual model development trials were revisited, often requiring novel solutions. A complete look into the research and results of the DYNAMO compiler can be found in Chapter 6. This overall design progression is shown in Figure 2.

The overriding objective of this research is to enable FPGA model development as a viable and accessible platform for the neural modeling community. This thesis provides a

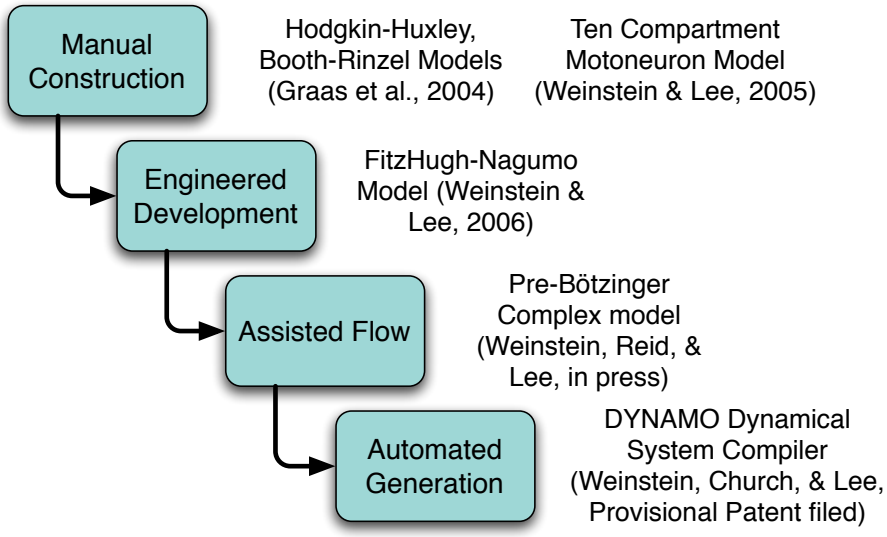


Figure 2: This illustration depicts the four phases of this thesis. The first phase was the manual art or manual construction phase where models were built using typical digital design principals. The second phase utilized an engineered approach specific to neural models. The third and fourth phases introduced an automated flow for model development on FPGAs.

guide for the manual implementation of neural models on FPGAs as well as the description and results of our DYNAMO compiler. We have successfully shown that our developed tools incorporating our novel algorithms provide neuroscientists with a powerful environment for realizing their own neural models on FPGAs with significant performance gains.

CHAPTER II

BACKGROUND AND SIGNIFICANCE

2.1 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are specialized integrated circuits that made up of reconfigurable computational primitives capable of implementing arbitrary calculations and logic. They are widely used in consumer and industrial products for accelerating processor intensive algorithms. Engineers designing networking, radar, wireless communications, video processing, aerospace, military, and test equipment, medical imaging, and other computationally intensive applications often utilize FPGAs as hardware co-processors [63, 92], DSP processors [46, 86], or stream processors [47, 50].

FPGAs have been less commonly used in bio-related fields, with several exceptions. Protein [59] and DNA [15] sequencing are starting to use FPGAs to reduce processing time. Real-time processing, registration and other image analyses from confocal microscopy are enabled by FPGAs [16, 66]. Most modeling applications on FPGAs have been limited to studying neural networks consisting of reduced neurons [7, 11, 21, 60].

2.2 Neural Modeling Technologies

Advances in both analytical tools and desired model complexity have pushed the computational requirements of simulations past the realm of personal computers. Large scale models can execute as slow as 1 simulation day per 1 second model time. Analytical tools, such as parameter searches, might require thousands of iterations before converging on a solution. The scope of neural modeling should not be held hostage to current limits to computational efficiency.

FPGAs are both well performing and reconfigurable. The performance can exceed both computers and digital signal processors (DSPs) while being reconfigurable within the scale of hours. This is contrasted with analog VLSI (very large scale integration) circuits which are high performance but require custom manufacturing to produce or modify (see Figure 3).

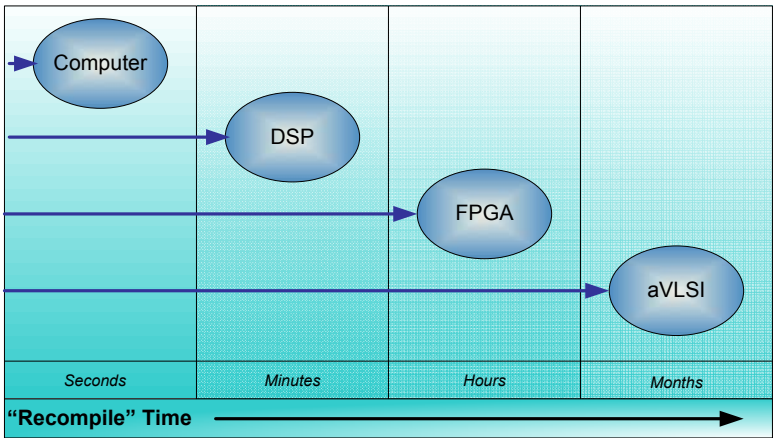


Figure 3: Depiction of the relative reconfigurability of the primary platforms for neural modeling. This graph illustrates the increasing time for reconfiguration beginning with adjusting a computer simulation to redeveloping a custom analog circuit.

The following subsections detail the different aspects of performance-sensitive neural modeling. Existing high performance neural applications are described and compared to FPGA implementations. An overview of existing neural simulation platforms and a survey of existing methodologies for FPGA development is provided. By themselves, none of the platforms or methodologies described are sufficient to provide high performing, reconfigurable neural modeling solutions that are accessible to the greater neuroscience community.

2.2.1 High Performance Neural Modeling

Other performance-critical applications utilize clusters [91, 8] or supercomputers [62] for their high complexity modeling efforts. IBM and the EPFL in Switzerland have partnered to simulate a neocortical column, thought to be the smallest neural unit to exhibit complex functionality. Some 10,000 neurons make up this model with 10M to 100M synapses. The IBM supercomputer is a BlueGene/L with estimated peak performance of 22.8 teraflops (IBM/EPFL Press Release), scalable to 360 teraflops, the performance of the fastest BlueGene/L machine and supercomputer in the world (www.top500.org).

These custom designed supercomputers, while beneficial to those laboratories that have access to them, are not generally available to the broader neuroscience community due to their high cost. It does illustrate the need for high performance computing solutions that not only set performance records, but also are highly accessible and cost effective.

Graas, in her seminal FPGA work [37], found that by utilizing off-the-shelf FPGA hardware, the Xilinx Virtex-II Evaluation Kit (Avnet Design Services), she was able to emulate Hodgkin–Huxley neural models [40] at up to 16× the rate of her desktop computer. Graas found a 72× improvement in performance for a motoneuron model developed by Booth & Rinzel [9]. The development board, currently available from Avnet for $279 (US$, less commercial software costs), showed that order of magnitude performance gains can be realized for low costs.

While FPGAs have rarely been used for cellular level neural models, there have been many cases where neural networks have utilized programmable hardware. Examples of implementations of single neurons have been suggested [23, 24, 60], which are generally of an integrate and fire nature with sigmoidal activation. A compiler that can target VHDL, a hardware description language, has been implemented for these simple models [29]. Snider, *et al.*, considered more complex neural models that include basic ionic conductances with a threshold function for spiking for the purpose of generating a neural model of the cricket circi [80]. Networks of these simple neurons have been shown for artificial neural networks [84, 11, 21, 67] and specifically for image processing applications [86].

Physiologically relevant models have found a place in analog VLSI, (very large scale integration), or custom analog integrated circuits. One example is a motoneuron model that features a configurable recruitment threshold and a current modulated firing rate [13]. Extensions to this work have been offered, but not beyond the complexity of a basic Hodgkin–Huxley [40] type model [31]. Other work has been done to implement reconfigurable neural systems in an aVLSI (analog VLSI) at a higher level of granularity [34] whereby the user can utilize the model without the domain knowledge of the circuitry.

Still, little work has been done to date in utilizing FPGAs for studying cellular level systems. Current neural models rarely have the structure inherent in artificial neural networks (ANN) or other idealized models. These models can contain ionic conductances based on Hodgkin–Huxley [40] type kinetics or Markov models (as an example, Kuo and Bean's Na^+ model [48]). Calcium, a second messenger implicated in a host of cellular processes and ligand-gated ion channels, is often included in physiological relevant models, but has not

been implemented as of yet in an FPGA or in an aVLSI device. The scope of this work targets these and other arbitrary mechanisms for FPGA modeling efforts.

2.2.2 Neural Simulation Tools

The neuroscience community has a variety of effective modeling tools at their disposal. While some modelers implement models directly in a traditional programming language, such as C or *Delphi* (Borland Software Corporation, Cupertino, CA), others utilize traditional modeling environments (Matlab/Simulink, Mathworks; LabView, National Instruments; Mathematica, Wolfram Research, Inc.) or dedicated neural simulation platforms (NEURON [39], GENESIS [12], or NCS (NeoCortical Simulator) [91]).

These modeling tools operate on varying levels of abstraction and require different sets of domain knowledge to utilize. The traditional programming languages provide maximal flexibility and the potential for optimal performance. These come at the expense of increased development time as these environments often lack library routines desired by modelers (ODE solvers, data structures, etc.). High level neural modeling tools provide a rich set of resources and model construction tools that can allow for rapid model generation. Often, some flexibility is lost, but scripting tools built-in to these modeling environments can compensate. Other tools try to extract the benefits of both, such as a neuromusculoskeletal translation package to convert a model from the construction tool to Simulink for simulation [25]. The *Dynamo* compiler is more similar to a traditional modeling environment in providing the library functions to perform modeling, but not encompassing the neuroscience domain knowledge directly.

The above mentioned tools in the base forms are limited to general purpose computer architectures, with the exception of NCS, which is targeted towards cluster computing. The *Dynamo* compiler, with its direct-to-FPGA capability, complements these existing tools by targeting modeling environments with its Matlab and C back-end and dedicated hardware with its FPGA back-end. Current collaborations with Michael Hines of the NEURON group will allow for integration of *Dynamo* with the NEURON environment to allow a NEURON-to-FPGA flow.

2.2.3 FPGA Compilers

Since the advent of FPGAs and high density digital ASICs (Application Specific Integrated Circuits), hardware engineers have looked for higher level representations for their designs. Early microprocessor designs, such as the Intel 4004, were implemented in layout at the transistor level. Following the development of CMOS processes, mini-cells, or standard cell libraries, emerged, allowing design of basic building blocks, boolean logic, latches, and flip-flops. This "gate" level design methodology quickly became cumbersome as transistor counts accelerated. Synthesis tools such as Design Compiler (Synopsys, Inc.) simplified digital development by supporting RTL, or register transfer logic, an approach to describing synchronous logic in VHDL or Verilog. Within the digital design community, there have been various attempts to create higher level languages, such as SystemC or SystemVerilog, with limited success. These methodologies to design take a bottom-up approach, whereby the target device guides the design decisions at the higher levels.

In parallel, there has been much work in adapting either sequentially executed software targeting general purpose computers or algorithm descriptions in Matlab to parallel hardware. Professor Prithviraj Banerjee at Northwestern University has made impressive seminal progress in this area for over a decade. Beginning with original work with the PARADIGM tool translating sequential Fortran code to run in distributed memory computers in the early to mid 1990's [4], Banerjee refocused his efforts towards heterogeneous distributed systems of FPGAs, DSP processors, and general purpose processors with a Matlab compiler dubbed MATCH [6]. MATCH, later adapted to SoC (System-on-a-Chip) with the PACT compiler [44], incorporated SYMPHANY, a scheduling algorithm capable of exploiting parallelism at various levels of granularity [73]. SYMPHANY scheduled at the device resource level, using a lazy scheduling algorithm and linear programming to optimize the allocation of code to DSP, FPGA, and other processing units. As modern day FPGAs take on characteristics of DSPs and include processors, such as a PowerPC in the Xilinx Virtex4 or Virtex-II Pro architectures, the concepts behind the SYMPHANY scheduler are applicable to technology today.

Co-founded by Banerjee, AccelChip (Milpitas, CA) provides a commercially available

Matlab to FPGA design flow [38, 5]. These tools overcome traditional hurdles by providing, for example, utilities to help convert floating point to fixed point numerics [3]. The target audience for these tools remains DSP algorithm designers with a strong domain knowledge of hardware design. The tools require meta-commands and an incremental design flow to guide the compiler towards a desired outcome, making it unsuited for physiologists and modelers who would rather be shielded from the compilation process. The FREEDOM compiler [95], marketed by another company co-founded by Banerjee, BinaChip (Glenview, IL), aims to avoid interactive flow via a direct translation of DSP binaries. The performance to date has been limited to 5× improvement. Cardoso and Neto have similarly developed a byte code to FPGA compiler [20].

Other commercial vendors have produced custom language extension to C, enabling a modified C-to-FPGA design flow. Mitrion C (Mitrionics AB, Sweden) and its customizable, highly parallel virtual soft-core processor are geared towards the supercomputing market. The Handel C language (Celoxica, Oxfordshire, UK) uses additional language constructs to explicitly mark parallelism in the code. Celoxica's product has made inroads into the neural network community [67, 61]. Impulse C (Impulse Accelerated Technologies, Kirkland, Washington) requires no additional custom language extension for FPGA translation, though the output is highly dependent on the specific coding style and therefore requires optimal code for best results [83]. The XPRES Compiler (Tensilica, Inc., Santa Clara, CA) approaches the problem uniquely by generating a custom processor/instruction set architecture (ISA) based on sample C/C++ code. The resulting processor comes equipped with a C compiler and debugger.

These commercial offerings are all capable of implementing the neural models of interest; however, each commercial offering requires domain knowledge in FPGA development, digital design, numerical differential equations solvers, etc. There is yet to be a product that focuses the design flow towards a modeling paradigm, enabling neurobiologists or neurophysiologists to work within their own domains of knowledge.

CHAPTER III

MANUAL ART MODEL CONSTRUCTION

Preface:

As the first step towards realizing neural models on FPGAs, we embarked on a test case to develop a multi-compartment model using a standard digital design methodology. We termed this as the manual art model construction approach as there were no documented techniques for implementing neural models on an FPGA. Where existing digital design practices were limited, an intuitive design approach was adopted. Our overall objective was to use this test case to garner greater insight into performance bottlenecks and potential difficulties in implementation.

We had three specific goals for this study. First, we wanted to implement the first multi-compartment, multi-conductance neuron model on an FPGA. Previous work has been limited to neural models with few conductances that readily fit within the constraints of the FPGA [37]. Second, we intended to demonstrate the execution of the neural model at real-time. Finally, we wanted to investigate and assess the overall performance capability of the FPGA for neural models.

We successfully implemented a ten-compartment and seven-conductance motoneuron model on the FPGA. When executing on an FPGA, we were able to achieve a 33 MHz clock frequency. With a 40 cycle sample period (40 clock cycles for one time-step of the simulation) and a 5 μs time step, the model executed at greater than 3$\times$ real-time. Additionally, through adjustment of the clock frequency and time-step, we altered the model to run at exactly real-time. We were then able to use an experiment-based data-acquisition system to stimulate and record from the model motoneuron.

The performance of the model was less than expected. There were two contributors to the reduced performance. First, the critical timing path in the design limited the clock

frequency to 33 MHz. It was expected that the fastest clock, driving only embedded multipliers, would achieve a clock frequency of greater than 100 MHz. Second, a 40 cycle sample period was deemed excessive for a ten-compartment, pipelined model implementation, given the low clock frequency.

This manual art approach to model construction was successful in terms of implementing a faster than real-time multi-compartment, multi-conductance neural model. However, we found that this approach was very time consuming and error prone. Timing, or synchronization, errors were common and often very difficult to track down. Precision determination, or the process in which a model is converted from a floating-point representation to a fixed-point representation, was done by trial and error, requiring many iterations before reaching a desired numerical accuracy. We additionally found that any significant modifications to the design required considerable development effort and would often create additional errors in the design. This test case inspired the work in the next chapter, the design of the first general neural-modeling framework.

3.1 Introduction

Neural model complexity has steadily increased as more underlying processes are discovered and evaluated. These complex models require reconfigurable, high-speed simulations to tune and evaluate model characteristics. Based on original work in this laboratory[37], we have utilized FPGAs to generate a physiologically based implementation of a medial gastrocnemius motoneuron. This state-of-the-art model consists of somatic and initial segment compartments coupled to eight dendritic compartments. The motoneuron representation/-computer model includes voltage sensitive ionic conductances enabling sodium, potassium, and calcium currents. A Xilinx Virtex-II XC2V3000 FPGA provides a simulation platform capable of greater than 4 times real-time performance.

3.2 Methods

3.2.1 Motoneuron Model

The motoneuron model implemented was based on a custom neural software simulator. The motoneuron is divided into 10 compartments consisting of a soma, an initial segment, and a single 8-part dendrite oriented proximal to distal (see Figure 4). Spatial separation of dendrites is a necessary requirement for the voltage gradients, or plateau potentials, that underlie many active dendrite phenomena. A highly excitable, tightly coupled initial segment serves as the origin of somatic spiking.

Various ionic conductances emulating sodium, potassium, and calcium channels were distributed throughout the 10 compartment model. Sodium channels were implemented as 12-state Markov models with voltage-gated activation and non-voltage dependant inactivation. These channels, as described by Kuo and Bean[48], required deactivation, or a repolarization below activation threshold, before complete inactivation. These channels, at depolarized potentials, provide persistent current, a key requirement in generating plateau potentials[71, 51].

Two potassium channels were implemented with sigmoidal activation functions and a voltage-sensitive time constants. A faster delayed-rectifier channel provided for a fast after-hyperpolarization (fAHP) while a second slower activating conductance limited firing rate

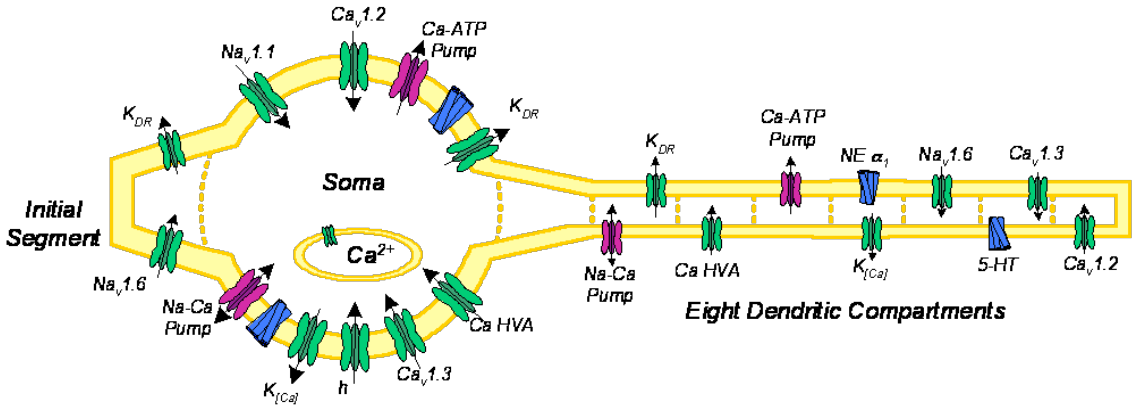


Figure 4: Cartoon of the 10 compartment motoneuron model depicting a sampling of channels and receptors. Channels depicted in the dendrites can exist in all compartments.

following successive spikes. An h-channel provided inward current at hyperpolarized potentials limiting AHP depth. Two calcium channels with sigmoidal activation functions and voltage-independent time-constants were included for completeness but had little functional role in the model.

3.2.2 Design Flow

Redevelopment from a software-based model to an FPGA required a conversion from sequentially executed program instructions to parallel executed hardware modules. On an FPGA, these hardware modules additionally require manual manipulation to ensure synchronous behavior. Additionally, all calculations were translated from floating-point types to fixed-point, an integer-based number system, for efficient processing on the FPGA. We chose to implement the model using Xilinx System Generator v6.2, an add-on toolkit for the Simulink modeling environment within Mathworks MATLAB R14. The toolkit additionally enabled co-simulation, or the concurrent simulation of a portion of the model in the FPGA hardware and the remainder simulating in software.

The resulting design was targeted towards a Xilinx XtremeDSP Development Kit-II based on a Nallatech BenONE carrier board consisting of one DIME-II module slot and populated with a Nallatech BenADDA expansion board. The BenADDA is preconfigured with a Xilinx Virtex-II XC2V3000 FPGA, 2 14-bit 65 MS/s analog to digital converters (ADC), 2 14-bit 160 MS/s digital to analog converters (DAC) and one megabyte of on

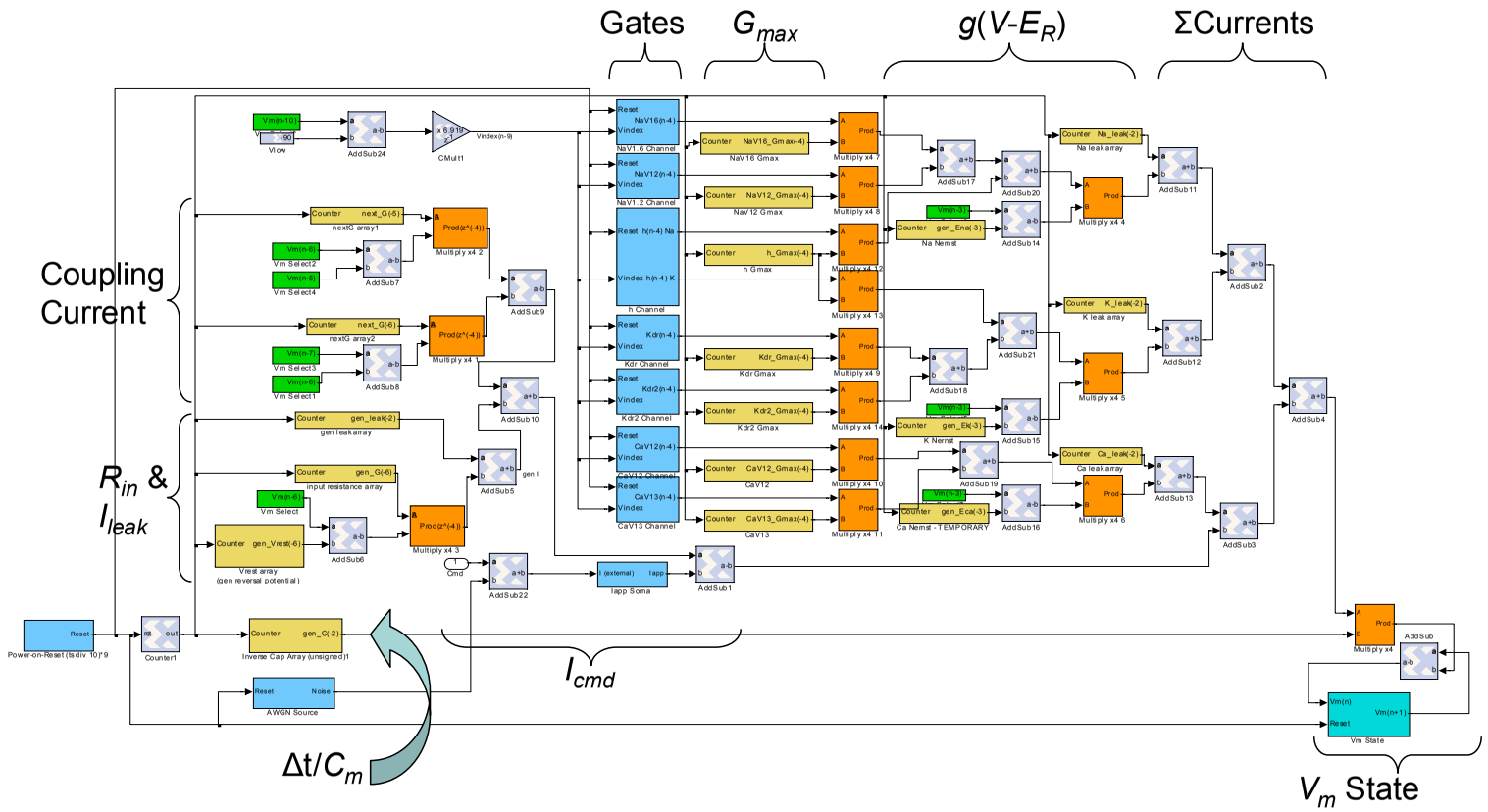


Figure 5: Top-level view of the model. Xilinx blocks are colored light blue with a white X logo, subcomponents (channels, glue logic, etc.) are colored blue, state variables are colored cyan, parameter and LUTs (lookup tables) are colored yellow, signal demultiplexors are colored green and multipliers are colored orange.

board SRAM. Using a function generator, somatic current input was applied via an ADC channel to perform simulations. Voltage potentials were recorded through the on-board DACs using standard data acquisition hardware (16-bit, 100 kS/s).

3.3 Hardware Design

The motoneuron model was developed as a fully synchronous design consisting of a 10-stage pipeline corresponding to the number of simulated compartments. Each channel was implemented once in hardware and executed 10 times (once per compartment) per simulation time step as shown in Figure 5. When a channel was unused in a compartment, conductance was set to zero. All state variables were computed using Euler integration and executed synchronously.

All combinational logic paths were constrained to run within 10 cycles. Certain blocks, such as the multiplier required 4 cycles to execute. To reduce the impact on the logic pipeline, multiplication operations involving two variable operands were implemented as hardware multipliers using a multicycle approach, running at 4 times the pipeline clock rate for 4 cycles, resulting in one pipeline delay per use. As a result, the fastest clock in the system was 40 times faster than the simulation timestep, fanning out only to hardware multiplier units.

Fixed-point precision was empirically chosen to minimize hardware usage while maintaining numerical integrity. Upper and lower bounds for quantities were applied when determining the number of integer bits avoiding overflow conditions.

All parameters were set static in the simulation. When a parameter was constant, *e.g.* channel constants, across all compartments, the value was folded into the next arithmetic block. Most parameters were varied per compartment and were stored in circularly referenced 10 deep read-only memory (ROM) blocks. The ROM was initialized as to provide the correct delay in the parameter corresponding to the insertion stage of the pipeline, mitigating the need for additional synchronizing registers providing delay. At each cycle, the parameter value corresponding to the appropriate compartment was produced on the output of the circular ROM.

Certain costly functions, such as sigmoids and exponentials, were implemented as look-up tables (LUTs). All LUTs were voltage dependent and had an addressability of 11-bits.

3.4 Results

The synthesized design resulted in a Xilinx equivalent gate count of 2.6M. 39% of flip-flops, or registers, were utilized while 40% of the base combination logic elements, or 4 input LUTs, were employed. The model used all 96 hardware multiplier resources available in the Virtex-II FPGA. Roughly 1/3 of the block RAMS were used and configured as function LUTs or parameter ROMs.

The critical path, or the longest single cycle propagation time, was found to be 30 ns, implying a maximum 33 MHz clock. Within 40 cycles, 10 compartments were computed, enabling a computation rate of 8.25M compartments per second. With the time step fixed at 5 s, over 4× real-time performance was achieved.

For the purposes of testing, the model was slowed down to real-time by decreasing the time step to 2 s and reducing the clock rate to 20 MHz. Additionally, a noise source consisting of additive Gaussian noise ($<$ 10 kHz) was added to the model to emulated realistic physiological recording conditions. A five second ascending and descending current ramp was applied to the input while the soma and most distal dendrite potential were recorded (see Figure 6). We found the model produced action potentials at an increasing frequency as the input current increased. Additionally, the model exhibited hysteresis, a result of the activation of dendritic originating persistent inward currents, as found *in vivo*[52].

3.5 Conclusion

The resulting model output is not yet fully characteristic of motoneuron output. Many deficiencies can be remedied by tuning the parameters and perhaps altering mechanisms modeled, neither of which is constrained by the implementation in hardware. Despite the discrepancies in output, the model demonstrates a level of neural complexity not previously modeled in FPGAs. The hardware demonstrates a high degree of flexibility and can readily be adapted to any reasonable evolution in the motoneuron model.

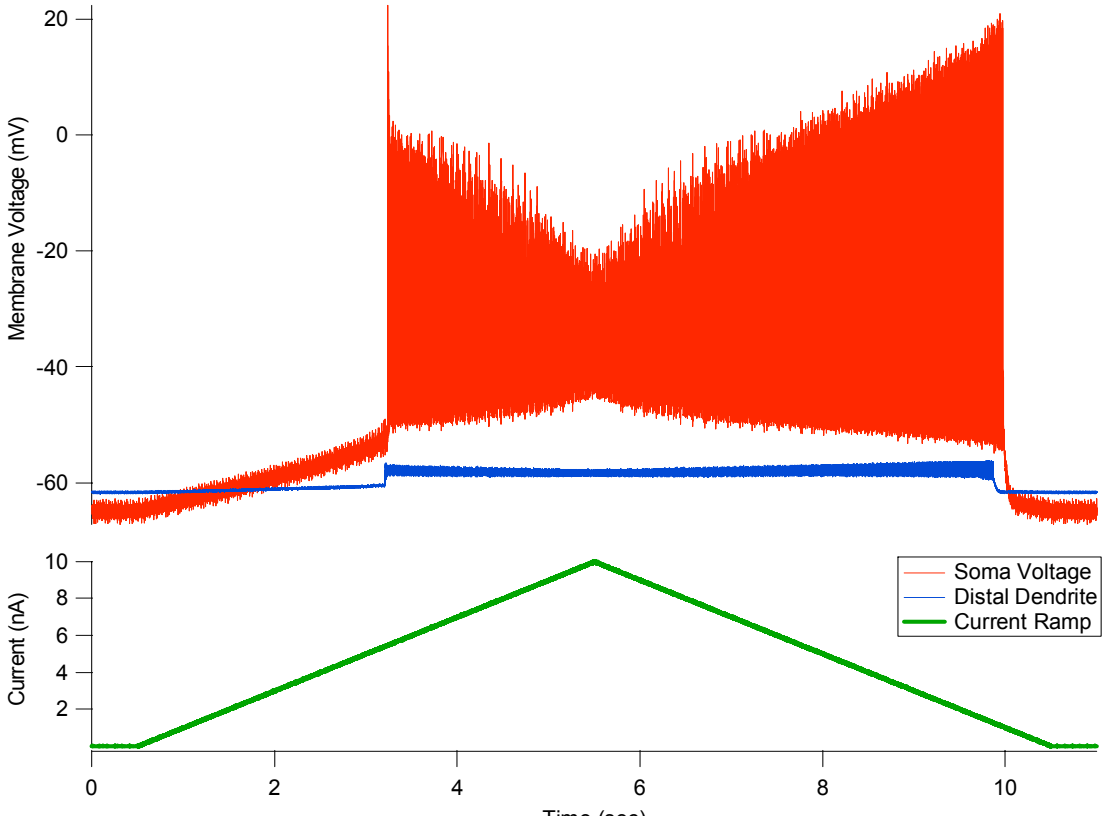


Figure 6: Real-time capture of output from FPGA hardware with a generated current input (5 s up/down ramp). Voltage trace for soma and most distal dendrite is illustrated.

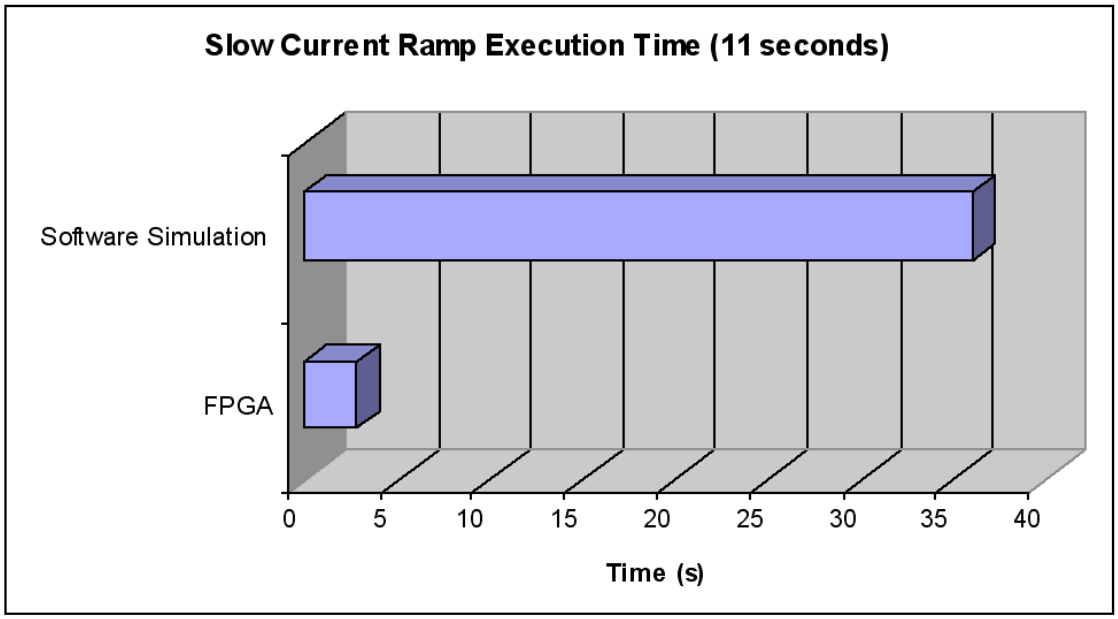


Figure 7: Graph depicting the simulation execution time between our custom simulation environment and the model as it run on an FPGA. The software model is compiled code implementing variable time-steps for performance. It was executing on a Pentium 4, 3.2 GHz with 2 GB RAM. The FPGA is at the current maximal frequency of 33 MHz and a simulation sample rate of 200 kHz.

The performance reported in this chapter is based on an unoptimized FPGA implementation. Without extensive optimizations, the model was capable of greater than 8M compartments/second or 4× real-time with a time-step of 5ms. This level of performance is significant as our custom software simulator executes at approximately 0.3× real-time as illustrated in Figure 7. Furthermore, our FPGA implementation can be refined using several techniques, for example, manual rebalancing of registers across the pipeline. Additionally, multiplier performance can be driven to over 100 MHz enabling additional throughput enhancement. We would expect a 10× improvement in performance is within reason bringing the system to 40× real-time.

Hardware design of models is substantially more difficult than software design, but can be made easier through the use of well defined hardware primitives and library cells. Our design is based on reusable components enabling fast additions or corrections to the model. This chapter demonstrates FPGA modeling as a solution to performance constrained simulations of complex neural models or dynamical systems.

CHAPTER IV

MANUALLY ENGINEERED MODEL CONSTRUCTION

Preface:

The previous chapter introduced a multi-compartment and multi-conductance model as a test-case for the purpose of learning strategies for successful implementation of neural models on FPGAs. We found that our first attempt produced a well performing model with throughput greater than 3× real-time. However, we also recognized limits in performance and ease of use. It was apparent that our initial attempt at using standard digital design techniques and intuition was insufficient when implementing neural models on FPGAs. This chapter utilizes the lessons learned and describes an engineered framework for which a neural modeler can readily implement their models.

The manual engineered approach described in this chapter was driven by three goals. First, we wanted to develop a set of rules and heuristics to guide the construction of models on FPGAs. Next, we wanted to evaluate the suitability of three generic types of neural models for implementation on an FPGA: a Hodgkin–Huxley-type model [40], a population of single-compartment models, and a multi-compartment model similar to the model of the previous chapter. Finally, we wanted to generate strategies for maximizing the overall performance of the FPGA-based model.

This chapter describes the first generalized framework for the manual construction of neural models. This consisted of a basic set of techniques for implementing generalized systems of first-order differential equations. Next, we proposed three candidate archictures covering a minimal model, an unconnected population of neurons, and an interconnected neural model which includes population models with synapses and multi-compartment models. Finally, specific operation optimizations were developed to improve the performance of each mathematical operation within the model.

This chapter as published [89] became the first amalgamation of rules, heuristics, and

techniques enabling neural modelers to utilize FPGAs. The candidate architectures described here cover a large swath of actively-research neural models. Furthermore, the performance optimizations performed at both an architectural level and at a per-operation level successfully boosted clock frequencies upwards of 100 MHz.

This chapter still left some questions unanswered. For example, the approach described here did not offer much support for reducing design errors. Trial and error fixing of design flaws still constitutes a large portion of the model development process. Finding ways to reduce the propensity of mistakes will be examined in the following chapter. Additionally, how data should be digitally transferred between the FPGA and a personal computer remains a question after these studies. This too will be addressed in later chapters. Finally, a method for floating-point to fixed-point precision determination was only partially addressed in this work. More progress on this question is forthcoming in later chapters.

4.1 Introduction

Recently, in our laboratory, several implementations of conductance-based neural models have emerged including Hodgkin–Huxley [40] and Booth–Rinzel [9] models [37], and a ten compartment motoneuron model [88]. However, each of these designs was somewhat specific to the model being implemented. This chapter expands on those efforts by describing generalized algorithms and architectures that provide a migration path for current software-based neural models into FPGA-based implementations.

4.2 Methods

4.2.1 FPGA Hardware

All designs were targeted towards a Xilinx XtremeDSP Development Kit-II based on a Nallatech BenONE carrier board consisting of one DIME-II module slot and populated with a Nallatech BenADDA expansion board. The BenADDA is preconfigured with a Xilinx Virtex-II FPGA (XC2V3000-4fg676), two 14-bit 65 MS/s analog to digital converters (ADC), two 14-bit 160 MS/s digital to analog converters (DAC) and one megabyte of on board SRAM.

All model designs were constructed using System Generator (ver. 6.3i), an add-on toolkit for Mathworks Simulink (ver. 6.2). System Generator provides a library of blocks that can be converted into an HDL (hardware description language) for synthesis. For simple blocks such as multiplexors, logic gates, and registers, the tool does a direct translation into the HDL. For more complex structures such as memories, multipliers, and adders, the tool relies on the Xilinx CORE Generator (CoreGen). CoreGen combines user specified design constraints (bit width, depth, etc.) with timing constraints (latency) and area constraints (parallel vs. serial). The Xilinx ISE Foundation 6.3i package was utilized for synthesis, place and routing, and generation of a bitstream for programming. For certain results, Synplicity's Simplify Pro (ver. 7.0), an alternative 3rd-party synthesis tool was employed for comparison.

System Generator provides a unique interface for FPGA digital design through its rich library of synthesizable blocks. Clocking is implicitly defined through the setting of sample

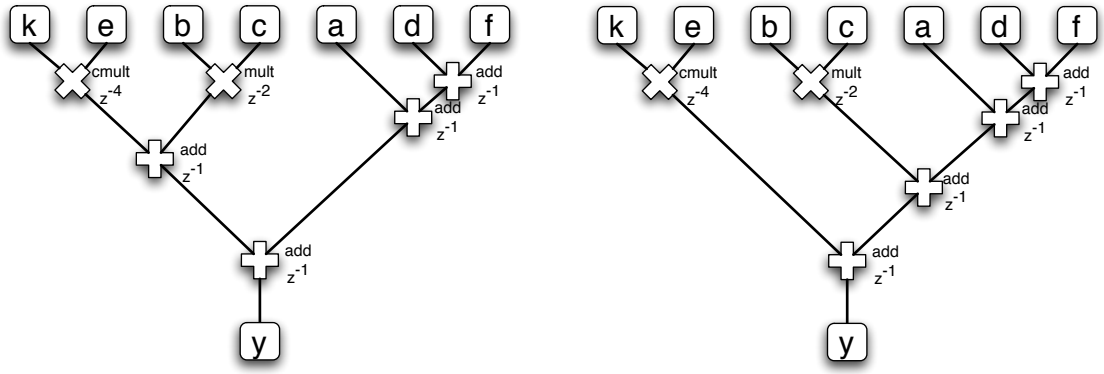
periods (of arbitrary units) for each block. Reset states are easily defined through the interface. Additionally, automated support for buses and explicit declaration of fixed-point type format simplify what would take a considerable amount of effort when programming in a traditional HDL. System Generator combines both an interface helpful to the traditional hardware designer while hiding underlying details to the neural modeler.

4.2.2 FPGA Building Blocks

The vast majority of processing within the data-path involves four blocks: *addsub*, *mult*, *cmult*, and lookup tables. The former three encompass the arithmetic operations (addition, subtraction, multiplication, multiplication by a constant) and the latter is for all other operations. Division is absent from this list as there is yet to be a high-speed, low latency implementation. The disadvantage is minimal as divisions can generally be mapped to a multiplication of the inverse of the denominator. (The denominator is very often a constant or parameter in neural models, making for a trivial implementation.)

Each block, when mapped into the FPGA, can take on a number of different forms, each with its own tradeoffs. Each block can be characterized by its throughput, latency, area, bit width, and sign format. The throughput is a measure of the number of operations that can be performed by the unit in a given amount of time. The latency of an operation is the delay represented as a maximum number of cycles the block requires to propagate an input to an output. When a block is pipelined (i.e. broken into multiple suboperations each of one cycle duration) or capable of completing one operation per cycle regardless of latency, there is often a negative correlation between throughput and latency. Resources within an FPGA are utilized in varying ways depending on the parameters of the block. Additional pipeline stages require additional registers per block, while a wider data-path requires more logic. Different architectures can save area while sacrificing performance, such as in the case of a sequential multiplier using a single accumulator to sum of partial products [94].

Optimally, the data-path will exploit as much parallelism in the model as possible. In the simple equation $ab + cd$, the multiplication of a and b can occur in parallel to the multiplication of c and d. Then the addition of the two terms can follow. The metric used



(a) Operation balancing

(b) Latency balancing

Figure 8: Example expression trees for the equation $a+bc+d+ke+f$ where k is constant. Two operand multiply operations are shown with two units of delay while constant multiplies have four units of delay. All adders are given one unit delay. Additional pipeline registers can be added arbitrarily to each operation. a) The tree is balanced with respect to the number of operations per path. b) The tree is balanced with respect to the number of cycles of latency per path.

for determining parallelism is not simply the number of operations. Instead, the latency of the operation has to be considered. In the equation $y = a + bc + d + ke + f$ where k is a constant, the multiplication of b and c, the constant multiplication, and a pairwise sum are done in parallel (see Figure 8). Additional arithmetic steps are done biased towards the paths of lesser latency. If each of the addition operations have a latency of one cycle, then the skewing of the addition steps away from the multiplies enables a balanced tree of $3-5$ cycles of latency per path. If the number of operations was the metric for balancing expression trees, the additions can be rebalanced shifting the latency per path range to between 2 cycles ($a \rightarrow y$) and 6 cycles ($k, e \rightarrow y$).

4.2.3 Look-up Tables

Often, the neural models require computations that are difficult, either resource heavy or too slow, for use in a neural model simulation. These can be trigonometric functions, exponentials, square roots, or any other trancendental function. When the difficult to evaluate expression is a function of a single input, a lookup table provides an area efficient and high performance way to estimate the output of the function. For an example, the steady state of a gating variable can be estimated via a sigmoid or Boltzmanns equation as

defined by:

$$m_\infty = \frac{1}{1 + \exp\left(\frac{V_{\text{mem}} - \theta}{\sigma}\right)} \tag{1}$$

where θ is the half activation voltage of the gate, σ is a measure of the slope of the sigmoid, and V_{mem} is the membrane potential. This equation is difficult to solve directly for two reasons. First, the exponential, as a trancendental function, has no simple closed form solution that is efficient on an FPGA. Second, the inverse function is as difficult as a division, which is generally solved iteratively, not directly like a multiplication. Division by can be simplified by reframing the expression as multiplication by a new parameter, $\frac{1}{\sigma}$.

Since Eq. (1) is difficult to evaluate directly in hardware, it is a good candidate for a lookup table. A ROM indexed by V_{mem} can produce a suitable estimation of m_∞, thus removing the need for any of the arithmetic operations within the equation. A simple mapping is required to convert the V_{mem} input to an address for the ROM. For the general case:

$$addr(x) = (x - \min(x)) \cdot \frac{2^n - 1}{\min(x) + \max(x)} \tag{2}$$

where n is the number of bits of addressability in the lookup table. The implementation requires a subtraction block and a multiplication by a constant to perform the linear transform. The output of this multiplication should be set to saturate to avoid overflows when addressing the table. This mapping can be shared for multiple lookup tables that use the same input.

In a Virtex-II FPGA, lookup tables are chosen to utilize SelectRAM, or block RAM within System Generator. A XC2V3000 FPGA contains 96 SelectRAM of which each contains 18 kbits of configurable RAM. Each SelectRAM can be configured as 512 x 36-bits, 1k x 18-bits, 2k x 9-bits, 4k x 4-bits, 8k x 2-bits, or 16k x 1-bit. Partial SelectRAMs are wasted, so each lookup table can expand to fill the full RAM. In general, lookup tables fit within the 1k x 18-bit configuration.

4.2.4 Arithmetic Performance

Generating optimal models often requires tradeoffs between pipeline depth and clock rate. In general, a deeper pipeline enables high clock rates. In cases where the logic is fully

utilizing the pipeline, a deeper pipeline should translate to higher performance, up until the point area on the FPGA becomes limited. In the case where the overall execution latency is to be minimized independent of the number of clock cycles, the clock rate is no longer the determinate of performance, but rather the timing delays through the pipeline. This delay is calculated as the product of the clock period (T) and one plus the pipeline depth ($1 + d$). This is a conservative estimate as it assumes that the operation uses the entirety of the clock period prior to and after the internal pipeline.

In order to determine the influence of latency on operation throughput and area, each operation was synthesized and mapped in the target Xilinx FPGA, and is shown in Figure 9. Daisy chained logic blocks (arbitrary length chain of 6 blocks) were utilized to obtain average performance results per operation. To mitigate any performance hit (by wire or logic delay) caused by interfacing the inputs and outputs of the operations, double-buffered registers were used to map directly to input and output pins. All operations were set at 14-bit 2s complement signed numbers with the fixed point set at the 13th bit. This allows a range of ± 1 with approximately four decimal places of accuracy. Synthesis was performed with the standard Xilinx Synthesis Technology (XST) built into ISE. Following place and route and generation of a bitstream, area utilization was assessed and the critical path from the log files was noted. The inverse of the critical path period was used as the peak clock frequency and is shown in Figure 9, plot a. The area per operation was split into two components, registers, or flip-flops, and look-up tables (LUTs). Each value plotted (see Figure 9, plot b) is the average area per operation; the base overhead (from the double-buffering of data converters) was subtracted from total utilization of registers and LUTs and the result divided by six to obtain the average.

System Generator provides an optimization flag, *Pipeline to Greatest Extent Possible* which was set for the adder operations but not set for the constant multiplier and standard multiplier tests. We did performance comparisons for each operation with this flag set and unset and found negligible differences. When set, however, the tool restricts the range of latencies that are programmable, in this case, limiting to at least three cycles for multiplications and one cycle for additions.

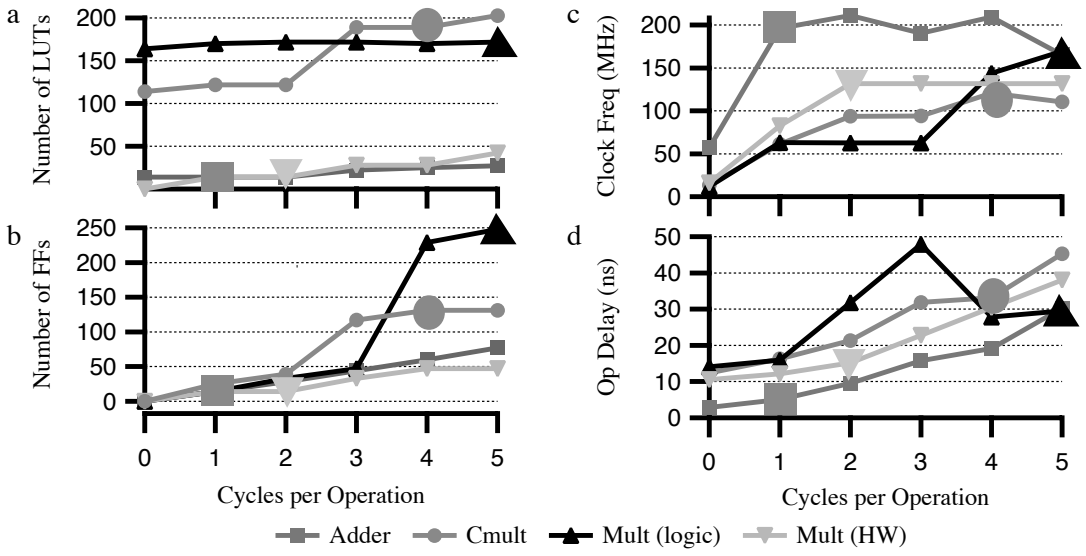


Figure 9: Area vs. Performance tradeoff. a, b) Performance and c, d) area utilization of basic arithmetic operations. The benchmarks were performed with 14-bit operands and 14-bit output. The inputs and outputs were double buffered and assigned to external pins on the FPGA. All performance results are derived post synthesis and place & route. a) Clock frequency is found as the maximum operating frequency when synthesized at for each cycle latency and operation. b) The operation delay here is shown as the delay (in ns) for an output to adjust following a change in an input. c, d) Area utilization is split between flip-flops and lookup tables, whereby two of each make a slice. The enlarged markers designate the proposed delays for each block to minimize the area vs. performance tradeoff. These plots also demonstrate the performance and area advantage to using adders and hardware embedded multipliers vs. constant multipliers and logic-based multipliers. When throughput is to be maximized above all else, than adders and logic-based multipliers are preferred.

Table 1 summarizes the results for optimizing to two different design goals (peak throughput and minimum latency) post synthesis and place & route. Area is depicted as the number of slices utilized, In the Virtex-II architecture, the logic fabric is broken up into CLBs, or Configurable Logic Blocks. Essentially, each CLB can output 8 bits of information. Within each CLB, there exists four slices and two tristate buffers. Each slice contains two 1-bit registers, two 1-bit lookup tables (LUTs), and dedicated arithmetic carry chains and SOP (sum of products) logic. Each LUT can be configured as one 4-bit addressable lookup table, one 16-bit RAM, or 16-bit shift register. (the XC2V3000 has 3584 CLBs). These slices may be partially utilized in this design but may be shared between multiple operations in a resource-constrained design. The slice count in table 1 shows the sum of all fully and partially utilized slices. This data shows that performance can be optimized for either throughput or latency depending upon the requirements, and that different design choices will have a large impact on overall model performance. When the resources and model architecture allow for a deep pipeline, each operation can be heavily pipelined maximizing throughput, where the frequency is the maximum operating frequency based on the critical path period. When pipelining is not desired (see Single-Cycle Architecture), peak performance is achieved with no cycle latency and minimal delay per operation. The block option flag Pipeline to Greatest Extent Possible had no noticeable effect on performance under these conditions and was disabled.

While we have not repeated this study for data widths greater than 14-bits, it is expected that similar trends will follow. Throughput performance is generally maximized when using larger pipelines. It is expected that the optimal throughput would be found when latency is set to higher values as the bit width increases. Area becomes especially constrained when data-paths become wider. The Resources Estimation Tool, included as part of System Generator is an invaluable resource for the FPGA modeler when implementing area-constrained designs [77].

We can utilize a formal approach for defining each operation as a discrete-time transfer function based on the cycle latency from input to output. Based on the timing results depicted in Figure 9, each operation can be represented by the following expressions, where

Table 1: Peak performance of operations

	Target	Max Throughput	Min Latency
Adder	Depth	2 cycles	0 cycles
	Frequency	211.2 MHz	-
	Delay	9.5 ns	2.9 ns
	Area	14 slices	6 slices
CMult	Depth	4 cycles	0 cycles
	Frequency	120.9 MHz	-
	Delay	33.1 ns	12.4 ns
	Area	98 slices	59 slices
Mult	Depth	5 cycles	0 cycles
	Frequency	169.7 MHz	-
	Delay	29.5 ns	10.5 ns
	Area[1]	135 slices	0 slices

x, y are intermediate values, states, or parameters, α is a constant, and p is the stage in the execution pipeline.

$$\begin{aligned} Add(x[p], y[p]) &= x[p-1] + y[p-1] \\ Sub(x[p], y[p]) &= x[p-1] - y[p-1] \\ CMult(k, x[p]) &= k \cdot x[p-4] \\ Mult(x[p], y[p]) &= x[p-2] \cdot y[p-2] \end{aligned} \tag{3}$$

These mappings redefine arithmetic operations for an FPGA implementation taking into account operational delay. The multiplier delays are valid for bit widths 18 (the size of the built in multipliers). Alternative mappings are defined for addition and subtraction based on particular architecture and design constraints.

4.2.5 External Interfacing and Data Collection

FPGA models execute at extremely high throughputs. Roughly speaking the performance level (defined as simulated time/execution time) is timestep multiplied by the number of models and the FPGA clock frequency then divided by the pipeline depth. This number is maximized only when the FPGA is completely utilized.

All of the models presented here are intended as examples and as such are not very complex. Consequently, they all have on-chip execution times of less than one millisecond

(even if they were all placed on the chip simultaneously.) Thus, for the sake of convenience, the data presented here is generally from emulation of the FPGA directly in Simulink.

4.3 The Base Architecture

4.3.1 FitzHugh-Nagumo

For ease of presentation we will present the architecture as an example implementation of a simple neuron model. However, the ideas can be applied to any neuron model. The FitzHugh–Nagumo model [32, 58] is a reduced, dimensionless representation of the Hodgkin and Huxley model [40]. This model makes the following assumptions: 1) the activation gate of the sodium channel has extremely fast kinetics and therefore reaches the steady state value instantaneously, and 2) the potassium channel gate has similar, but reverse kinetics (time-scale and gate characteristics) to the inactivating gate of the sodium channel. The Hodgkin and Huxley equations centred around four ordinary differential equations of voltage (V_{mem}), sodium activation (m), sodium inactivation (h), and potassium activation (n) can be reduced to a simple potential state and a recovery state. When non-dimensionalized, the following coupled differential equations emerge:

$$\frac{du}{dt} = u - \frac{1}{3}u^3 - w + I \tag{4}$$

$$\frac{dw}{dt} = \epsilon\left(b_0 + b_1 u - w\right) \tag{5}$$

where u is the potential of the system and w is the recovery state. The parameters ϵ, b_0, and b_1 modulate the shape of the spike and I is the input (in a dimensionless current) to the system.

This model, despite being drastically simplified, has characteristics that make it stereotypical of neural circuit models. Each equation is a first-order ordinary differential equation with Eq. (4) having a nonlinear term. The equations are coupled and cannot be solved analytically. This system enables a demonstration of our techniques for FPGA model development in an easy to understand example.

4.3.2 Generating the Data-Path

Differential equations of the standard form can be split into two terms: the differential term or the time varying state variable, and the intermediate calculation, or equation for the rate of change of the state variable. In Eq. (4) and Eq. (5), relative to the equal sign, the left-hand side is the differential term and the right-hand side is denoted the intermediate term. It is the intermediate term that will be converted into a data-path for calculation. Defining the functions f and g from Eq. (4) and Eq. (5), respectively, as follows:

$$\begin{aligned} f(u, w) &= u - \frac{1}{3}u^3 - w + I \\ g(u, w) &= \epsilon\,(b_0 + b_1 u - w) \end{aligned} \tag{6}$$

makes clear the delineation between the state and the intermediate; the generated data-path becomes independent of any particular numerical solving techniques. First- and higher-order, fixed time step and variable time step solvers can be implemented around the data-path without any modification to the design. Additionally this isolates the data-path from any particular simulation or protocol requirements, such as starting, stopping, and resetting. As will be described later in this chapter, this separated data-path provides a general case for rapid mapping into various architectures including population and multicompartment modeling.

4.3.3 ODE to Difference Equation

Each equation defined in the continuous time domain must be mapped to discrete time for numerical analysis. Simulation on a general-purpose computer can utilize the following discrete time representation by means of forward-Euler integration, where n is the iteration step:

$$\begin{aligned} u[n+1] &= u[n] + \Delta t \left(u[n] - \frac{1}{3}u[n]^3 - w[n] + I \right) \\ w[n+1] &= w[n] + \Delta t \cdot \epsilon\,(b_0 + b_1 u - w) \end{aligned} \tag{7}$$

The above equations can readily be calculated in a general-purpose processor where instructions are executed sequentially for each iteration of the loop. These equations are

not explicit with respect to processing in an FPGA, where each operation has timing requirements that must be considered. Additionally, there is no explicit parallelism defined. Utilizing the commutative and associative property of addition and subtraction, we reorder the arithmetic operations to enable parallelism based on operation latency as described in the expression tree in Figure 8.

$$\begin{aligned} f(u,w) &= \left[((u-w)+I) - \left((u\cdot u)\cdot\left(\frac{1}{3}u\right)\right)\right] \\ g(u,w) &= [\epsilon\cdot((b_0-w)+(b_1\cdot u))] \end{aligned} \tag{8}$$

In general, multiplications should be performed as early as possible in the calculation as the latency is greatest. This will skew the resulting expression tree to force faster additions and constant multiplications outside of the maximum latency path, when possible. To formally define the above expressions in the operation space of the FPGA, we can use the mappings defined in Eq. (3) to the redefined functions f and g and obtain:

$$\begin{aligned} f(u[p],w[p]) &= \mathrm{SUB}(\mathrm{ADD}(\mathrm{SUB}(u[p],w[p]),I[p]),\mathrm{MULT}(\mathrm{MULT}(u[p],w[p]),\mathrm{CMULT}(1/3,u[p]))) \\ g(u[p],w[p]) &= \mathrm{MULT}(\epsilon[p],\mathrm{ADD}(\mathrm{SUB}(b_0[p],w[p]),\mathrm{MULT}(b_1[p],u[p]))) \end{aligned} \tag{9}$$

Evaluating these mappings yields Eq. (10). This representation is useful to describe the overall delay through the data-path, in this example, 6 cycles. Since there is a skew between the shortest latency paths and the longest latency paths (from 1 cycle to 6 cycles), additional pipeline registers can be added in the shorter delays without increasing the pipeline depth, possibly enabling increased throughput with a faster clock rate. Additions and subtractions are set to have no latency, but this can be readily changed by reindexing the parameters and states by the additional delay.

$$\begin{aligned} f(u[p],w[p]) &= ((u[p-1]-w[p-1])+I[p-1]) - \left((u[p-6]\cdot u[p-6])\cdot\left(\frac{1}{3}u[p-4]\right)\right) \\ g(u[p],w[p]) &= \epsilon[p-3]\cdot((b_0[p-3]-w[p-3])+(b_1[p-6]\cdot u[p-6])) \end{aligned} \tag{10}$$

These equations define the required cycle latency for the input to reach the output synchronously. Once implemented, this data-path can be used for well-performing implementations of single-compartment models, multi-compartment models, population models, etc. The data-path can be constructed identically in all of the above cases. Figure 10 shows

the System Generator data-path corresponding to Eq. (10). For these different architectures, only the implementation of the state variables will change as is expounded upon in the following sections.

4.4 Single Model, Single Unit Case

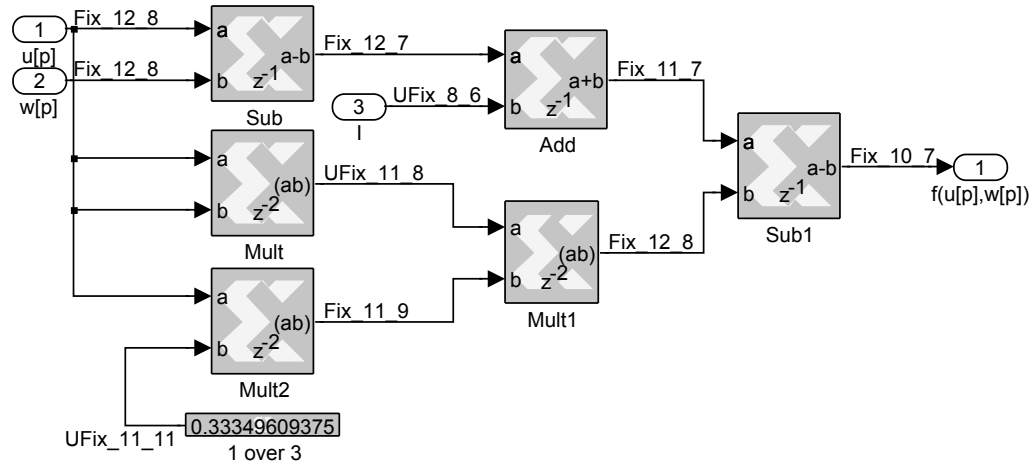
4.4.1 Multi-cycle Architecture

The general model simulator will execute a single version of a model according to a set protocol. The modeler will often run simulations to manually tune parameters, trying to replicate a particular behavior. Sometimes this model will have particular performance requirements such as real-time execution. For these cases, we developed a general architecture for running a series of differential equations with arbitrary delay as a multi-cycle processor. We employed two synchronous clock domains, one providing a slower outer loop to interface with the outside world and a faster inner loop for the data-path processing. In this way, it is very much like a multi-cycle computer architecture, overclocking the internal pipeline in a hidden fashion from the outside.

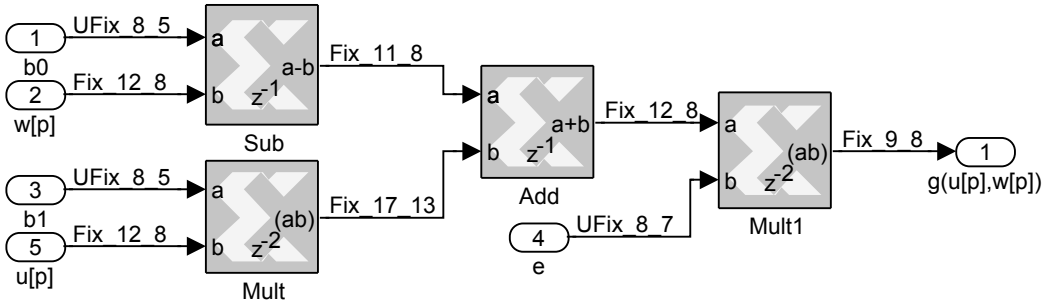
The original equations for the state variables, u and w can be expressed as difference equations around the functions f and g. We use forward-Euler integration as a numerical solver due to its ease of implementation. Additionally, numerical accuracy can be improved with smaller step sizes, a reasonable tradeoff given the high performance of an FPGA. This architecture is easily extendable to higher order ODE solvers including Runga-Kutta, Predictor-Corrector, or even variable time-step solvers if desired.

$$\begin{aligned} \frac{du}{dt} &= f(u,w) \rightarrow u[n+1] = u[n] + \Delta t \cdot f(u[n], w[n]) \\ \frac{dw}{dt} &= g(u,w) \rightarrow w[n+1] = w[n] + \Delta t \cdot g(u[n], w[n]) \end{aligned} \tag{11}$$

Eq. (11) representing the state calculation can be combined with the data-path formulation in Eq. (10) to generate the full expression for the model. The data-path needs to be modified to include the time-step multiplication adding another 2 to 4 delays to the data-path. The total delay for the data-path is increased to 7 clock cycles. The translation of the state equation into hardware is only a single register clocked at the slower clock rate.



(a) $f(u, w)$ block diagram



(b) $g(u, w)$ block diagram

Figure 10: Simulink block diagram of intermediate calculations a) $f(u, w)$ and b) $g(u, w)$. Each operation is identified by the label on the block. All fixed-point data types are labeled on the wires interconnecting the blocks, where the first portion (Fix/UFix) defines the value to be signed or unsigned, respectively. The second term is the number of total bits in the representation and the last term defines the number of fractional bits, or the number of bits to the left of the decimal place. Data ports for the subsystem are the numbered oval blocks, where the inputs are parameters or states and the output is the intermediate calculation. Delays through the system are expressed in the z domain where the superscript is the latency in cycles from output back to input. Constants are shown with additional significant digits to illustrate the affect of quanitization.

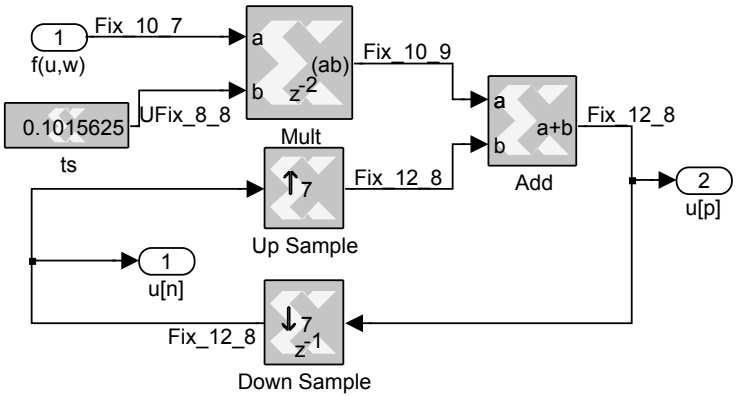
Eq. (12) are the combined data-path and state expression where n is the iteration of the outer, slower clock and p is the iteration of the pipeline, or faster clock.

$$\begin{aligned} u[n+1,p] &= u[n,p] + \Delta t \cdot \Bigg[\quad ((u[n,p-2] - w[n,p-2]) + I[n,p-2]) - \cdots \\ &\qquad \left((u[n,p-7] \cdot u[n,p-7]) \cdot \left(\frac{1}{3} u[n,p-5] \right) \right) \Bigg] \\ w[n+1,p] &= w[n,p] + \Delta t \cdot \Bigg[\quad \epsilon[n,p-3] \cdot ((b_0[n,p-3] - w[n,p-3]) + \cdots \\ &\qquad (b_1[n,p-6] \cdot u[n,p-6])) \Bigg] \end{aligned} \tag{12}$$

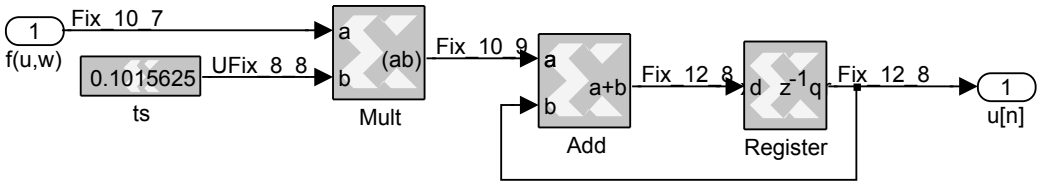
In this architecture, a simplification emerges enabled by the slower clocked state. All execution paths within the data-path must settle by the time the next value is clocked into the state register. Therefore, for paths of fewer cycles than the critical path (in this case, the path with the longest latency), there is no difference if the inputs arrive earlier than required. Therefore, all paths can receive their input on the previous outer cycle and latch the new value at following outer cycle. All timing requirements of the pipeline with respect to insertion delay in the pipeline become unnecessary. The new equations follow from the simplification:

$$\begin{aligned} u[n+1] &= u[n] + \Delta t \cdot \left[((u[n] - w[n]) + I[n]) - \left((u[n] \cdot u[n]) \cdot \left(\frac{1}{3} u[n] \right) \right) \right] \\ w[n+1] &= w[n] + \Delta t \cdot [\epsilon \cdot ((b_o[n] - w[n]) + (b_1[n] \cdot u[n]))] \end{aligned} \tag{13}$$

The forward-Euler method is relatively simple to implement within System Generator, taking only four blocks as shown in Figure 11(a). The state at each iteration is stored in a single register clocked at the output rate. We use a combination of an *up sample* and *down sample* block (the clock multiplier/divider is set to the maximum delay through the data-path, including any calculation within the state). The *up sample* block is configured to copy the value from each input period to all corresponding output periods. This is not necessary for hardware generation, *i.e.* it does not translate directly to hardware, but does allow the software to verify correct clocking of data through the system. The *down sample* block is configured to copy the last frame of the input to the output, which in hardware is a register clocked at the output rate.



(a) Mult-cycle state approach



(b) Single-cycle state approach

Figure 11: Simulink block diagram of state calculations for single unit models. a) The multi-cycle approach to single unit forward-Euler integration. The intermediate calculation is scaled by the time constant and added to the previous state. The output of the state, $u[n]$ is clocked once for every seven cycles of $u[p]$. The *Up Sample* block is set to copy the slower clocked samples across all seven fast clock cycles while the *Down Sample* block is implemented as a register at the slower rate. b) By removing the delays within the pipeline, up and down sampling becomes unnecessary, requiring only a register to store state between iterations. There is only one clock rate in this scenario.

The remainder of the state consists of a multiplier block at the output of the data-path to scale the function by the time step and an adder to add to the previous value. The previous value is at the output of the up sample block. More complex integration algorithms can be implemented as an extension to this base case. The multiplication by the time step can also be incorporated within the data-path depending on the particular integration algorithm possibly saving a multiplication step. This can be shown for the function $g(u, w)$ where the constant multiplication of ϵ can be substituted by the constant multiplication of the product $\epsilon \cdot \Delta t$. This is only possible in the trivial case of Euler integration with fixed step sizes.

4.4.2 Single-cycle Architecture

The multi-cycle architecture enables a reduction in state registers while still utilizing a fully-pipelined data-path. It provides a means of integrating a pipeline of arbitrary depth into an integration state subsystem producing only one output per time step. There are three drawbacks with this approach: wasted area, reduced performance, and high power consumption.

First, area is not utilized efficiently as pipeline delays within the intermediate calculations use registers that could be used for additional logic or extra data storage. Each delay requires a number of registers equal to the bit width of that calculation. Significant area savings can be realized by removing those extra delays.

Second, performance is reduced for two reasons: 1) Extra delays contribute an additional time delay in the form of a setup and hold time. The register setup time, or the minimum amount of delay between the data becoming stable prior to the edge of the clock signal, for an XC2V3000 speed grade -4 FPGA is 370 ps. The hold time, or the minimum duration following the clock edge for the data to be ready to read is 90 ps. The sum of those values constitutes a window around the clock edge where data must be stable and is wasted within the sample period. Long pipelines can accumulate this 460 ps dead-time in the period for each delay in the path. 2) The overall delay through the pipeline is equal to the product of the depth and cycle period. Ignoring setup and hold times, the delay

through the pipeline is ideally the sum of all the arithmetic combinatorial logic delays. If the total logic can be equally distributed (delay-wise) between an arbitrary number of registers, then latency/throughput is independent from the pipeline depth. Practically, the period is a function of the longest combinatorial path between registers. Therefore, shorter combinational paths must execute within larger clock periods, reducing efficiency. As the number of pipeline registers increase, the more difficult it is to maintain symmetry between combinatorial path delays.

Third, power consumption and clock frequency are directly proportional for a given model design. The power consumption of a device is not generally an issue for the modeler, but does constrain the design of the device itself; the peak clock frequency of an FPGA is partially constrained by the limits of heat dissipation as power consumption increases. Achieving the same or increased throughput at a slower clock rate is generally preferred.

Therefore, to utilize minimal area and power while achieving peak performance requires the reduction of the pipeline depth to the minimum achievable. The majority of neural models can be modified to execute in a single cycle per time step iteration by changing all latencies to zero. (Note that one register always remains due to integration, resulting in the "single-cycle" designation.) For multiplier blocks, the "Pipeline to Greatest Extent Possible" flag must be unchecked. This change can be made to all arithmetic blocks. Then the up-sample and down-sample blocks can be changed to a single register to complete the single-cycle design approach.

The results of a comparison between the two design approaches are shown in Table 2 with the area utilization and performance results determined post synthesis and place & route targeting an XC2V3000-4fg676 FPGA. The top-level design depicted Figure 12 was used for testing and synthesizing the single-cycle and multi-cycle model. In the single-cycle version, the entire data-path is executed within each clock cycle and requires only 17 ns to complete. When delays are distributed in the multi-cycle, a total of 7 for the longest paths, the latency is tripled. The maximum clock frequency is increased by almost 2.5 times, not enough to compensate for the additional cycles required per iteration. The maximum frequency supported by the XtremeDSP-II Development Board is 120 MHz capping the

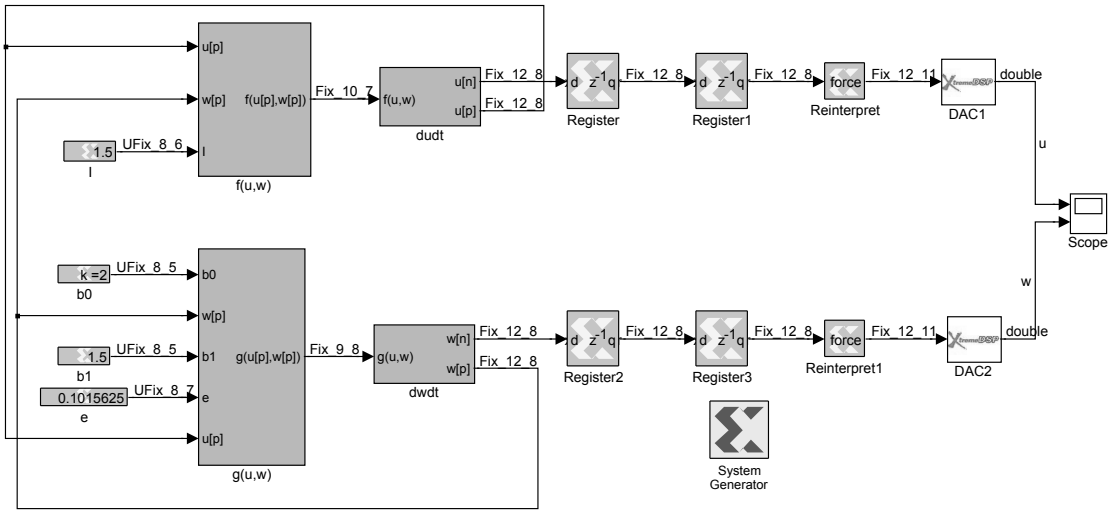


Figure 12: Top-level view of the FitzHugh–Nagumo model. Parameters are defined in Xilinx constant blocks (on the left) moving into the intermediate calculation. The states are evaluated next, with feedback paths to the inputs of the intermediate calculations. The outputs, u and w are double-buffered for performance and scaled for analog output via the on-board data converters (on the right).

Table 2: Single model architecture design comparison.

	Frequency (MHz)	Output Frequency (MHz)	Latency (ns)	Area (Slices)
Single-Cycle	58.9	58.9	17.0	94
Multi-Cycle	143.6	20.5	48.7	143

peak multi-cycle model throughput.

In this example, the single-cycle model enables a 187% improvement in performance with a 34% reduction in area over the multi-cycle model. In general, the single-cycle method is preferred over the multi-cycle method when all the blocks within the data-path can be set to zero latency. When that is not the case, the multi-cycle method is a suitable fallback technique.

4.5 Multiple Model, Single Unit Case

Deep pipelines allow for multiple simultaneous processes executing within the data-path, where the number of processes equals the depth of the pipeline. In other words, if the data-path requires a latency of 10 cycles until the first output appears, 10 simultaneous

models can be executed without a loss of performance. The data-path would produce the 10 models interleaved at the inner, faster clock rate. In contrast, within the single-model, multi-cycle architecture, the pipeline produces an output at the outer, slower clock frequency. Ultimately, the throughput of each model does not change, each output for the same model will be at the slower frequency, but the aggregate bandwidth of the system will substantially improve.

Two scenarios are common candidates for a multiple model, single unit simulation: 1) There are a set number of models of interest to simulate, for example, all the neurons in a particular nucleus or circuit or 2) when the number of simultaneous simulations are flexible and more is better, such as in automated parameter searching or population modeling. The first scenario applies more constraints to the model and produces a deterministic output. Only the available area on the FPGA limits the second scenario. Within these architectures, a change in the number of models simulated requires a straightforward modification to the model design.

4.5.1 Pipelining the Data-path

The structure of the arithmetic operations in the data-path in the multiple model case is identical to that of the single model case. The differences lie in distributing latencies and managing the parameters in the pipeline.

Latencies, or additional clock cycle delays, are inserted to maximize throughput of the system. As the longest combinatorial path between any two registers is the sole determinate of clock frequency and therefore model throughput, care must be taken to distribute the delay as uniform as possible throughout the pipeline. The following algorithm describes an approach to distributing the latency within the data-path:

1. The target number of simultaneous models will set the depth of the subsequent pipeline generated.

2. The longest, weighted arithmetic path is isolated and delays are judiciously added such that the total delay does not exceed the depth of the pipeline. The longest, weighted arithmetic path is defined as starting from a single endpoint (parameter of

the system or a state) and terminating with the completion of the differential of the state.

(a) Delays are added in an initial pass providing a ratio of 4:2:1 cycles (see Eq. (3)) of latency to constant multipliers, hardware multipliers, and additions/subtractions, respectively. Non-hardware embedded multipliers have similar delay requirements to constant multipliers.

(b) Add an additional delay for each operand of a multiplier that is greater than 18-bits.

(c) Tables, both ROM (read-only memory) and single- and multi-port RAM (random access memory) blocks require one unit of delay regardless of bit-width or addressability when fit into one SelectRAM. Additional glue logic is required when more then one SelectRAM is required which could benefit from an additional delay.

(d) When the number of delays in the pipeline exceeds the number of simultaneous models, remove delays evenly throughout the path until the number of delays equals the number of models.

3. Repeat on all other paths taking care to never use more latency cycles then the critical path. Adding extra delays such that all paths are balanced are not necessary and will waste FPGA resources.

This algorithm post-processes the expression trees making up the data-path solving the intermediate calculation, as defined above, with the timing information required to interface with the state solvers and parameter subsystems. The leaves of these expression trees are the parameters and the state inputs. Each leaf has an insertion delay associated with it defined as the sum of the delays along the path from the leaf to the root of the tree, including any calculation that may occur within the state solver (ex. multiplication by the time constant). For example, in Figure 8(b), k, a, b, c, d, e, and f, are mapped to $k[p-5]$, $a[p-3]$, $b[p-4]$, $c[p-4]$, $d[p-4]$, $e[p-5]$, and $f[p-4]$, respectively. Synchronization of the

paths within the expression trees is therefore accomplished by providing delayed version of states and parameters to the leaves consistent with the insertion delay of the particular leaf node.

4.5.2 Pipelining the States

Executing n models simultaneously requires the continuous storage of n sets of information within a pipeline that is n stages deep. In previous work [37], all information was stored within the pipeline via delay blocks, requiring careful synchronization of the expressions. States were implemented as a simple delay block with n cycles of latency. This architecture recognizes the states and parameters as forming a basis set of model information. All intermediates can be shown to be a function of the states and parameters. Therefore, only the states must be explicitly stored within each time step.

The delays of the previous work are replaced with a chain of n registers. This has two benefits: First, each register can now contain an initial value for the state that can be unique for each model simulated. By convention, the tail of the chain (last output) contains the initial state of the first model and the head register of the chain stores the state for the last model. Second, the outputs of the n registers represent the set of all delayed versions of the state. These outputs are now accessible to be routed back into the expression trees at the delay required for the particular operation. For example, if a change on a particular state leaf of the expression tree has an insertion delay of 6 cycles, then the input of that state should be tapped 6 registers deep in the state pipeline. This allows the state information to follow cycle by cycle the intermediate logic within the expression tree. This algorithm applied to the FitzHugh–Nagumo model is shown in Figure 13.

4.5.3 Shared vs. Unique Parameters

When executing a set of models, parameters can be either static across all models or arbitrarily varying across all models. In the simple case where a parameter is static, it can be represented as a System Generator constant block and has no particular timing requirements. Unique parameters per model can readily be exploited through the use of a circular buffer of length n counting through each parameter per cycle. These parameters must be

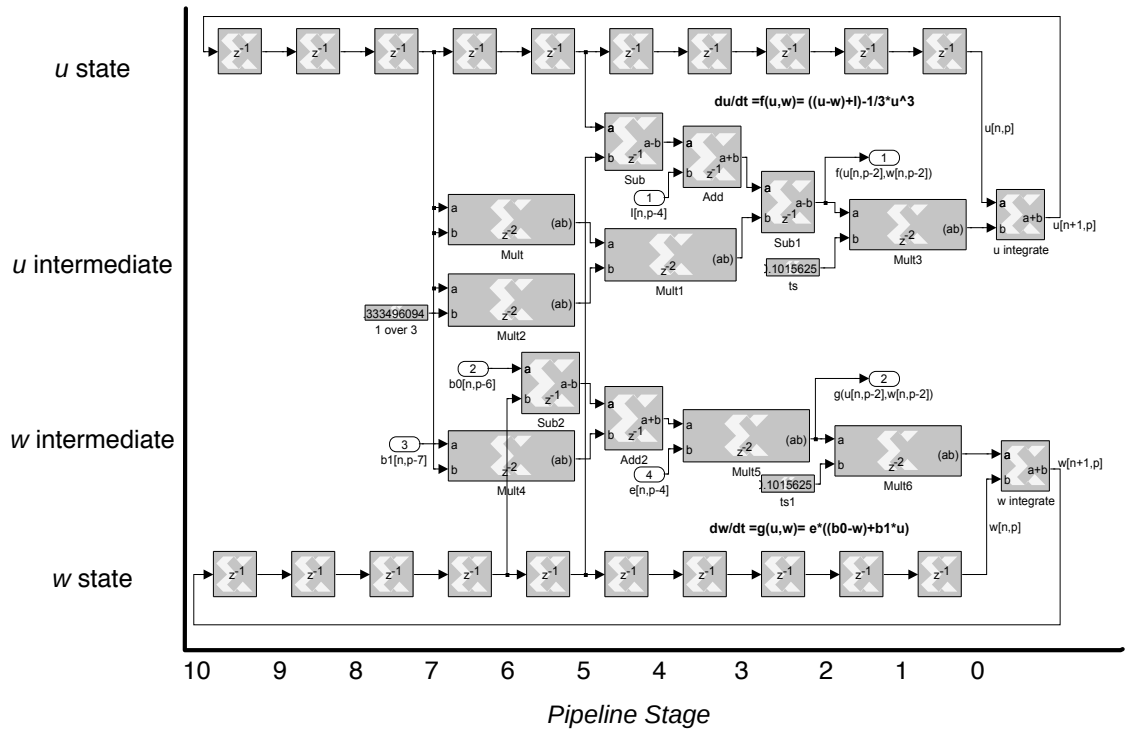


Figure 13: Simulink block diagram of a multiple-unit model. Ten iterations of the FitzHugh-Nagumo model are run simultaneously through the constructed 10 stage pipeline. All blocks are depicted to be in scale with respect to cycle latency. Since the data path requires only 7 cycles per iteration, the state register chain must be at least 7 stages within this architecture. It was chosen to be 10 for this example.

synchronously available relative to the target model at the correct insertion point within the model.

Given an input I, delayed by d cycles, as represented in the difference equation as $I[p-d]$, within a pipeline of depth n, the circular buffer must be initiated with parameters forward rotated by $n-d$ steps in order to maintain synchronization. Therefore, after $n-d$ increments of the pipeline, the parameter I will be inserted into the pipeline such that d cycles later, the output for that model will appear at the root of the expression tree.

In System Generator, this circular buffer is implemented as a count limited counter ranging from 0 to $n-1$ addressing a ROM with the parameter values pre-initialized. The pre-rotation of the circular buffer can be accomplished in two ways, either through a change in the initial value of the counter or by a rotation in the initial values in the ROM. The former method requires a dedicated counter per unique parameter set in the model. Therefore, the

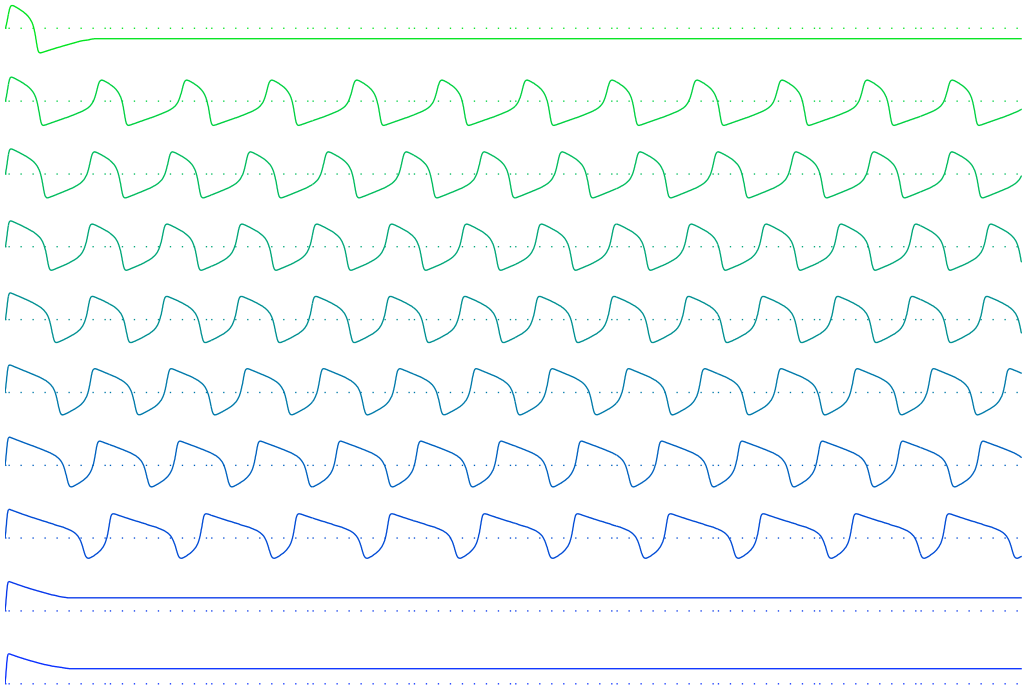


Figure 14: Output traces from a cycle-accurate, fixed-point simulation within Simulink of 10 concurrently executing Fitzhugh-Nagumo models. The traces were generated from the model shown in Figure 13. All parameters were kept constant with the exception of I, which was varied from 0.99 to 2.97 with a step of 0.22 illustrated from top trace to bottom trace. The first 3000 points are shown. Note that this simulation would execute in approximately 0.25 ms on the FPGA, and 50 such designs could run concurrently for a total of 500 models.

latter method is preferred as a rotation in the ROM requires no extra resources allowing one counter to be shared for all parameter tables. This approach was used for the "current" input I, for the traces illustrated in Figure 14.

The ROM macro in System Generator becomes synthesized as a synchronous memory such that the output is registered. Rotating the buffer by $n-d-1$ indices compensates for this one-cycle latency. When n is relatively small ($n < 15$), it is advisable to use distributed RAM resources when available. In distributed RAM, each slice can store up to 32-bits of data. When n is large or when logic resources are limited, these parameter tables can be kept in block RAM. The decision to use block RAM or distributed RAM largely depends on the particular limiting resource constraint in a model.

4.6 Coupled, Multiple Unit Case

Coupled, multiple unit models are a straightforward extension to the isolated multiple model, single unit case. Coupling can occur between compartments in a morphologically complex neuron model, as electrical connection between neurons in the form of gap junctions, or as chemical connections, or synapses within a population. This case deals exclusively with homogenous populations of neurons or neural compartments with some form of coupling or real-time interaction. A particular example of a ten-compartment motoneuron designed in this manner is described in previous work within this laboratory [88].

A coupling is defined as a state variable from one unit acting as a term or factor of an intermediate calculation of a different unit. A unit can be coupled to all other units of a model in the case of a fully interconnected population model. In contrast, a unit might only be coupled to its neighbours in the case of a linear multi-compartment chain. These two cases are considered the general cases of unit coupling, where there is regularity between the connections.

When modeling non-general cases where few connections are created, it is often simpler to map the scenario back into a general case when possible. This may not turn out to be the most optimal implementation. For example, in a model of 20 neurons such that each neuron is coupled to another to form pairs of half-center oscillators, each neuron will take input from only one other neuron and output to only one neuron. If adjacent neurons within the pipeline are coupled, then odd neurons will couple to the next neuron and even neurons will couple to the previous neuron. The following describes two such approaches to generating this coupling logic.

The first approach is a literal conversion of the coupling algorithm into System Generator blocks. An even-odd test can be accomplished by using the LSB (least significant bit) of the parameter set address counter as a select line into a 2:1 multiplexer. The counter will go from 0 to 19 in this example, toggling the LSB at each cycle. The inputs of the multiplexer will be state from the adjacent unit. Following the convention where the first model is at the tail of the register chain, when the multiplexer select line is 0, the unit is odd and requires the state from the following unit. The state used will be tapped from the state register

chain at the point of the insertion delay of the leaf node plus one. When the multiplexer select line is one, then the unit is even, and the previous units state is used, which will be tapped at the insertion delay minus one. Any parameters acting on the coupled state can be processed as usual with a 20 element ROM.

The second approach generalizes the coupling and removes the multiplexer from the implementation. This approach provides for two inputs to each model, one from the previous unit and one from the next unit. Very often neural models have an intensity parameter in the form of a maximal current or conductance. These parameters can be set to zero for the cases where there is no coupling and the proper value when there is coupling. Two parameter tables will be required, one for the even units and one for the odd. This approach, while wasteful in resources, simplifies design as only the standard arithmetic blocks are required.

Models requiring full interconnection between all units are reasonable and straightforward to design but are generally resource constraining. In the case of synapses for a fully interconnected population of n neurons, given recurrent connections, the logic requirements include n^2 synapses with n implementations of the synaptic mechanism including at least n state solvers, n parameter tables of n depth hold the synaptic weights, and $n-1$ adders in a tree with $\log_2(n)$ levels to sum the synaptic input. As n becomes larger, the synapses take on an ever-increasing percentage of FPGA resources, quickly limiting the scale of population models. Future work is needed to consider alternative design approaches for increasing the size of neural population models.

4.7 Discussion

This chapter serves to provide the methods for implementing stereotypical neural circuit model elements in an FPGA. While designing a model in itself is straightforward, there are some key limitations and areas of future work to make it as easy to use as a software simulation. This discussion documents the challenges of converting floating-point calculations to fixed-point representations, interfacing the model to external systems, and the limits of scalability within an FPGA.

4.7.1 Precision Determination

System Generator provides no means for handling real, or floating-point numbers. Instead, all parameters, states, and intermediate calculations utilize fixed-point numbers, defined by a sign, number of total bits, and number of fractional bits. Before executing in hardware, it is necessary to set each operation to have sufficient precision to avoid overflows, underflows, or functional mismatches due to quantization errors. Excessive precision should be avoided as area utilization and performance will suffer.

Optimal precision for all operations is a difficult if not an impossible goal and is still an active area of research and is revisted again in later chapters. Ignoring an optimal result, the number of bits required to guarantee full precision continuously increases after each operation. (Full precision here means precision based on the argument precisions rather than the arguments themselves). Full precision for a multiplication, as implemented in System Generator, is the sum of the number of bits of each operand. An addition has full precision when the number of integer bits of the output is the maximum of the integer bits of each operand. Similarly, the number of fractional bits of the output is the maximum of the number of fractional bits of each operand. This is a worst-case, pessimistic approach to determining precision through the pipeline by assuming that all the full range of values are valid for each operation and that maximum fractional precision is always necessary. As an example, a 17-bit multiply (17-bit unsigned or 18-bit signed inputs) requires just one embedded multiplier. A 34-bit multiply requires 4 embedded multipliers to calculate the partial products and adders to sum the two partial products. When moving to a 51-bit multiplier, the resources jump to 9 embedded multipliers [77]. Excess precision will require additional latency to maintain the same throughput and waste logic resources that could be used for additional parallel operations.

We have found several techniques to reduce the required precision but have yet to report on a general algorithm for determining the optimal precision. First, we can reduce the number of integer bits per operation by bounding the parameters and states within practical ranges. For example, for a particular neuron firing, the membrane potential might range from -70.00 mV to 30.00 mV requiring one sign bit, seven integer bits, and a number

Table 3: Calculations to determine range values per operation type. Division assumes that the range of the inputs does not cross zero.

$R' = R_1 \bigcirc R_2$	High Range Value (H')	Low Range Value (L')
'+'	$H_1 + H2$	$L_1 + L_2$
'-'	$\max(H_1 - L_2, L_1 - H_2)$	$\min(H_1 - L_2, L_1 - H_2)$
'×'	$\max(L_1 \cdot L_2, L_1 \cdot H_2, H_1 \cdot L_2, H_1 \cdot H_2)$	$\min(L_1 \cdot L_2, L_1 \cdot H_2, H_1 \cdot L_2, H_1 \cdot H_2)$
'/'	$\max(L_1/L_2, L_1/H_2, H_1/L_2, H_1/H_2)$	$\min(L_1/L_2, L_1/H_2, H_1/L_2, H_1/H_2)$

of fractional bits to achieve the desired resolution. Those signed seven integer bits allow a range of -127 to 126. Full precision uses the full range of the fixed-point representation, but in reality, only the usable range is required. Given S denoting a signed number (vs. an unsigned, positive only value) and $R_i = (L_i,H_i)$, where L_i and H_i are the low and high values of the usable range of the i-th leaf node. When L_i and H_i are of different signs or both negative, the number is signed and requires an extra bit to denote the sign. Formally, the sign, S_i and the number of integer bits, Z_i, given a range R_i is determined by:

$$S_i = \begin{cases} 1 & L_i < 0 \\ 1 & H_i < 0 \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

$$Z_i = \begin{cases} \lfloor \log_2(maxint) + 1 \rfloor & maxint > 0 \quad (maxint = \max(\lfloor |L_i| \rfloor, \lfloor |H_i| \rfloor)) \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

Using the range notation, R_i, instead of the full precision to represent each value, the number of integer bits, Z_i, required throughout the pipeline is generally reduced. We recalculate these range values at the output of each operation using the expressions in Table 3. Without further *a priori* knowledge of the range of intermediate values within the pipeline, this provides an approach to determining the number of integer bits required throughout the data-path, avoiding any overflow conditions.

Underflow conditions are more difficult to optimize around. An underflow occurs when the fraction precision of an output of an operation is truncated from full precision such that

a small number is represented as zero. Underflows may or may not cause any discrepancies in a simulation, depending on where in the data path they occur. In the case of a Hodgkin–Huxley style potassium channel with n^4 kinetics, given an activating gate with f fractional bits of precision, the output via two cascading multiplications will require $4 \cdot f$ fractional bits when utilizing full precision. Realistically, fourth-order kinetics requires no additional precision over a first-order expression, in which case an underflow would be acceptable. An adder tree combine all currents calculated per channel would also be a possible candidate for allowing underflows as very small currents are diminished by larger transients or leakage paths. Other calculations, such as scaling by the time step when performing integration, must be free from underflows to maintain functional behaviors.

The remaining class of errors, quantization errors, is substantially more difficult to reason through intuitively and require a more rigorous approach. This can be through error analysis techniques such as propagating relative and absolute error through the data-path. The inherent feedback in the system via the integration steps makes this analysis exceedingly difficult. Alternatively, simulations can help isolate the differences between floating- and fixed-point representations of the model. There is still considerable work to be done in investigating ways of analyzing fixed-point neural models to minimize quantization error.

4.7.2 External Interfacing

By utilizing the techniques described in this text, a modeler can readily implement a custom data-path on an FPGA. A challenge still remains in controlling and monitoring the simulations, capabilities common in publicly available simulations environments. System Generator provides limited capabilities for interfacing via a co-simulation option. In this mode, dedicated input and output ports are accessible to the Simulink environment via a hardware/software wrapper created by System Generator. In order for states and intermediates to be monitored, System Generator must retain cycle-level control over the simulation, in effect, single-stepping through the simulation to access each value. No buffering is done on-board, greatly limiting the performance of the system.

Alternatively, the FPGA hardware can be executed via a free-running clock. When in

this mode, the FPGA will be at full performance levels, but the software will not be able to keep up with the throughput requirements, dropping data regularly. When the FPGA development board contains analog data converters, inputs/outputs can be transferred to the FPGA at full speed. The slower, register based access to the FPGA via System Generator can be used for modifying parameters on the fly.

Future work is needed to maximize digital transfer rates to and from FPGA models. In the case of our development board, the Virtex-II XtremeDSP II, data must be buffered and transferred via DMA (direct memory access) over the PCI (peripheral component interconnect) interface. In other development boards where a processor is accessible either in the fabric (PowerPC hard core, MicroBlaze soft core, etc.) or external to the FPGA, additional interfaces become reasonable, including USB, Firewire (IEEE 1394), Ethernet, IDE, etc. These are possibilities that could potentially be exploited with future hardware/software co-development work.

4.7.3 FPGA Constraints

Software and hardware implementations of neural models deal with increased complexity in different ways. In a traditional software model, an increase in complexity causes a proportional increase in processing time and memory usage. On an FPGA, an increase in complexity will not cause an increase in processing time if the following conditions are met: The addition to the model can be processed in parallel to the rest of the model (*e.g.* adding another ion channel) and there are sufficient logic resources available to implement the additional complexity. When there are not sufficient resources available for additional parallel data-paths, existing data-paths must be modified to add additional pipeline stages. In this case, processing time and memory usage scales linearly much like software implementations.

Limitations arise when the current FPGA cannot support an increase in the depth of the pipeline. While increasing the depth does not increase the area requirements of the data-paths, it does linearly increase but the number of states, thereby register-constraining the design. When all models are interconnected, for each additional model simulated, additional logic is required allowing for full integration of that model into the system. This can

quickly grow the requirements of the model, limiting the number of simultaneously simulated models. In general, tens to potentially a hundred models are possible to implement using the described techniques in this chapter. However, future work is needed to find the techniques to enable hundreds to thousands of simultaneous systems to run.

CHAPTER V

ASSISTED ENGINEERED MODEL CONSTRUCTION

Preface:

The previous chapter illustrated a generic methodology for constructing neural models. We termed this a manual engineered approach as it described an unassisted procedure for the modeler to design for an FPGA. While this did enable the eventual construction of their neural models, the process was often mired by difficulties. This process was time consuming, error-prone, and made for difficulty when interfacing to a computer. The objective of this work was to design a methodology for rapid construction of relatively complex models. The model chosen as a test-bed was a 40 neuron population emulating the Pre-Bötzinger Complex [18, 19]. This work was a collaboration with Michael S. Reid where my relevant contribution is the assisted-flow design methodology. The methodology was utilized for the construction of the model required by Michael's research. It should be noted that the description of the model in this chapter was written primarily be Michael.

There were three primary goals for the research highlighted in this chapter. First, we wanted to create a semi-automated methodology optimized for models with a high degree of regularity. These models include population and compartmental models. Next, based on our assumption that the majority of model modifications following initial development was of two categories, the alteration of the model size or the tuning of a parameter, we sought a flexible design methodology requiring minimal changes to the model. Finally, we intended to build an interface for the purpose of logging data with minimal or no performance impact.

For the first goal, we built an infrastructure consisting of a database of model parameters and specifics, along with a series of tools to aid in the construction of those portions of the model that are not model-specific. These structures within the model were clearly delineated by the work in the previous chapter. The second goal was intended to aid the modification of models once they were built. This was successfully implemented through

two enhancements, a memory-mapped parameter tuning system for adjusting parameter values on-the-fly and a series of techniques and methodologies incorporated into user tools capable of altering the number of repetitive structures implemented, *e.g.* the number of neurons in a population. Finally, we explored means to transfer data at high-rates from the FPGA for processing on a computer. We settled on an auto-generated system to selectively inspect any predetermined quantity in the model and route that signal to a high-speed analog output. We settled on this approach since a high-performance digital interface was exceedingly difficult to implement.

Overall, this approach was highly successful as we were able to build a 40 neuron model with 1600 synapses and over 2000 parameters. Despite this success, the implementation of this model was very challenging. For example, timing and precision errors consistently stymied the design process. We found that partial automation tools could never fully prevent user mistakes or oversight, which ultimately cause timing errors to emerge. Additionally, as this model was particularly stiff, we gained considerable insight through the process of precision determination. In particular, we learned that precision within approximation functions was often the critical factor in determining whether or not a model would emulate the full range of output behaviors. The considerable difficulty still faced by modelers when implementing their own simulations provided the impetus to continue to work towards a fully automated design methodology, as described in the next chapter.

5.1 Introduction

FPGAs have been previously shown to be a high-performance platform for neural-modeling applications [37, 88, 55, 81]. Given this fact, our recent research has been focused on addressing and reducing the development time. FPGA implementation is not simply an enumeration of the model's equations in a programming language. Instead, a variety of transformations must be made to work within the confines of the FPGA architecture. These transformations include numerical-precision conversion, operation substitutions, and resource-constrained expression folding. Although these architectural confines can be overcome, they nonetheless frustrate and extend the design process and often introduce errors throughout the design flow.

In this chapter, we present a new process utilizing auto-generated scripts and run-time interaction tools to further enhance the use of FPGAs as neural-modeling platforms. This process is demonstrated via the construction of a fully interconnected population [18, 19] of pre-Bötzinger complex (PBC) neuron models, each containing three Hodgkin–Huxley (HH) conductances [40].

5.2 Background

The implementation of a model, using conventional software modeling tools or an FPGA, can be divided into a structural design phase and a parameter tuning phase. For mechanistic models, the structural design phase consists of building components (ion channels, synapses, etc.) that are then combined into a neural-membrane model [43, 39]. This process is repeated, generating multiple neurons and synapses in a population model. Morphological models are structurally grown by linking multiple adjacent membrane models via coupling conductances [72, 85].

Once constructed, the resulting model must be tuned for the desired outputs. Parameter sets can be found by automated search tools or by hand tuning. Often, experimentation on the model requires frequent adjustment of parameter sets. Structural changes, such as insertion or deletion of ionic currents, are less frequent.

First-generation FPGA implementations of neural models [37] were "handcrafted," one-of-a-kind designs that required weeks to months for each design iteration and hours for parameter updates. This generation of models involved direct translation from a Simulink-based continuous-time, variable-time step, floating-point precision model into a System Generator (Xilinx, San Jose, CA) blockset-based discrete-time, fixed-time step, fixed-point precision model. This process was enhanced through pipelining, a methodology by which multiple models can be executed simultaneously utilizing the inherent delays of the system [37]. This model was groundbreaking in that it first introduced FPGAs to mechanistic neural modeling, but it required significant effort for design generation and iterations. However, this approach based directly off the Simulink model was not optimal for hardware generation, utilizing excessive resources for limited performance and offering little flexibility once implemented. Furthermore, this generation limited the complexity of models that could be implemented. Specifically, this design methodology lacked a clear paradigm for model generation and required a new development cycle as the model changed.

Based on the lessons learned from the handcrafted designs, second-generation FPGA implementations first formalized [88] then generalized [89] the design process. This "engineered" approach made a clear distinction between the computational components (the data-path) and memory-based components (the states). In short, for each simulated time step of the model, the data-path computes the next state of each differential equation, the results of which are then stored back into memory (implemented as a shift register). This distinction between the computational and memory components eliminated the need for reworking the data-path when changing, for example, the number of neurons. Furthermore, the memory-based components were developed independently according to a set of rules and heuristics. Rules for choosing operation latency (the number of cycles required for computing a result per operation) and precision were suggested, easing the design process.

Structural modifications with the handcrafted approach required weeks to months compared to days for the engineered approach. While this has been a significant savings in time, further improvements are desirable. Parameter adjustment effort remains somewhat constant, often taking hours for a recompilation of the system. This chapter presents

the third-generation approach to developing FPGA-based neural models. This "assisted" approach offers a number of advantages over the previous architectures, for both the construction of the model and the interaction with the implemented model. This approach consists of design tools that offer the ability to make many structural changes within hours and parameter adjustments on-the-fly.

5.3 Methodology

The methodology we use is described in the following paragraphs. In short, we utilize an off-the-shelf FPGA development board interfaced through Xilinx (San Jose, CA) System Generator, a graphical front-end development environment that is a component of Matlab and Simulink. This chapter explains a series of algorithms implemented as Matlab scripts that partially auto-generate design components.

Specifically, this design flow utilizes System Generator v8.1 within Matlab v7.1. System Generator is a Simulink blockset that provides a set of library blocks directly translatable into hardware constructs. These library blocks include math operations such as addition, subtraction, multiplication, logic operators including multiplexors, and various forms of memory, such as single-bit registers, shift registers, logic-based RAM, and block RAM.

System Generator translates the model into VHDL code and executes the associated Xilinx ISE v8.1i tools. The VHDL code is synthesized into the primitives on the FPGA, then placed, routed, and finally converted into a bitstream. This bitstream is programmed on the FPGA, providing its unique configuration for that particular model. System Generator then constructs a harness for simulating the model within Simulink. This entire process takes tens of minutes to hours, depending on model complexity.

The work described in this chapter utilizes the Xilinx XtremeDSP series of Virtex-II and Virtex-4 development boards. The XtremeDSP development board is a repackaged Nallatech (Glasgow, UK) BenONE PCI carrier board containing a BenADDA module. The module contains the user programmable FPGA, either a Virtex-II XC2V3000 or a Virtex-4 XC4VSX35, dual 105 MSPS analog-to-digital converters, and dual 160 MSPS digital-to-analog converters. Note that these particular development boards and design tools undergo

version changes over time requiring minor revisions in our design flow.

5.3.1 Co-simulation

The simultaneous execution of a model using both Simulink blocks and directly on the FPGA is termed co-simulation. The co-simulation environment is provided as a means for interacting with the model while it is executing. There are two different clocking modes—a full-speed, free-running clock that can be set to a number of standard clock frequencies and a slower, simulator-controlled, single-cycle clock. In general, the free-running clock will provide enhanced performance; extracting data at the maximal data rate via software, however, is difficult. In contrast, having Simulink control the clock provides full interaction and observability albeit at a loss of performance.

Several blocks are available for use with co-simulation. The *Gateway In* and *Gateway Out* blocks provide one input or output, respectively, to the system. In single-cycle mode, the clocks are synchronized such that inputs are immediately available to the system at the next clock cycle and outputs are immediately available without delay. In free-running mode, data is transferred by best effort often with significant loss of data.

With the release of System Generator v7.1, *Shared Memory* blocks were introduced as an improved means of transferring large quantities of data. With *Shared Memory* blocks, a link is generated at run-time between a block of RAM on the FPGA and an associated memory buffer implemented in software. This allows the continuous monitoring and overwriting of values within the memory buffer. Two modes are available—a locked mode allows block transfers to and from the buffers via DMA and a standard mode makes no guarantee with respect to contention. The approach described in this chapter utilizes shared memory in the standard mode for low-speed parameter updates while the FPGA is free-running.

5.3.2 Parameter Database

The assisted approach presented here requires a new abstraction to describe the components and quantities within the model. In general, neural models fit a basic framework consisting of a system of first-order nonlinear differential equations. Each of these equations is made up a state variable (*i.e.*, memory-based component) and the data-path (*i.e.*, computational

component); the data-path is a function of other states, parameters, and global constants within the system. In certain cases, a single differential equation is divided into intermediate calculations whereby a transient quantity is calculated for use in the same time step in a successive state or intermediate calculation.

We have generated a simple database within the Matlab environment to track these identifiers (names of variables in the system) and quantities of interest (parameter values, state initial values, etc.). Four main categories are represented: states, intermediates, parameters, and constants. A variety of information is appended to each entry in the database. This central repository of pertinent model information enables the simplification of the System Generator model. Since parameter values and initial values of states are stored in the database, a particular System Generator model does not have to store these values locally. In effect, the "construction" details are clearly separated from the "model" details. Further gains are made via auto-generation, which is described in Section 5.4.

In addition to parameter values and initial values of states, other model and entry-specific quantities are stored. For example, type information is stored in the form of fixed-point notation, specifying the sign, the number of total bits, and the number of fractional bits. Parameters, in particular, have several flags associated with them:

- *adjustable*: the quantity is adjustable via the parameter subsystem (see Section 5.4.2)
- *visible*: the parameter is adjustable by the user rather than an internal control signal
- *dependent*: the parameter is adjustable via one or more "visible" parameters

These flags allow the definition of multiple classes of parameters of the system, some of which are left hidden from the user. Many parameters, such as a maximal sodium conductance, $\bar{g}_{\mathrm{Na}}$, can be *adjustable* and *visible*, implying that it is a quantity used directly as an operand in the FPGA and is tunable by the user. In contrast, the output subsystem (see Section 5.4.3) uses the parameter subsystem to provide control signals to dictate which variable in the system is routed to an analog output (a feature of the XtremeDSP Development Board series). These control signals are not modeler-tunable parameters, but can

be modified using a special software routine. In this case, the output select parameter will be described in the database as *adjustable* but not *visible*.

The derived parameter is another special case. For example, membrane capacitance, C_{mem}, should be tunable by the user, but since it appears in the denominator of the membrane-voltage equation and is not in a form that can be computed efficiently on an FPGA (would require a reciprocal), that parameter should not be adjustable in the parameter subsystem. Instead, the parameter $1/C_{\text{mem}}$ is a *dependent*, *adjustable*, but not *visible* parameter while C_{mem} is a *visible*, but neither an *adjustable* nor a *dependent* parameter. Full support, in the form of functions for all of these derived parameters, is further enabled by the database.

As shown in previous work [37, 89], models are often pipelined to increase the utilization of an implemented data-path. For example, in a particular model, a 20-stage pipeline for the voltage state corresponded to 20 compartments or 20 neural models. The voltage state in this case can be thought of as a vector quantity consisting of 20 initial values. This work also suggested the use of circular buffers to store parameter values. Our parameter database, however, enables both states and parameters to take on vector quantities to ease the implementation. These circular buffers are preloaded with parameter values offset by addresses based on the total insertion delay in the pipeline. The database stores the appropriate offset value to aid in construction of these buffers.

5.4 Auto-generation of the Infrastructure

By using the parameter database, a large portion of the infrastructure can be readily auto-generated, freeing the modeler to focus on the modeling task at hand. A variety of tools were generated in Matlab focusing on three particular components—the state subsystems, the parameter subsystem, and an output subsystem to interface the analog outputs. These subsystems are created as dynamically linked Simulink subsystems, such that when the database is altered, a simple command will automatically update the library blocks and the affected models.

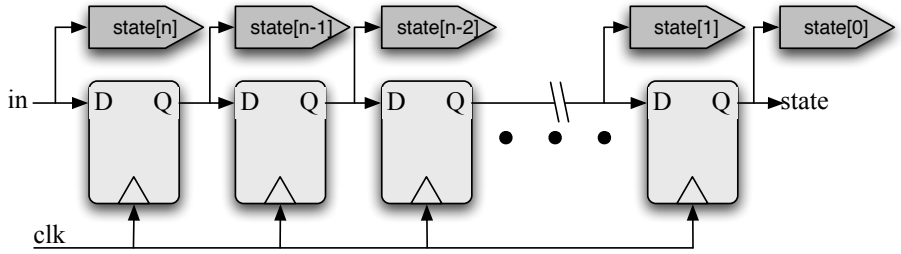


Figure 15: Schematic of the auto-generated state-storage subsystem. The lightly shaded blocks represent a register chain and are built using System Generator *Register* blocks. The darkly shaded blocks represent Simulink *From* blocks which will link with the corresponding Simulink *Goto* blocks in the state read subsystem. For the described $N = 40$ neuron model, there are N registers and $N + 1$ *From* blocks.

5.4.1 State Generation

As described in our previous work [89], multiple ways are possible to design the states and the differential-equation solver depending on the requirements. When multiple models are reusing the same data-path, we suggested to implement the states as a sequence of registers and to add "taps" to connect the registers to the inputs of other portions of the data-path. In an alternative case where the simulation contained only one model of interest, one register is used and clocked at the slower overall sample period. Both of these scenarios are auto-generated by the tools.

The PBC neural population [18, 19] described in this chapter consisted of 40 neurons. Therefore, a 40-stage pipeline was chosen, and all states were defined as vectors of length 40. Auto-generation of the states consisted of the creation of two library blocks per state. One library block is used for the storage of the stages as shown in Figure 15. The state-storage subsystem can optionally include the Euler-integration circuitry or those options can be relegated to the user-defined data-path. The other library block is a compile-time configurable tap into the shift register (see Figure 16).

The state-storage system is simply a shift register of length equal to the number of simultaneous models (40 in this case). A shift register could be made from a single 40-cycle latency *Delay* block. While area efficient, this approach has two drawbacks. First, System Generator does not allow initial values to be stored in the delay block, causing additional initialization logic to be required. Second, internal states are not accessible within a shift

register; data is instead only available at the output of the chain. The solution was to form a shift register based on individual *Register* blocks. These registers can be initialized and allow each output to be accessible. The drawback of this approach is the significant area resources required for each register. For example, a 40-stage pipeline with a 20-bit wide state variable consumes 30 slices using *Delay* blocks and 400 slices using *Register* blocks.

The auto-generated state-read subsystems are designed to tap a particular position in the shift register. Each state-read block is parameterized by an offset value which is chosen to be equivalent to the total delay (latency in clock cycles) from the input, or leaf of the tree, to the output, or root of the tree (see Figure 20 where the h and the V_{mem} state-read blocks have an offset of 3 and 15 cycles, respectively, corresponding to the total delay from the state-read block to the state-storage block). This value is equal to the sum of the latencies of each operation that a particular input must propagate through to reach the output. By carefully designating these delays for each input, multiple models can be simulated in lock step without interference. Note that at each clock cycle, every model that simultaneously uses the data-path is accounted for within the system. Each instance of the model is delayed from the root of the tree, which subsequently decreases per cycle, eventually returning to the end of the queue for the next cycle.

The read subsystem utilizes a multiplexor (see Figure 16), a primitive that chooses the appropriate output according to the select input. This is determined at compile-time, *i.e.* before synthesis. Since the synthesis tool recognizes that the select line of a multiplexor is a constant, the block is reduced to a single bus connecting input with output. Therefore, no resources are used in the instantiation of this block.

System Generator places a hard limit of 32 on the number of inputs to a multiplexor. To accommodate pipelines exceeding 32 stages (for models with over 32 units utilizing the same data-path and up to $32^2 = 1024$ units), a two-level multiplexor scheme is employed. A total of $N_{\text{mux}} = \lceil d/32 \rceil$ first-level multiplexors are generated and fed into an N_{mux}-input multiplexor. The select line of each multiplexor becomes a function of the compile-time parameter dictating the appropriate latency at the leaf of the tree.

This auto-generated state subsystem is a valuable contribution to the modeling process

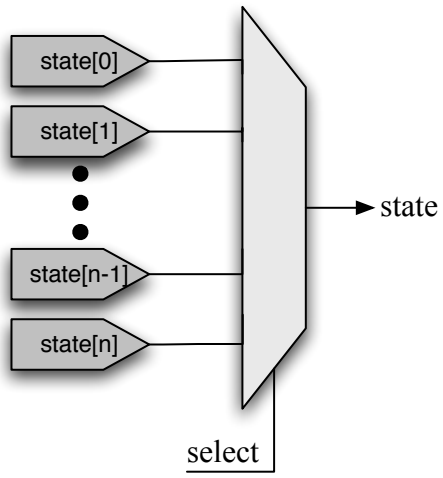


Figure 16: Schematic of the auto-generated state read subsystem. This library subsystem block is unique per state and is instantiated for each read of the state throughout the model. The lightly shaded block represents a System Generator *Mux* block, which selects the appropriate state. The select line of the multiplexor is set at run-time and corresponds to the insertion delay required at the read block. Each darkly shaded block corresponds to a Simulink *From* block which is linked to a *Goto* block in the state-storage subsystem.

as it eases a common class of construction-related model modifications. Often, more or less models are desired in a simulation, *i.e.*, the neuron pool is enlarged as the population size increases, the dendritic tree is enlarged or further subdivided, or a neural circuit model grows in complexity. Assuming the model does not shrink such that the total number of models is less than the maximum latency through the data-path, the number of models can be adjusted via a change in the parameter database and a regeneration of the state subsystems, with no change required to the actual user developed data-path. Additional changes to the parameters will also likely be necessary, but will similarly require little or no user modification to the model.

5.4.2 Parameter Generation

The parameter generation tools provide the full infrastructure to handle on-the-fly parameter tuning within the model. Based on a memory interface, they also allow for optional control registers to set internal states of the model and, for example, allow the model to start, stop, and reset. The parameter subsystem (see Figure 17) utilizes a System Generator *Shared Memory* block to store every quantity requiring dynamic modification.

A memory map is first created constituting each scalar and vector quantity in the system. For the model described in this chapter, the memory map contained 1882 parameters (47 vectors of length 40 plus 2 scalars) for this subsystem. The majority of those parameters, 1600 in total, represented the synaptic weights. Four conductance vectors representing the maximal ionic conductance of a fast inactivating sodium current, a persistent sodium current, a potassium current, and a leak current for all 40 neurons accounted for 160 parameters. Individual leakage reversal potentials and excitatory/inhibitory synaptic reversal potentials per pre-synaptic neuron accounted for 120 parameters. In addition, two internal parameters stored in the memory map were used in the analog output subsystem.

The data-path does not directly read from this large memory, but instead reads from local registers that are constantly updated by this RAM. A counter asynchronous to the data-path continuously cycles through every address in the memory. Simultaneously, a single token is propagated through a circular shift register of length equal to the depth of the RAM. This token becomes an enable for one of two storage elements. In the case of a scalar value, this token feeds the enable of a register that will be updated with the value from the RAM. For a vector, more circuitry is employed to refresh a dual-port RAM with depth equal to the length of the vector (32 in this example). This token will trigger a counter (from 0 to 31) and the write-enable signal of one port of the dual-port RAM. The token will stay active for 32 cycles, sufficient for all values to be updated within the vector RAM.

Since all parameters are not of the same size or type, logic must be added for System Generator to ensure that the correct types are used in synthesis. The *Shared Memory* block (in Figure 17) output type is set as an unsigned integer with width equal to the widest parameter value. Each integer representation, Z is initialized by

$$Z = \begin{cases} \text{round}(x \cdot 2^f), & x \geq 0 \\ \left(! \left(\text{round}((-x) \cdot 2^f)\right) + 1\right) \,\&(2^n - 1), & x < 0 \end{cases} \tag{16}$$

where x is the real-valued number, f is the number of fractional bits, n is the total number

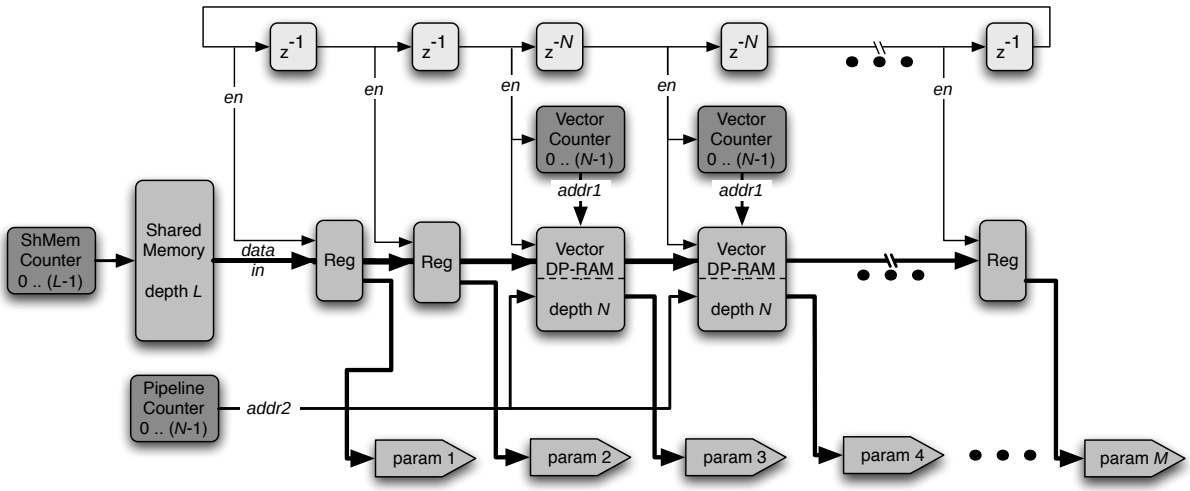


Figure 17: The parameter subsystem. This auto-generated subsystem enables the on-the-fly tuning of scalar and vector parameters via a *Shared Memory* block. A total of L values representing a combination of M scalar and vector parameters are supplied to a N-stage pipeline. The lightly shaded blocks at the top represent the token ring structure consisting of a bit-wide register/delay chain. Delays are either a single cycle for a scalar quantity or a N-cycle delay for a vector quantity. The darkly shaded blocks represent counters in the system. Thin traces represent control signals such as enables to the vector counters and a write enable to the first port of the dual-port vector RAM. Medium traces indicate address busses to each of the RAMs. The thick traces depict those busses containing parameter data.

of bits, and the round function rounds the real number to the closest integer. Negative numbers must be first negated, then converted to an integer, whereby the 2's complement (inversion of all bits marked by ! unary-operator plus 1) is masked (AND binary-operator indicated by &) by a sequence of n-digit binary sequence of 1's. The output must go through a conversion phase before reaching the input port of the register or RAM. This consists of a System Generator *Convert* block changing the type to an unsigned integer at the appropriate number of bits for the parameter, cascaded with a System Generator *Reinterpret* block to set the signedness and the fractional point of the parameter. The *Convert* block can change a value while the *Reinterpret* block can only alter the representation of the signal. In this case, the *Convert* block acts to truncate the unused most significant bits. The auto-generation tools will only place these blocks as necessary for the parameter.

The outputs of this parameter subsystem feed the data-path. For scalar values, a register enables the data to always be available. For vectors, the situation is a little more complex. The other port of each dual-port vector RAM is addressed by a counter equal to the vector, or pipeline, length. This counter is kept synchronized with the sample period of the entire data-path. Since parameters need to be delayed to correspond to their relative insertion points within the data-path, the vectors are initialized and continously refreshed in a sequence that is circularly rotated by the number of cycles equal to the desired offset (see [89] for more on circular buffers supplying parameter values to a pipelined data-path).

The parameter RAM suffers high fanout as it supplies the input of every vector RAM and scalar register in the parameter subsystem. While neither implemented nor required in the past, the high fanout can readily be remedied through the use of a register tree to balance the signal. The initial value of the token can then be changed to align with the delays added to the data signal.

5.4.3 Output Generation

The output subsystem is the third of the three auto-generated components (see Figure 18). This system links the outputs of the data-path with the analog outputs available on the XtremeDSP development board. This subsystem generator overcomes two main limitations

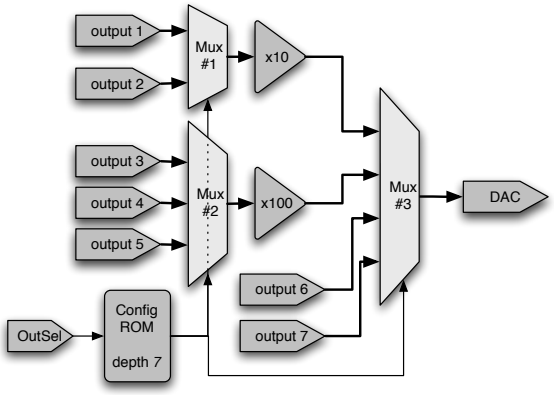


Figure 18: The output selector subsystem. A stereotypical structure is depicted with 7 variables. Outputs 1 and 2 are scaled by a factor of 10, outputs 3–5 are scaled by a factor of 100, and outputs 6 and 7 are unscaled. The OutSel signal, driven by the parameter subsystem, has a range of $0-6$ and sets the address line to the configuration ROM. In this example, the ROM outputs a 5-bit signal, of which one bit is dedicated to Mux #1, two bits are routed to Mux #2, and the remaining two bits connect to Mux #3. Mux #1 and #2 make up the first-level multiplexors, and Mux #3 is the sole second-level multiplexor.

within the included System Generator *DAC* blocks: 1) only two analog output channels are available although more signals are often wished to be viewed, and 2) analog outputs require a signed type of 14 total bits with 13 fractional bits.

Variables are designated and assigned to a particular analog output in the parameter database. The subsystem generator constructs a two-level multiplexing scheme to route the appropriate variable to the analog output. The multiplexors that comprise the first level are separated by type. Since the variable type might not align with the required type (signed, 14 total bits, 13 fractional bits), those variables with the same type are grouped and scaled collectively at the output of the multiplexor. Two scaling modes are supported: a zero hardware power of 2 and a power of 10, which requires the use of multiplier blocks. Each scaled or non-scaled output is then selected via a second-level multiplexor (see Figure 18).

Control logic is required to select the appropriate input per multiplexor. The outputs are enumerated and a control signal specifying the desired variable to be routed to the output is assigned as an input to the system. Two additional *adjustable*, but not *visible* parameters (*DACSelA* and *DACSelB*) are added to the parameter database so that the existing infrastructure can aid in run-time configuration of the output subsystem. The

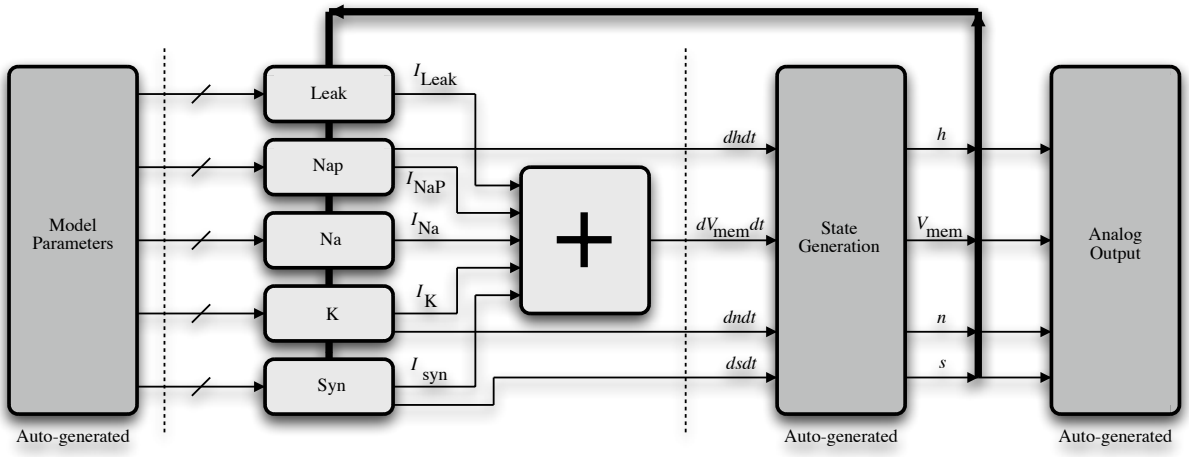


Figure 19: Schematic of the PBC population model generated via the assisted flow methodology. The three darkly shaded block comprise the auto-generated components—the parameter subsystem, the state subsystem, and the analog output selector. The lightly shaded blocks indicate those systems that are manually generated. The dark feedback line indicates state information propagating back into the data-path. Although only one model is depicted, many models are being simulated simultaneously through the pipeline.

output select signal per analog output is used as an address into a configuration ROM (read-only memory), which drives each of the select lines for all of the multiplexors in the subsystem. In particular, System Generator *Slice* blocks are used to separate the appropriate subset of signals within the configuration ROM (see Figure 18) output, which is then routed to the multiplexors.

5.5 Generating the Model

We generated a 40-neuron, fully interconnected model of the PBC as a case study in using the assisted approach to FPGA model development (see Figure 19). In many ways, this model fits within the standard paradigm: a deep pipeline of characteristic, identical components based solely on a system of differential equations. In other ways, this model has required some custom engineered solutions to overcome architectural or systemic limitations. Within Figure 19, those components that are auto-generated are marked as such while all other blocks are manually generated based on our previously published data-path design methodology [89].

5.5.1 Neuron Model

A computational model of the PBC model [18] that we implemented is based on a single-compartment HH formalism [40], and its dynamics are described completely by a set of autonomous differential equations. The transmembrane current, I_{c}, is defined as

$$C_{\mathrm{mem}}\frac{dV_{\mathrm{mem}}}{dt} = -(I_{\mathrm{Leak}} + I_{\mathrm{NaP}} + I_{\mathrm{Na}} + I_{\mathrm{K}} + I_{\mathrm{syn}}) \tag{17}$$

where C_{mem} is the whole cell capacitance (21 pF), V_{mem} is the membrane potential, and t is time. The intrinsic currents include a passive leakage current (I_{Leak}), a persistent Na current with slow inactivation (I_{NaP}), a fast Na current (I_{Na}), and a delayed-rectifier K current (I_{K}). The subthreshold currents are I_{Leak} and I_{NaP}, and the spiking currents are I_{Na} and I_{K}. The extrinsic current is the sum of the synaptic currents (I_{syn}) from the other $N-1$ neurons. This endogenously bursting neuron does not require external stimulation.

Voltage-dependent activation and inactivation variables regulate the conductances of the ionic currents (I_{NaP}, I_{Na}, I_{K}), and the dynamics of a single gating variable, x, is given by

$$\frac{dx}{dt} = (x_\infty - x)/\tau_x, \tag{18}$$

$$x_\infty = \left(1 + \exp^{(\frac{V_{\mathrm{mem}}-\theta_x}{\sigma_x})}\right)^{-1}, \text{ and} \tag{19}$$

$$\tau_x = \bar{\tau}_x / \cosh\left(\frac{V_{\mathrm{mem}} - \theta_x}{2\sigma_x}\right). \tag{20}$$

A sigmoidal, steady-state, voltage-dependent (in)activation function of x with a slope that is inversely proportional to σ_x (also referred to as a slope voltage) and a half-(in)activation at $V_{\mathrm{mem}} = \theta_x$ (also referred to as a half maximal voltage) is given by x_∞. A bell-shaped voltage-dependent time constant that has a maximal value $\bar{\tau}_x$ at $V_{\mathrm{mem}} = \theta_x$ and a half-width determined by σ_x is given by τ_x.

Table 4: Canonical model parameters separated by type—intrinsic currents and gating variables.

Intrinsic Currents		
i	E_i (mV)	$\bar{g}_i$ (nS)
Leak	-65	2.8
NaP	n/a	2.8
Na	50	28.0
K	-85	11.2

Gating Variables			
x	θ_x (mV)	σ_x (mV)	$\bar{\tau}_x$ (ms)
h	-48	6	10,000
n	-29	-4	10
m_∞	-40	-6	n/a
q_∞	-34	-5	n/a

The intrinsic currents, I_i for $i \in \{\text{Leak}, \text{NaP}, \text{Na}, \text{K}\}$, are defined as

$$I_{\text{Leak}} = \bar{g}_{\text{Leak}} \cdot (V_{\text{mem}} - E_{\text{Leak}}) \tag{21}$$

$$I_{\text{NaP}} = \bar{g}_{\text{NaP}} \cdot m_\infty \cdot h \cdot (V_{\text{mem}} - E_{\text{Na}}) \tag{22}$$

$$I_{\text{Na}} = \bar{g}_{\text{Na}} \cdot q_\infty^3 \cdot (1 - n) \cdot (V_{\text{mem}} - E_{\text{Na}}) \tag{23}$$

$$I_{\text{K}} = \bar{g}_{\text{K}} \cdot n^4 \cdot (V_{\text{mem}} - E_{\text{K}}) \tag{24}$$

where $\bar{g}_i$ is the maximal conductance and E_i is the reversal potential. Four gating variables are required (h, n, m_∞, q_∞), and two of them (m_∞, q_∞) activate instantaneously. Note that I_{K} does not have an inactivation term, and the activation of I_{K} and the inactivation of I_{Na} use the same gating variable, n. Table 4 specifies the published, or canonical, model parameters [18]. The following should be noted from the table: (1) The model has a fast time constant (10 ms) and a slow time constant (10 s), which represent two state variables (n and h, respectively). V_{mem} represents a third state variable; (2) Activation (inactivation) is represented by $\sigma_m < 0$ ($\sigma_m > 0$); (3) The two instantaneous activation time constants and the two activation exponents are not considered as model parameters.

The synaptic current *to* neuron j, $I_{\text{syn}}(j)$, *from* the population of N neurons is defined as

$$I_{\text{syn}}(j) = \sum_{i=1}^{N} (\bar{g}_{\text{syn}}(i,j) \cdot s(i)) \cdot (V_{\text{mem}}(j) - E_{\text{syn}}) \tag{25}$$

where $\bar{g}_{\text{syn}}(i,j)$ is the maximal synaptic conductance (values determined experimentally), $s(i)$ is the synaptic gating variable, and E_{syn} is the synaptic reversal potential. The model is a half-center oscillator whereby each half consists of an equal number of neurons. An all-to-all connectivity scheme in the neural population was implemented—all ipsilateral neurons

made excitatory connections ($E_{\text{syn}} = 0$ mV) and all contralateral neurons made inhibitory connections ($E_{\text{syn}} = -80$ mV).

The dynamics of a single synaptic gating variable *from* neuron i, $s(i)$, are defined by the following equations [19]:

$$\frac{ds(i)}{dt} = \begin{cases} [1 - (1 + k_r) \cdot s(i)]/\tau_s & V_{\text{mem}}(i) > \theta_s \\ [-k_r \cdot s(i)]/\tau_s & V_{\text{mem}}(i) \leq \theta_s \end{cases} \tag{26}$$

where $k_r = 1$, $\tau_s = 5$ ms, and $\theta_s = -10$ mV are fixed for all synapses. These equations have been simplified from their original form by making the following assumptions about $s_\infty(i)$: for $V_{\text{mem}}(i) > \theta_s$, the growth rate assumes $s_\infty(i) = 1$, otherwise the decay rate assumes $s_\infty(i) = 0$. Note that $s(i)$ is a piecewise function of $V_{\text{mem}}(i)$ and has N values.

5.5.2 Ion Channel Construction

We used a state-driven, pipelined architecture to implement the FPGA-neuron model [89]. For each of the four state variables (V_{mem}, h, n, s), N pipelined calculations are performed each sample period, where $\tau = 10$ μs is the integration time step (System Generator requires that τ/N be a rational number). The minimum possible N for the network was determined by the latency of the longest algebraic pathway in the pipeline. Figure 20 shows the implementation of the h state variable, where h is defined from Eq. (18), Eq. (19), and Eq. (20). For this example, the longest latency in the pipeline is 15, the algebraic sum of the delays for V_{mem}. The synchronization of these signals throughout the system is critical. Note that only one address needs to be calculated for the two look-up tables, which are required for the nonlinear calculations for h_∞ (shown as hss in the figure) and $\Delta T/\tau_h$ (shown as $\frac{dt}{t_h}$ in the figure). The n gate was similarly designed and implemented.

Six nonlinear calculations from the PBC model required look-up tables—four sigmoidal functions for the steady-state activation and inactivation values (m_∞, h_∞, q_∞, and n_∞) and two hyperbolic cosine functions for the time constant values (τ_h and τ_n).

The inequality of latency within the two look-up tables marks a departure from the standardized model generation introduced in our previous work [89]. In that architecture, we would have duplicated the V_{mem} read-state block with varying offsets and the multiplier

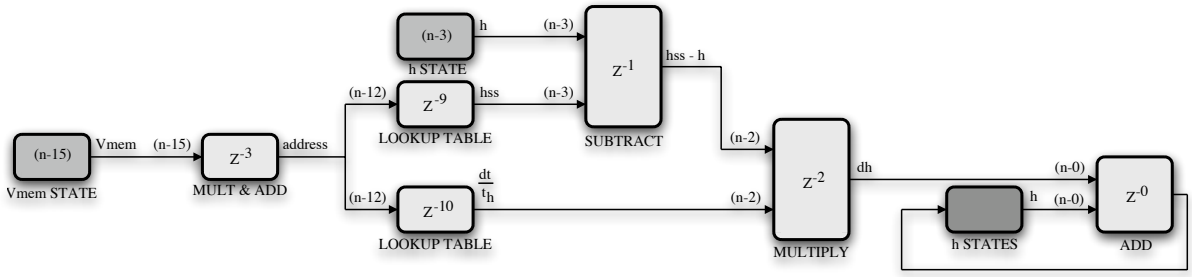


Figure 20: State-driven implementation [89] of the h state variable. The lightly shaded blocks represent the mathematical operations (addition, subtraction, multiplication, and look-up tables), and the pipeline delays are given in these blocks. The two medium-shaded blocks represent the pipelined state variables, V_{mem} and h. The one darkly shaded block represents the state-holding register. The numbers in parenthesis represent the pipeline delay of the signal.

and adder used for address generation. In this case, resources are saved by reusing the multiply and add circuitry and simply adding an additional eight and nine cycles of delay to the h_{ss} and $\frac{dt}{t_h}$ look-up table blocks, respectively, in Figure 20. This combination was used across all the look-up tables in the model.

The number of block RAMs used by one look-up table is given by the following:

$$\#\text{ of block RAMs} = \left\lceil \frac{\text{bits}_{\text{output}}}{18} \right\rceil \times 2^{\text{address\_lines}-10}. \tag{27}$$

As a good tradeoff between quantization error and resource usage, we implemented these tables with 16384 entries (14 address lines) and used 18-bit precision with each output. Therefore, we used 2^4 block RAMs for each look-up table for a total of 96 block RAMs. In addition, the intrinsic model variables that were designated as parameters within the FPGA (E_{Leak}, $\bar{g}_{\text{Leak}}$, $\bar{g}_{\text{NaP}}$, $\bar{g}_{\text{Na}}$, and $\bar{g}_{\text{K}}$) each required one block RAM, and the N^2 maximal synaptic conductances ($\bar{g}_{\text{syn}}(i,j)$) required N block RAMs.

To determine the inputs of the look-up tables, the following quantities were first defined:

$$\phi_x(V_{\text{mem}}) = \frac{V_{\text{mem}} - \theta_x}{\sigma_x} \tag{28a}$$

$$\phi_x(V_{\text{mem}}) = \frac{V_{\text{mem}} - \theta_x}{2\sigma_x} \tag{28b}$$

where Eq. (28a) is the argument of the exponential function in Eq. (19) and Eq. (28b) is the argument of the hyperbolic cosine function in Eq. (20). The range of V_{mem} for all look-up

tables was defined between −70 mV (address 0) and +30 mV (address 16383). The address, as a linear function of V_{mem}, is given by the following:

$$\text{address} = 163.83 \cdot V_{\text{mem}} + 0.7 \cdot 16383 + 0.5. \tag{29}$$

The additional offset of 0.5 was required because the quantization flag of the adder was set to truncate, and the overflow flag was set to saturate. Note that only one address calculation was necessary because all of the six look-up tables required the same address, which saved resources.

Table 5 gives the first and last elements of the arrays of inputs required for each of the look-up tables, and the magnitudes of the step sizes of each of the arrays are given by:

$$|\text{step size}| = \frac{|\phi_x(-70)| + |\phi_x(+30)|}{16383}. \tag{30}$$

In addition, to increase the accuracy of the look-up table for τ_h, the output was first multiplied by 2^{11}, to place the maximum value of the table in the 0.5 to 1.0 range, and later an 11-bit shift operation was used to adjust the result back to the correct value.

Table 5: First and last elements of the look-up tables for the gate kinetics calculations.

	$\phi_x(-70)$	$\phi_x(+30)$
Sigmoidal Functions		
m_∞	5.0	$-11.\overline{6}$
h_∞	$-3.\overline{6}$	13.0
q_∞	7.2	−12.8
n_∞	10.25	−14.75
Hyperbolic Cosine Functions		
τ_h	$-1.8\overline{3}$	6.5
τ_n	5.125	−7.375

We also used Matlab to analyze the quantization error for different sizes of the LUTs. We randomly picked 10^5 values of V_{mem} for $-70 \leq V_{\text{mem}} \leq +30$ and compared the actual value of the sigmoidal or hyperbolic cosine function to the quantized FPGA value and recorded the mean errors for each of the LUTs in Table 6. As expected, a doubling of the size of the look-up table roughly corresponded to a fifty percent decrease in the mean quantization error in the table.

Table 6: FPGA-Neuron Model—mean quantization errors for different look-up table sizes.

Look-Up Table Inputs	Mean Quantization Error
2^{10}	$20e^{-5}$
2^{11}	$10e^{-5}$
2^{12}	$6e^{-5}$
2^{13}	$3e^{-5}$
2^{14}	$2e^{-5}$

5.5.3 Synapses

Because of controllable maximal conductances (including synaptic weights), only one FPGA architecture was required to be implemented, and this flexible design resulted in the maximum number of possible neural connectivity configurations for a given number of neurons—an all-to-all connectivity scheme was implemented resulting in N^2 connections. Note that the N synaptic weights that corresponded to the connections *from* and *to* the same neuron were always set to 0 nS. Since $N^2 >> N$, the number of connections in our network was a much more important factor than the number of neurons. The synaptic weights were independent of the spiking frequency and were not implemented with distant-dependent delays (*i.e.*, long-distance connections were not be applicable).

The hierarchical synaptic network that implements Eq. (25) is shown in Figure 21. For clarity, the figure is only shown for $N = 8$, but the synaptic network is easily scalable. Each $\bar{g}_{\text{syn}}$ parameter contains N values; for example, the second element of $\bar{g}_{\text{syn}}(1)$ is the weight of the synapse *from* Neuron 1 *to* Neuron 2. Two E_{syn} parameters are required—one implements inhibitory connections and one implements excitatory connections. One of the N registers is updated every cycle with a new value of the s state variable and is held constant for N cycles; the counter, relational, register, and constant blocks implement this scheme to update the state variable on the proper line. Note that an encoder is effectively created from the constant and relational blocks in the dashed region.

5.6 Results

The above model equations were implemented as the data-path in the full-model implementation (see Figure 19). The data-path was made up of a small number of System Generator

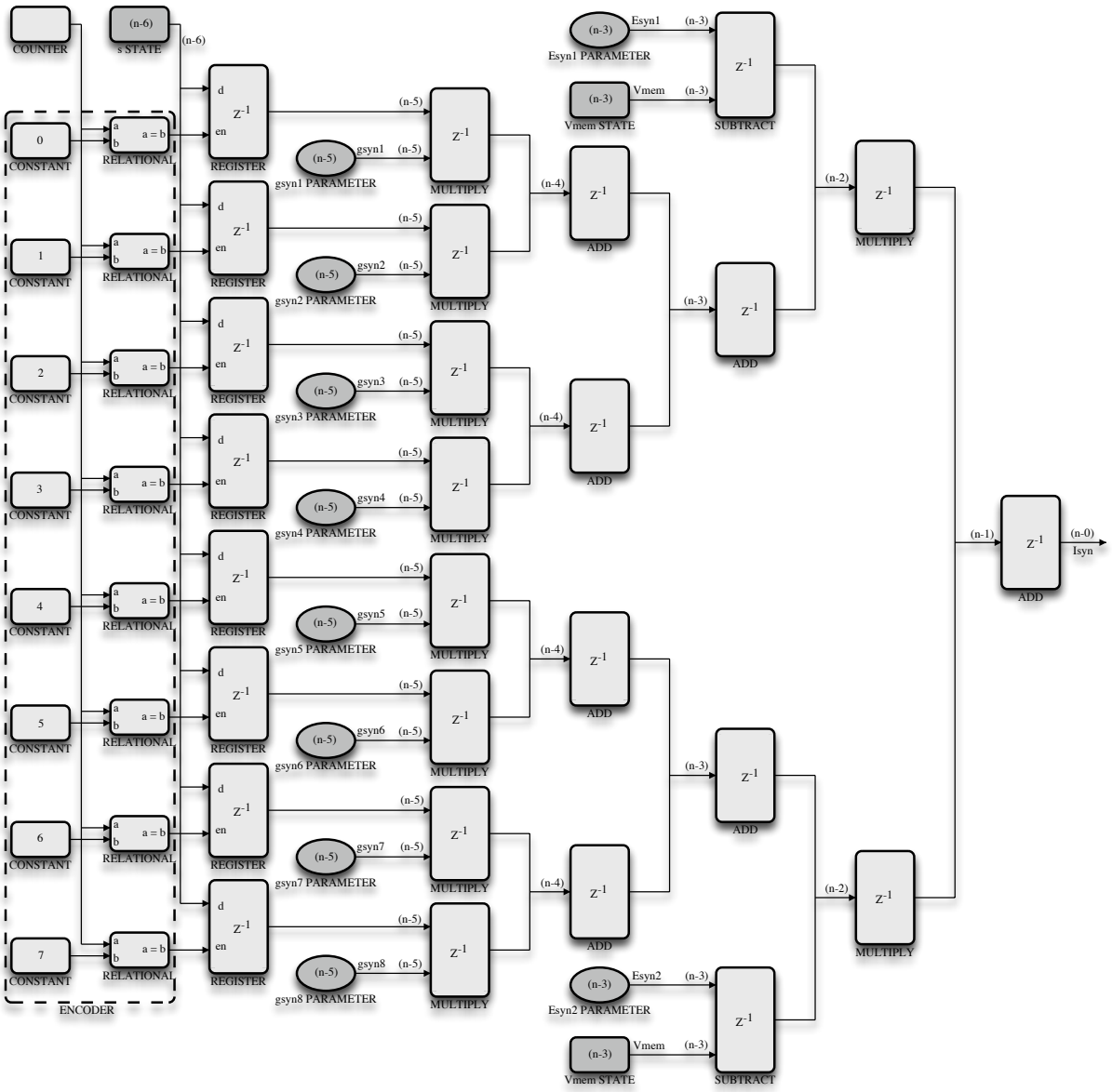


Figure 21: Implementation of synaptic current summing network. The lightly shaded blocks represent the mathematical and data operations (addition, subtraction, multiplication, counter, relational, register, and constant), and the pipeline delays are given in these blocks. The three medium-shaded blocks represent the pipelined state variables, V_{mem} and s. The medium-shaded ovals represent the pipelined variables, $\bar{g}_{\text{syn}}$ and E_{syn}. The numbers in parenthesis represent the pipeline delay of the signal.

building blocks—arithmetic blocks such as adders, subtractors, and multipliers, memory blocks (*ROM* block configured to use BlockRAMs) for the construction of look-up tables, relational and conditional blocks for implementation of piece-wise functions. Additional blocks were utilized sparingly for custom design requirements, such as *Register*, *Delay*, and *Counter* blocks.

The parameter subsystem and state subsystems were auto-generated from a description of the model in the described parameter database (see Section 5.3.2) and was used unmodified. The output-select subsystem was auto-generated, but since the model did not require the use of analog outputs, it was manually modified for digital transfer via co-simulation. The output subsystem instead down-sampled the V_{mem} states by a factor of 10 and drove a time-division multiplexed signal through a System Generator *Gateway Out* block. Post synthesis, each neuron's voltage output was saved directly to a Matlab binary data file for post-processing.

The performance on the PBC model described is summarized in Table 7. The 40-neuron population with its all-to-all synaptic connections was analyzed with respect to performance and resource utilization. The performance numbers provided are based on post place-and-route targeting the Virtex4 XC4VSX35-fg676-10 device. The tools were allowed to auto-constrain with respect to clock frequency. The resulting critical path time, T_c (maximum propagation from one clock edge to the next clock edge taking into account clock skew and jitter), was used as the basis for the maximal performance numbers in Table 7. These performance numbers are meant as upper limits to what is possible. If the analog outputs on the XtremeDSP-IV Development Board are utilized, performance approaches the theoretical limit, dependent on the generation of the exact required clock frequency. If co-simulation is used, performance is degraded substantially from the theoretical. Nonetheless, both modes are a substantial improvement over software-only implementations.

In Table 7, T_s, the iteration period, is computed by

$$T_s = N \cdot T_c \tag{31}$$

for an N-neuron size population. By factoring in the Euler-integration time-step, τ, a

Table 7: Performance and resource-usage results for a varying population size.

# of neurons (N)	Performance		Resources	
	T_s	RT_{factor}	Slices	Multipliers
16	0.38 μs	26.4	8,327 (54%)	111 (58%)
24	0.68 μs	14.8	10,374 (68%)	135 (70%)
32	1.01 μs	9.9	12,052 (78%)	159 (83%)
40	1.15 μs	8.7	13,840 (90%)	183 (95%)

real-time factor can be established as

$$RT_{\text{factor}} = \frac{\tau}{T_s}. \tag{32}$$

This real-time factor is theoretical and implies that the system clock is given by

$$f_{clk} = \frac{1}{T_c}. \tag{33}$$

The actual real-time factor is a function of the available system clock frequencies for the FPGA.

The resource utilization counts in Table 7 were determined from the mapping report in the standard Xilinx implementation flow. The total counts included only the model and not the co-simulation infrastructure. The resource utilized for the infrastructure was minimal compared to the model definition. The FPGA contained 15,360 slices and 192 DSP blocks (each DSP block contains a multiply–accumulate circuit).

Varying the number of models simulated required a number of minor changes to the design. The parameter database was adjusted such that vectors of conductances reflected the increasing number of neurons in the simulation. Additional vectors describing the synaptic activity were required for each incremental neuron added. Finally, the hierarchical synaptic adder tree was modified to handle the increased number of synapses. Further scalability was limited by available FPGA resources. Deviations from this architecture, such as the use of BlockRAMs for state information or reductions in precision throughout the model, might have increased the total number of implemented models.

A typical neuronal output gathered via co-simulation is depicted in Figure 22. The time-step, τ, used in simulation was 0.01 ms. The output was downsampled by a factor of

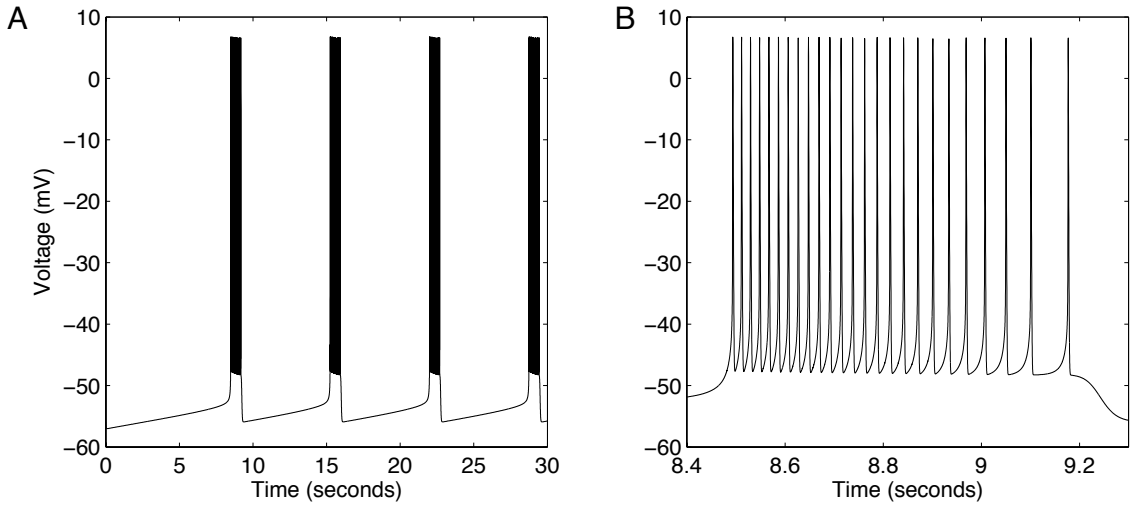


Figure 22: Output data from one of the 40 neurons within the PBC population model. Pane A depicts a 30-second simulation of characteristic bursting. Pane B shows a close-up view of the first of four bursts. This model was simulated on the FPGA in a single-cycle co-simulation mode with a 0.01 ms time step and downsampled by a factor of 10. A System Generator *Gateway Out* block was utilized to save the data directly into a file. All 40 neurons were captured in this one simulation.

10, for an output rate of 10 kHz. Typical simulations were run for 30 seconds producing $N \cdot 300,000$ data points for an N-neuron population.

5.7 Conclusion

The results of this study confirm that the construction of significantly complex HH conductance-based neural models, including population models, is well within the capabilities of existing FPGA design techniques. Furthermore, these designs can be readily modified in a reasonable time frame, and parameters can be adjusted on-the-fly. More complex models with increased numbers of conductances and varied kinetics can be implemented in this architecture with a possible reduction in the total number of neurons.

The success of any developmental effort is often linked to the capabilities of the available design tools. This belief continues to drive the construction of tools that aid in auto-generation. Inevitably, auto-generation tools are limited when a model has unforeseen characteristics. In particular, the PBC model implemented offered unique challenges such as precision issues, synaptic interconnections, and data throughput requirements. All of these limitations were overcome and the auto-generation tools were enhanced where possible.

By spending additional effort initially, neural modelers can utilize FPGAs for high-performance neural modeling. With the use of these auto-generated sub-components, construction-based changes are easily accomplished requiring just hours of effort. The PBC model in particular was developed initially with 16 interconnected models but was increased easily to 40 interconnected models. Second, the parameter updating system allowed for the running model to very quickly change parameters and re-run simulations, enabling a very rapid sweep of behavior over the parameter space.

CHAPTER VI

FULLY-AUTOMATED MODEL CONSTRUCTION

Preface:

The previous three chapters described a progression from a fully manual procedure for implementing neural models using digital design methods to an assisted flow technique where user tools help to not only build a model but also a simulation engine. The experience and lessons learned from these sections played a large role in initiating the research behind the efforts described in this chapter. Here we detail the fully-automated design flow based around the DYNAMO compiler. Much of the development of DYNAMO has been done in collaboration with Christopher T. Church. Those sections that he was primarily responsible for are labeled as such in the text.

The work described in this chapter was guided by three overriding goals. First, we wanted to enable modelers to utilize FPGAs for their neural simulations without any particular knowledge of FPGAs required. Second, we were interested in building a universal compiler capable of arbitrary model descriptions, for example, without any of the regularity requirements as seen in the previous chapter. Finally, we strove to design this fully-automated flow with high-performance at the forefront.

To achieve the first goal, we designed a custom modeling language with specific syntax useful for describing neural equations via systems of differential equations. This language was coupled to a hardware compiler through a simple programming language or intermediate representation. The full conversion from model description to hardware execution engine is fully automated, freeing the modeler from directly developing on the FPGA. Next, through the use of the generic modeling language and the intermediate representation, models of arbitrary structure can be described and passed from the front-end language to the back-end hardware compiler. Finally, while the performance of a completely auto-generated flow may

not always exceed that of a manually-tuned model, the DYNAMO compiler produced performance improvements exceeding 20× the performance achievable from a software compiler.

The DYNAMO compiler, now in its third major revision since its inception, has constantly been in a state of flux. New techniques, heuristics, methodologies, and features have consistently been added to extend the range of translatable models, improve the performance, and enhance the overall experience of the modeler. At the time of this writing, the usability, flexibility, and performance goals have all been achieved.

6.1 Introduction

We have made considerable progress in enabling FPGA technology for neural modeling applications. FPGAs have already been shown to be a high-performance, functional platform for generic neural models [37]. This performance advantage came at a cost—a complex, error-prone design flow requiring months of effort for successful design realization. This difficulty provided the inspiration to develop a series of methods and techniques to aid in the model construction process, reducing the time of realization from months to weeks to days to hours.

We have taken a multi-step methodical approach to streamlining the FPGA design process. First, we have developed an engineered approach to the manual construction of an FPGA model equation by equation, piece by piece. That work is summarized in Chapter 4 and in print [89]. Next, we utilized the experience gained from manual construction of models to create a set of tools to automate the development of repetitive constructs. This included the integration of the state variables and infrastructure corresponding to tuning parameters and data-transfer. This design methodology, termed assisted-engineered flow, reduced the time for construction of models to weeks, structural changes to hours and tuning of parameters to seconds as summarized in Chapter 5.

While the assisted-engineered flow greatly improved usability of FPGAs for the neural modeler, it did not isolate the modeler from the intricacies and specifics of digital system design on FPGAs. Instead, it required the modeler to be comfortable with the fixed-point number system, synchronous timing, and digital logic. Thus, the need arose to develop a fully automated approach to the development of neural models on FPGAs. Such a system would work in the domain of the neural modeler eliminating or drastically reducing the requirement for background knowledge in digital development on FPGAs.

This chapter details the efforts and successes to date of a fully automated approach to model development. Through my efforts along with Christopher T. Church, M.S. (2006), we have built the DYNAMO compiler, a tool for the compilation of model descriptions into a tightly coupled, synchronous execution loop for direct synthesis on an FPGA. DYNAMO has been built to be as generic as possible with minimal user intervention to be powerful,

flexible, and easy to use. Limitations in the compiler, design choices, and future efforts will be explained throughout this chapter.

6.2 Compiler Design

The DYNAMO compiler takes a traditional approach to compiler architecture. Suppose that a software compiler was to be written for m multiple languages (Java, C++, SML, etc.) and targeting n multiple architectures (Intel, PowerPC, ARM, etc.). One solution could be to write $m * n$ compilers for each possibility. This would be time consuming and inefficient. Instead, each of the languages can be sub-compiled into an intermediate representation (IR), each becoming a *front-end*. Similarly, each of the n targets can be written from the intermediate representation, or a *back-end*. This will require $m + n$ unique programs, a drastic reduction from the $m * n$ number when using multiple languages and multiple targets. Two passes would be required to convert a particular language to a particular target through the IR. This total reduction of required software programs requires careful planning and development of a complete and unambiguous IR.

We chose to develop DYNAMO as a two-pass compiler to create a more scalable long-term development platform. This approach has allowed us leeway to experiment with different language features and develop multiple back-ends, including software back-ends targeting C and MATLAB. The intermediate representation chosen is based on lambda-calculus, a functional, Turing-complete language representation well suited for mathematical evaluation [49, 26, 22]. Turing-complete implies that the lambda-calculus can describe any computable function.

The compiler was initially based off the MRCI system [65], a dynamical system compiler targeting dynamic-clamp applications. Following a full-rewrite, the second generation compiler, dubbed DYNAMO, first delineated the front-end and the back-end, with the lambda-calculus IR. An additional full-rewrite of the compiler produced the present day DYNAMO modeling language (DML) and its current feature set including modularity and vectorization. While possible to implement in the previous lambda-calculus, the current IR was optimized for that purpose, increasing its suitability for modeling applications. The

implementation of the IR was accomplished in its entirety by Christopher T. Church, but is included in this document for completeness.

6.2.1 Front-end

The front-end language, or DML, is a functional and modular language for describing both the equations representing the mechanisms of the model and the construction of the model from the underlying mechanisms. The DML was not designed specifically for neural modeling, despite the intended purpose. This was deliberate as to force the general problem to the forefront over the specific relating to neural models. We decided on this path to provide the most flexibility for implementing future models with likely dissimilar characteristics. As a result, there are no specific built-in constructions in the language relating to neural modeling. Nonetheless, an extensive user-defined library can provide specific functions and constructions for a particular modeling application.

Because flexibility was the primary design goal, less attention was placed on making the language easy to use. While this is a current, minor shortcoming, we have been proposing to enable application specific languages as a subset of the full DML, aiding the modeler in their particular domain. This enhancement would fit well within the two-pass compiler architecture as a new specific modeling language will not affect any of the back-end features. More on this and other future directions can be found in the Discussion in Chapter 8.

The DML features a variety of constructs present in modern programming languages and some powerful features rarely available in traditional languages. The common features include modularity, objects, library linking, and unit testing. More advanced features include a fully functional paradigm, built-in primitives for specifying first-order differential equations, and a type system designed for dynamical system modeling. Standard datatypes include states, parameters, intermediates, constants, inputs, and outputs. Each of these features are described by example in the following subsections.

6.2.1.1 Modeling-specific Constructs

Unique to Dynamo, program flow is implicitly defined to occur within a loop, mimicking the standard iterative approach to numerically solving differential equations. The computations

performed on each execution loop must be identical, enabling the construction of a single computation engine that can simulate any iteration. In the case of an FPGA back-end, this requirement forces each iteration to take an equal amount of execution time, maintaining a fully synchronous implementation (see Section 6.6). Future versions might allow for multiple execution engines, controlled by a state-machine.

While a standard programming language might contain such data-types as integers, booleans, and strings, the DML instead defines data-types specific to dynamical system modeling. The central data-type is the `STATE`. Identifiers marked as a `STATE` are stored explicitly each iteration. For example, a typical model might include a membrane potential, V_{mem}, which will be integrated each iteration. V_{mem} is therefore defined as a `STATE`. A `STATE` does not need to be part of a differential equation. For a value to be saved and output to the user, it must be declared as a `STATE`. Otherwise, algebraic optimizations might hide the computation of the value within other calculations (see Section 6.3). States must have an initial value defined when used within an integration algorithm.

For quantities that the user wishes to be tunable at run-time, the `PARAMETER` data-type provides for user-adjustable quantities in the model. For example, ionic maximal conductances are often defined as `PARAMETER` data-types, but any quantity in the system can be made tunable. Setting each quantity as a `PARAMETER` comes at a cost as additional hardware is required to implement each parameter. For those quantities that do require user-adjustment, the `PARAMETER` data-type provides a useful construct to the modeler.

When quantities do not require run-time tuning, the value can either be explicitly defined within an equation or specified with the `CONSTANT` data-type. The following specification of a sodium current, I_{Na}, will be handled identically in the compiler:

```
CONSTANT E_Na = 55;
I_Na = g_Na * (V_mem - E_Na);
```

or

```
I_Na = g_Na * (V_mem - 55);
```

In the above example, I_{Na}, is not an explicitly declared data-type. Instead, it is implicitly

defined as an intermediate result, with its name lost in the translation into the IR. By removing the variable name, the compiler is free to perform algebraic optimizations around this calculation.

The DML defines `INPUT` and `OUTPUT` for streaming run-time inputs into the system and outputs from the system, respectively. Inputs can include current injection, voltage commands, synaptic input trains, or any other continuous sampled sequence of values. Similarly, outputs often include membrane potentials but can also include any other value in the system for debugging purposes. Inputs are declared explicitly by name, while wildcards (* matches 0 or more characters, ? matches one character) are enabled to select multiple outputs. Since identifiers are not explicitly written but instead are evaluated based on the wildcard symbols, the `OUTPUT` keyword requires all variables be enclosed in quotes as shown below:

```
INPUT Iapp;
OUTPUT "Vm", "m", "h", "n"; // explicitly matches these outputs
OUTPUT "Neuron*.V?"; // matches Neuron1.Vs, Neuron15.Vd, but not Neuron.Vmem
```

Parameters and states have to be explicitly declared in the system for DYNAMO to provide additional compilation specific information. Each variable labeled as `PARAMETER` and `STATE` must include an initial value and a range. A initial value for a `STATE` is required for every variable labeled as such, but is used in simulation only for numerically computing differential equations. Parameters labeled with the `PARAMETER` keyword additionally require initial values.

States, parameters, inputs (specified by the `INPUT` keyword), and literals (includes values and constants) are termed *read* nodes of the system. Every computed quantity in the system is a function of the *read* nodes of the system. If each equation is represented by a tree where the new computed variable, or the root, is a *write* node, the *read* nodes are the leaves. In order to determine fixed point precisions (see Section 6.4.2), we propagate user-defined range and step information from the *read* nodes throughout the expression tree representing the computation. The range is expressed as a high and low value representing

the possible ranges of the evolution of a state, the tuning range of a parameter, and the span of an input. The step value corresponds to the granularity for tuning and is related to the relative level of precision required for the quantity.

```
PARAMETER gNA = 120 (0 to 120 by 0.2);
STATE Vmem = -65 (-90 to 60 by 0.001);
INPUT IappS (0 to 30 by 0.1),
      IappD (0 to 30 by 0.1);
```

Differential equations are specified with unique syntax, the function `d()`, to indicate the use of an implicit numerical solver. The particulars of the integration are specified by a reserved function name, `integrate`. Our tests have been limited to forward-Euler integration, but the system is intrinsically not tied to only that method.

```
FUN euler_integrate (dt, t, state, eq) =
  state + dt * eq(t);
FUN integrate (dt, t, state, eq) =
  euler_integrate (dt, t, state, eq);
STATE m = 0.1 (0 to 1 by 0.001);
d(m) = (m_inf - m) / m_tau;
```

In the above example, the time step is specified by defining `dt` as a `PARAMETER` or a `CONSTANT`. As a `PARAMETER`, it can be changed as any other parameter in the system. At this point, a true variable-time step solver is not possible (see Section 6.9 for more information). All differential equations in a system must use the same integration function and an identical time-step.

6.2.1.2 Modularity

The DML provides multiple means for developing a modular model description. At the lowest level, implicitly declared intermediate calculations enable long equations to be broken up into its representative parts. Functions are available in the DML for expressions that are reused often. An example follows where a power function is mapped from a binary

function to cascaded unary operations. This illustrates how functions can work to increase the functionality of the language.

```
FUN pow (x, y) = EXP( y * LN( x ) );
sqrt_x = pow(x, 1/2); // example of the pow function
```

Functions are considered first-order quantities in this language. A first-order quantity can be manipulated in the same fashion as a variable. It can be duplicated, passed as a parameter, or redefined. This is a common feature of functional languages, such as SML, Lisp, or mathematical languages such as Mathematica (Wolfram Reasearch, Champaign, IL).

For the simplest neural models, an enumeration of the equations is generally sufficient. A simple model such as the Fitzhugh-Nagumo two-state oscillator [32, 58] can easily be expressed in one module. At the other extreme, a multi-conductance, multi-compartment, neural population model might need many levels of modularity to remain tractable to the modeler. In this example, the population can be a module instantiating neural modules, whereby each instantiates neural compartment models, down to the modules describing the mechanisms present in each ionic conductance. The following illustrates the syntax for instantiating a module, termed a `SYSTEM`.

```
% Instantiation of my_neuron as a Fitzhugh-Nagumo model
SYSTEM my_neuron = new fn(integrate, dt, Iapp);


% Definition of another system - the Fitzhugh-Nagumo model
DEFSYSTEM fn(CONSTANT integrate, CONSTANT dt, DYNAMIC I)


  FUN cube (x) = x * x * x;


  PARAMETER b0 (1 TO 4 BY 0.01) = 2,
            b1 (1 TO 4 BY 0.01) = 1.5,
            e  (0.01 TO 0.3 BY 0.01) = 0.1;
```

```
  STATE u (-4 TO 4 BY 0.001) = 1;
  STATE w (-4 TO 4 BY 0.001) = 1;

  d(u) = u - cube(u) / 3 - w + I;
  d(w) = e * (b0 + b1*u - w);
ENDSYSTEM fn;
```

In the previous example, `my_neuron`, was instantiated with three parameters. The first two are required for the numerical solver, where `integration` is the function describing the integration method and `dt` is the time-step. An additional parameter is passed to the new system, `Iapp`, which can be a constant, parameter, state, input, an intermediate calculation, or even a function. The Fitzhugh-Nagumo system definition is the model that would be instantiated by the `SYSTEM` command above. The parameters of `fn` are labeled `CONSTANT` if the quantity does not change and `DYNAMIC` if it can change, such as the input current. The current need not have the same name in the system definition and the system instantiation—it is instead matched by parameter order. At the scope of `my_neuron`, each state can be referenced by `my_neuron.u` and `my_neuron.w`.

```
DEFSYSTEM hh_list (CONSTANT integrate, CONSTANT dt, CONSTANT num_neurons)
  STATE placeholder (0 TO 100 BY 0.0001) = 0;
  FUN hh_current i = i/num_neurons * 20; % Current range from 0 to 20 nA
  SYSTEM neurons = for 1 num_neurons 1
                       (HH(i) = new hh [integrate, dt, hh_current(i)]);
ENDSYSTEM hh_list;
```

This modular structure is not only helpful for encapsulating parts of the design. It can additionally be used to aid in construction. The previous code segment describes the construction of an arbitrary number of Hodgkin-Huxley models [40]. This number is determined at compile-time and set as `num_neurons`. The `for` function returns a list of neurons indexed from 1 to `num_neurons` by 1 with an current that is a function of its index.

Whitespace is ignored in the DML, so even though the system instantiation of `neurons` is over two lines, it is treated as one statement.

This example further shows the inherent vectorization built into the DML. Quantities are not limited to scalars. Instead, vectors or lists of elements can be manipulated in a seamless fashion. This makes the arbitrary construction of n neurons or m compartments simply within the DML.

6.2.1.3 DML Library

A large effort was made by the DYNAMO development team to create a powerful and flexible modeling language. This enabled many new features to be incorporated into the language through a library, rather then in the compiler itself. We have concentrated many of these functions into a library that is always interpreted prior to parsing the DML model description file. These functions are all written in DML and represent many common tasks in model building or general programming. For example, numerous functions are available for list manipulation, such as *head*, *tail*, and *length*. Standard functional language constructs such as *map* (which applies a function to each element of a list, returning a list), *foldl* (which applies a function iteratively across a list, returning a scalar result), and *tabulate* (which with a parameter n, generates a list from 0 to $n-1$). Users are fully empowered to generate their own functions based on the powerful included DML primitives.

6.2.1.4 Unit Testing

As models get larger, as the number of conductances grow or the size of the population increases, there is a need to test and qualify the model. We developed a unit testing paradigm for the DYNAMO compiler to verify each system. A DYNAMO model can be split into many files, one for each system declaration (`DEFSYSTEM`). The `IMPORT` statement can be used to inline include additional files. Each `DEFSYSTEM` can be tested individually if the file contains a `MAIN` code segment. The `MAIN` code is similar to a *main* function in C or in Java. All `MAIN` routines except in the top-most DML file are ignored during compile-time.

6.2.2 Intermediate Lambda-Calculus

The DML is compiled first into an intermediate representation before the translation into the backend begins. We chose a lambda-calculus representation, which has a number of advantages. First, it is the basis for functional programming which in turn forms the basis for common mathematical expressions and relationships. For example, in a Hodgkin–Huxley model, the membrane potential, V_{mem}, is generally studied as a voltage trajectory, $V_{\text{mem}}(t)$, a functional representation. A numerical approximation of a differential equation is similarly represented as the next solution as a function of the previous solution and the independent variable. Lambda-calculus basic unit is a function and therefore represents mathematical functions simply and elegantly. Second, lambda-calculus can be unrolled readily into expression trees with straightforward algorithms, where an expression trees is rooted in the output and has leaves for all the parameters, constants, or other inputs. Other representations can be converted into trees, but lambda-calculus does it with simplicity and ease. Next, lambda-calculus is a general-purpose language and is Turing-complete, implying any computable operation can be encoded in lambda-calculus. Finally, it is a canonical, simple representation. If a compiler should have the shape of an hour glass, where the front-end is complex, the back-end is complex, the connection should be basic and easily understandable.

Other mathematical environments such as MATLAB and Mathematica (Wolfram Research, Champaign, IL) utilize functional paradigms. In MATLAB, function handles enable functions to be first-class operators. Additionally, the `@` operator allows for anonymous functions, the basis of lambda-calculus. Mathematica offers the `&` function operator with the `#` symbol to represent parameters. A full suite of common functional constructs such as *map* and *apply* hint at functional and lambda-calculus underpinnings.

The lambda-calculus IR produces a series of three tree structures representing a flattened model description, a type table identifying each identifier, and a listing of all inputs and outputs of the system. The three trees generated by the IR are the run tree, the parameter tree, and the state initialization tree. In this data structure, the root of the tree is a *write* node and the leaves are all *read* nodes.

The run tree is the primary tree for the simulation and includes the calculations required for each iteration. No recurrent connections are allowed as this would limit the ability of the simulation to be bounded in time. The *write* nodes are the states that are written on each iteration. The *read* nodes are inputs, parameters, constants/literals, or states from the previous iteration.

The parameter and the state initialization trees represent the expressions for the initial values of the parameters and states, respectively. In the example below, E_{Na} is defined to be the literal 55. In the parameter tree, this is simply represented as a *read* node, the literal, feeding a *write* node, the parameter, `ENa`. The following example shows how constants, R, T, n, and F, can be evaluated to return the parameter NernstConstant. The algebraic simplification to reduce this to a single literal is not performed in the IR, so instead the parameter tree includes the entire expression. The final example shows how the state initialization tree would include the `Vmem` as a *write* node fed by the constant evaluated to be -65 as the *read* node.

```
PARAMETER ENa (45 TO 65 BY 1) = 55;
PARAMETER NernstConstant = (R * T) / (n * F);
CONSTANT Vrest = -65;
STATE Vmem (-90 TO 60 by 0.001) = Vrest;
```

The IR produced type table provides a means for the system to classify each identifier as a state, constant, input, parameter, or derived parameter (see Section 6.3.3). A enumeration of the inputs and outputs of the system complete the model specific information sent to the back-end. Additional information such as command-line arguments and sampling rates are passed to the back-end for varied use depending on the target.

6.2.3 Back-end

The back-end of the DYNAMO compiler refers to all operations and manipulations from the tree output of the lambda-calculus IR to the ultimate generation of the simulation-ready output. There are multiple back-ends developed in DYNAMO. They include the FPGA back-end targeting System Generator, a MATLAB backend that supports fixed-point and

floating-point simulations, and a C based back-end also supporting fixed-point and floating-point simulations. While the original intent of DYNAMO was to produce an FPGA compiler, it has become useful to maintain additional software back-ends for testing purposes and for testing smaller neural models. Additionally, the capability of a floating-point back-end provides a useful comparison verifying fitness compared to the fixed-point back-end on the FPGA.

The back-end contains numerous components, each of which will be explained in further detail below. These components fit into various categories. General optimizations are utilized for all back-ends and include algebraic optimizations, pruning unused logic, and eliminating redundancy in the tree and graph representation. These optimizations promise performance increases independent of the desired back-end.

When computing on an FPGA, precision analysis is performed to estimate fixed-point precisions per operation. A timing analysis is performed to estimate latencies through operations as a function of precision. Finally, lookup tables are generated for all those functions that are not directly or efficiently computable using the available FPGA blocks.

Software back-ends can then convert the manipulated and annotated graphs into code for execution in MATLAB or C. The software back-ends can either evaluate with or without fixed-point precision and/or lookup tables. Timing information is not simulated in the software backends, *i.e.* cycle-accurate simulations are not performed.

Specific hardware back-end manipulations are then performed including hardware resource estimation and allocation, resource scheduling, area estimation and resource-usage refinement, and a variety of schedule optimizations. The resulting accepted schedule is netlisted into a generic, internal netlist format. The infrastructure allowing tunable parameters, outputs, inputs, and flow control of the model is auto-generated and netlisted as well. The top-level netlist containing the full model and infrastructure is then mapped to System Generator blocks and a run script is generated to programmatically build the design and compile it in System Generator and Simulink.

Any change in the front-end language, assuming it can pass through the IR to produce the set of input graphs for the back-end, will require no changes to any of the back-ends.

This creates a very modular and easily maintainable code-base for the future. While some revisions of the language require propagated changes to the entire system, such as the addition of the `INPUT` type, many enhancements to the system only affect a small module.

6.3 General Optimizations and Heuristics

The general optimizations and heuristics described in this section provide back-end independent performance enhancements. Each of these optimizations, tree pruning, operation analysis (algebraic optimizations), derived-parameter generation, and redundancy elimination are described in greater detail. The development of this code base has been primarily done by Christopher T. Church, while the underlying algorithms have been jointly developed.

6.3.1 Tree Pruning

The outputs of the model, *e.g.* membrane potential, ionic currents, etc., can either be explicitly specified by the modeler or implicitly specified to include every state of the system. When the outputs are explicitly specified, it is possible that certain trees within the run tree forest are not required for the generation of the output. If a population of independent neurons is described in the DML model but only one membrane potential output is specified, the additional, independent neurons can effectively be pruned from the system, with no loss to the desired output.

The tree pruning optimization finds all trees in which the root node, or the *write* node is not included in the list of desired outputs. This candidate list of trees to be pruned is cross-referenced against all the *read* nodes of the trees that directly supply outputs. If a *read* node matches a *write* node of the candidate list, then that tree is indirectly required for an output and is removed from the list. The remaining trees on the candidate list are pruned. Warnings are displayed to the user specifying which states are pruned, as this can very often be a result of a modeling error.

6.3.2 Operation Analysis

The operation analysis component provides for all the tree-based algebraic optimizations. These include constant folding, strength reduction, and manipulation by algebraic laws and properties [57]. The operation analysis rules shown make up all the rules that are currently implemented and will be discussed in more depth below. In these rules, x_n represents arbitrary expressions while k_n represents literals in either real or integer number formats. An operation in all caps (ADD, SUB, etc.), represents an operation that is passed on for future evaluation while infix binary or unary operations are computed as part of the rule. The run forest is evaluated across each tree (rooted at a *write* node) via a pre-order traversal.

Identity rules are the simplest of the algebraic rules (see Eq. (34)). These reduce addition by zero and multiplication by one to a single argument.

$$\begin{aligned} \text{ADD}(0, x_1) &\rightarrow x_1 \quad \text{(Rule add0)} \\ \text{MULT}(1, x_1) &\rightarrow x_1 \quad \text{(Rule mulby1)} \end{aligned} \tag{34}$$

The commutative and associative arithmetic laws allow for reordering of operands in certain operations (see Eq. (35)). In this case, the rules are implemented for scalar addition and multiplication. The commutative property allows the two operands of a binary operation to be flipped and are formally described in rules #2 and #4. The associative property allows for the two binary operations required to compute a three-operand operations to be reordered. The last three rules reorder operations based on the associative property to simplify the application of other rules.

$$\begin{aligned} \text{ADD}(x_1, k_1) &\rightarrow \text{ADD}(k_1, x_1) \quad \text{(Rule \#2)} \\ \text{MULT}(x_1, k_1) &\rightarrow \text{MULT}(k_1, x_1) \quad \text{(Rule \#4)} \\ \text{ADD}(x_1, \text{ADD}(x_2, x_3)) &\rightarrow \text{ADD}(\text{ADD}(x_1, x_2), x_3) \quad \text{(Rule \#7)} \\ \text{MULT}(x_1, \text{MULT}(x_2, x_3)) &\rightarrow \text{MULT}(\text{MULT}(x_1, x_2), x_3) \quad \text{(Rule \#8)} \\ \text{MULT}(k_1, \text{MULT}(x_1, x_2)) &\rightarrow \text{MULT}(\text{MULT}(k_1, x_1), x_2) \quad \text{(Rule prop\_lit)} \end{aligned} \tag{35}$$

Distributive transformation enable an outer operation to propagate to the operands of the inner operation. In the first two transformations (see Eq. (36)), a multiplication by a constant is propagated through an addition or subtraction to produce two inner multiplications by a constant. This is not always desirable and highlights the fact that algebraic transformations do not always produce the most desired result. The latter three transformations force the negation operation to go to the leaves, where it can potentially be removed via a derived parameter (see Section 6.3.3) or constant folded (see below).

$$\begin{aligned}
\text{MULT}(k_1, \text{ADD}(x_1, x_2)) &\rightarrow \text{ADD}(\text{MULT}(k_1, x_1), \text{MULT}(k_1, x_2)) \quad \text{(Rule \#13)} \\
\text{MULT}(k_1, \text{SUB}(x_1, x_2)) &\rightarrow \text{SUB}(\text{MULT}(k_1, x_1), \text{MULT}(k_1, x_2)) \quad \text{(Rule \#15)} \\
\text{NEG}(\text{ADD}(x_1, x_2)) &\rightarrow \text{SUB}(\text{NEG}(x_1), x_2) \\
\text{NEG}(\text{MULT}(x_1, x_2)) &\rightarrow \text{MULT}(\text{NEG}(x_1), x_2) \\
\text{NEG}(\text{DIV}(x_1, x_2)) &\rightarrow \text{DIV}(\text{NEG}(x_1), x_2)
\end{aligned} \tag{36}$$

Constant folding combines multiple literals into one literal for reduced processing. This removes the need for the FPGA to compute the same value every cycle, wasting resources in the process. These rules are expressed in Eq. (37). The trivial constant folding rules deal with binary operations with only literals as its arguments. The more complex rules deal with two level functions whereby operands in both levels are combined and reduced. Certain transformations such as rule #10 provide obvious benefit, reducing two multiplications into one multiplication. Other transformations such as rule #11 has questionable benefit as the number of operations are equal even with a partial result pre-computed.

$$
\begin{aligned}
\mathrm{ADD}(k_1, k_2) &\rightarrow k_1 + k_2 \quad \text{(Rule \#1)} \\
\mathrm{MULT}(k_1, k_2) &\rightarrow k_1 \cdot k_2 \quad \text{(Rule \#3)} \\
\mathrm{SUB}(k_1, k_2) &\rightarrow k_1 - k_2 \quad \text{(Rule \#5)} \\
\mathrm{DIV}(k_1, k_2) &\rightarrow k_1 / k_2 \quad \text{(Rule folded\_const\_div)} \\
\mathrm{NEG}(k_1) &\rightarrow -k_1 \\
\mathrm{MULT}(k_1, \mathrm{ADD}(k_2, x_1)) &\rightarrow \mathrm{ADD}(k_1 \cdot k_2, \mathrm{MULT}(k_1, x_1)) \quad \text{(Rule \#11)} \\
\mathrm{MULT}(k_1, \mathrm{DIV}(k_2, x_1)) &\rightarrow \mathrm{DIV}(k_1 \cdot k_2, x_1) \quad \text{(Custom Rule)} \\
\mathrm{ADD}(\mathrm{ADD}(k_1, x_1), k_2) &\rightarrow \mathrm{ADD}(k_1 + k_2, x_1) \quad \text{(Rule \#9)} \\
\mathrm{MULT}(k_1, \mathrm{MULT}(k_2, x_1)) &\rightarrow \mathrm{MULT}(k_1 \cdot k_2, x_1) \quad \text{(Rule \#10)}
\end{aligned}
\tag{37}
$$

Operation strength reduction occurs when an more difficult to compute operation is replaced with simpler operations. Specific to an FPGA architecture, divisions are more complex to compute then multipliers, which are more complex then additions or subtractions. Often fewer calculations are performed as a result of a strength reduction transformation. The first transformation (see Eq. (38)) reduces a multiply by -1 to a unary negation operation. The second removes an outer negation of a subtraction by a simple reordering of the terms. The third removes two cascaded negations. Finally, the last operation transforms a division by a literal into a multiplication by the inverse of the literal.

Many other operations are possible in this area. For example, a shift is a trivial operation to compute on an FPGA. A shift, however, produces either a multiplication or division by a power of 2. A proposed transformation can remove multiplication or division by these constants into a zero-resource shift. This technique can be extended to multiplication by constants such as 10, which is equivalent to the sum of the argument left shifted by 1 and by 3.

$$\begin{aligned}
\text{MULT}(-1, x_1) &\rightarrow \text{NEG}(x_1) \quad \text{(Rule mulby-1)} \\
\text{NEG}(\text{SUB}(x_1, x_2)) &\rightarrow \text{SUB}(x_2, x_1) \\
\text{NEG}(\text{NEG}(x_1)) &\rightarrow x_1 \\
\text{DIV}(x_1, k_1) &\rightarrow \text{MULT}(x_1, 1/k_1) \quad \text{(Rule div-inv)}
\end{aligned} \tag{38}$$

The operation analysis portion of the DYNAMO back-end can substitute computable operations for functions defined in the front-end language. Current, the square and cube operations are remapped to multiplications as shown in Eq. (39). The operation analysis in this way can act to enlarge the library of functions by mapping them to know computable constructs.

$$\begin{aligned}
\text{SQR}(x_1) &\rightarrow \text{MULT}(x_1, x_1) \\
\text{CUBE}(x_1) &\rightarrow \text{MULT}(\text{MULT}(x_1, x_1), x_1)
\end{aligned} \tag{39}$$

The operation analysis phase can provide increased performance for software and hardware targets. Based on domain knowledge of FPGA computation, additional hardware-based optimization can be performed. The current implementation can be augmented with additional rules to further improve performance, such as strength reduction of multiplications and divisions to shifts.

Our current implementation converts the tree to a *sum-of-products* representation. This occurs primarily due to the rules following the distributive laws, producing multiple terms of factors. This may not be ideal on an FPGA since multipliers are more costly than additions. A future version of the operation analysis routines should look at *product-of-sums* representations. This requires factoring terms into its representative factors. Example subset of proposed rules follows:

$$
\begin{aligned}
x_1x_2 + x_1x_3 &\rightarrow x_1\left(x_2 + x_3\right) && \text{(Proposed Rule \#1)} \\
x_1x_2 - x_1x_3 &\rightarrow x_1\left(x_2 - x_3\right) && \text{(Proposed Rule \#2)} \\
x_1^2 + 2x_1x_2 + x_2^2 &\rightarrow \left(x_1 + x_2\right)^2 && \text{(Proposed Rule \#3)} \\
x_1^2 - x_2^2 &\rightarrow \left(x_1 + x_2\right)\left(x_1 - x_2\right) && \text{(Proposed Rule \#4)}
\end{aligned} \tag{40}
$$

Proposed rules #1, #2, and #4 reduce two multiplications to one multiplication. Proposed rule #3 reduce four multiplications to one multiplication. These are just a small subset of the rules that can aid in reducing multiplier dependency on the FPGA.

Other rules based on recasting [69], might be helpful to alter and remove difficult to compute functions. For example, an exp function is not directly computable on an FPGA and instead requiring a lookup-table. In its place, a simple differential equation can be used calculate the exp function:

$$
\exp(x_1) \rightarrow x_2, \quad \left\{ \frac{dx_2}{dt} = x_1, x_2(0) = \exp(x_1(0)) \right\} \tag{41}
$$

While this approach adds an additional differential equation, it removes a lookup-table. Another example can be found for trigonometric functions such as the sin function.

$$
\sin(x_1) \rightarrow x_2, \quad \left\{ \frac{dx_2}{dt} = x_3, \frac{dx_3}{dt} = -x_2, x_2(0) = \sin(x_1(0)), x_3(0) = \cos(x_1(0)) \right\} \tag{42}
$$

These approaches require additional effort to implement and understand from precision and stability perspectives. A general rule for performing these recasting transformations will require enhanced algebraic and analytical differential capabilities in the DYNAMO compiler and thus can not be completed at this stage of development.

6.3.3 Derived-Parameter Generation

The operation analysis routines additionally generate quantities we refer to as *derived parameters*. These derived parameters are generated by the system to have reduced computational overhead over the original parameter set. Each derived parameter is a function of the other parameters and literals in the model. For example, neural models often evolve membrane potential according to

$$
\frac{dV_{\text{mem}}}{dt} = \frac{I_{\text{app}} - \Sigma I}{C_{\text{mem}}}. \tag{43}
$$

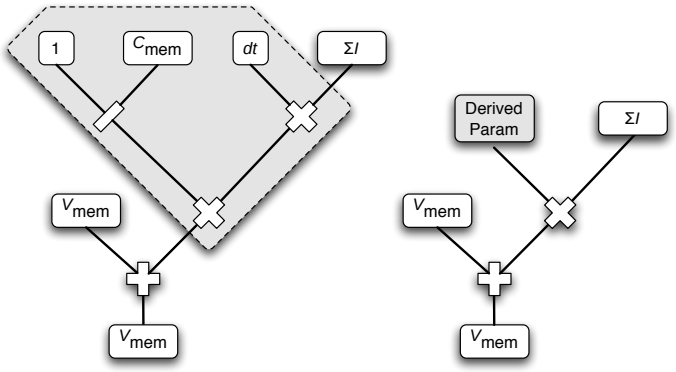


Figure 23: Example of a derived parameter transformation replacing a reciprocal of a parameter, C_{mem}, scaled by an additional parameter, dt. Two multiplications and a division in the first tree are replaced by a single multiplication via this transformation. The derived parameter, DP, is now determined by $DP = (C_{\text{mem}})^{-1}dt$.

In this equation, C_{mem} is a parameter representing membrane capacitance. As a result of algebraic optimization rule div-inv, the division by the parameter is transformed into multiplication by the inverse, $\frac{1}{C_{\text{mem}}}$. Performing this calculation on every iteration of the model will become unnecessary as this quantity only changes when C_{mem} changes. This expression becomes an ideal candidate to become a derived parameter.

When the model is executing, a change in C_{mem} will trigger the software infrastructure to compute the inverse. This new parameter is sent to the system, freeing the computation engine from computing $\frac{1}{C_{\text{mem}}}$. On an FPGA, this is particularly beneficial since the reciprocal operation would require a lookup table to implement. An example taken from a model compiled through DYNAMO is shown in Figure 23. Here, two parameters, C_{mem} and dt, and three operations were compressed into one derived parameter and one multiplication operation.

Specifically, a number of tree operations are performed to generate derived parameters. A pre-order traversal of the run tree at each node flags all preceding leaf nodes that are literals/constants and parameters as candidates for collapsing. Leaf nodes that are states, inputs, or of the input–avail type are not candidates for collapsing. When all predecessor leaf nodes are flagged as collapsible, that tree is pruned from the system at that node, replaced with a uniquely indexed derived parameter *read* node. The pruned tree is then added into the parameter initialization tree with the write node set to the previously generated derived

parameter index.

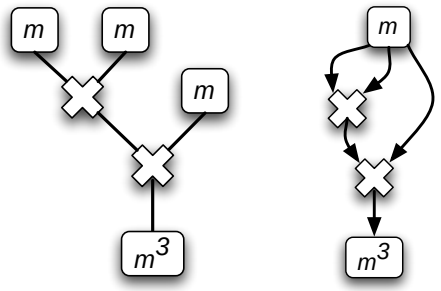
6.3.4 Redundancy Elimination

Redundancy elimination attempts to find identical expression within the run tree forest and replace with a single representation. The first step in this process is the conversion of the data structures from a tree representation to a directed flow graph (DFG). A DFG lacks the clear structure inherent in a tree. In a tree, the root node is preceded by unique nodes which are preceded by other unique nodes until the leaf nodes are reached. In contrast, a DFG node can precede multiple other nodes removing a clear structure. Therefore, a pre-order traversal from a DFG node can double-count preceding nodes as multiple paths can utilize these nodes.

A comparison between the tree form and the DFG is show in Figure 24. While these two data structures represent the same expression, m^3, there are significant differences between the two representations. An expression tree can be trivially traversed, reaching each node exactly one time. A DFG is not as obviously traversed as multiple paths might include the same node. Additionally a DFG can contain cycles, or loops where a path originating at a node can return to the same node. This would amount to there being an algebraic loop within the graph itself, such as the equation, $a = a + b$, might produce. This is not allowed unless a is a state variable, in which a would then become a write node. DYNAMO instead uses a subset of the DFG representation, a directed acyclic graph (DAG) which allows no inner loops. For the remainder of this chapter, DAG and DFG will be considered interchangeable in the context of DYNAMO.

The graph shown in Figure 24(b) is a *directed* graph due to the arrows indicating program flow. These arrows are termed edges of the graph. While a tree data representation will take the form of a linked list, a sparse matrix is often used to represent a DFG. This transformation to a DFG is required to perform redundancy elimination. The resulting graphs will take the form of Figure 24(b) where identical expressions within the graph are consolidated and edges redrawn to the root node of the new expression.

Redundancy elimination works on a larger scale than the m^3 expression example. The



(a) Expression Tree (b) Directed Flow Graph

Figure 24: Illustration of the expression m^3 in both the tree form and the directed flow graph (DFG). The DFG representation is able to link a single *read* node for the quantity m to both multiplication operations. The arrows show the directional flow of data through the graph.

graph is traversed from the bottom nodes, those nodes with no successors. Sub-expressions are compared recursively and edges redrawn to remove duplicate computations. This is operation is costly in terms of compiler performance but still finishes in polynomial time, approximately $O(n^3)$. Despite the compiler cost, this optimization free the modeler from having to simplify and collect sub expressions.

The optimizations in this section all work to enable the modeler to work in their own domain, describing the model in such a way that is clear for them. The DYNAMO compiler works to improve the computational effectiveness of the target by simplifying algebraic expressions, removing static calculations from the data-path, and eliminating redundant computations.

6.4 Hardware Back-end Specific Analysis

This phase of DYNAMO is designed to optimize the model around a hardware/FPGA back-end. At this point, the model consists of three DFGs: the run graph, the parameter graph, and the state initialization graph. There are additional data structures available, such as the type table, command line arguments, inputs, outputs, and output rate which are generally not used in these analysis. The output of this step consists of four graphs where the run graph is marked up with precision and timing information. The fourth graph contains expressions that can not be approximated or are optimized out in the back-end. This graph

is referred to as the lookup-table graph.

6.4.1 Lookup-table Generation

In a general-purpose computer, there are built-in ALUs (algebraic logic units) and libraries available for almost any type of computable function. For example, a processor might have a built-in adder/subtracter, multiplier, and divider. Other functions such as a square root or exponential function are often calculated via an iterative algorithm [42]. Thought is rarely given as to the computational overhead of such an algorithm as computers are generally considered to be "fast enough."

To maximize performance on an FPGA, we have developed our architecture to be synchronous and to utilize maximal pipelining (see Section 6.6 for more information). Therefore, none of our blocks utilize internal loops, instead enabling one operation to complete per clock cycle (the operation latency can still be greater than 1, where the internal registers become the pipeline). Iterative operations are difficult to implement in this architecture since the loops must be unrolled prior to evaluation. To remain synchronous, the number of iterations must be known at compile time. Increased accuracy often requires more iterations. For many operations, the resources required to compute would use a disproportionately large set of resources on the FPGA. Therefore, we consider these operations to be difficult to compute and make them candidates for lookup-table approximations.

Lookup-tables are large storage elements implemented as *Block RAMs* within the Xilinx architecture. A simple transformation linearly maps the input into a address for the *Block RAM* according to the following relationship:

$$\mathrm{addr(x)} = (\mathrm{x} - \min(\mathrm{x})) \cdot \frac{2^{\mathrm{n}} - 1}{\min(\mathrm{x}) + \max(\mathrm{x})} \tag{44}$$

where x is the input, n is the addressability of the RAM, and the functions min() and max() represent the range of the input value. These tables often have between 1k (1024) and 4k words, which provides a suitable approximation for a function. No interpolation or extrapolation is performed. The inputs are set to saturate at the minimum and maximum to avoid wrapping the addresses of the table if out of range. In total, an adder and multiplier

are required for the address translation along with the *Block RAM* to complete the lookup table structure.

Lookup-tables are only applicable to unary functions. The division operation is difficult to compute, however it can not be directly converted into a lookup-table since it is a binary operation. Instead, the division has to be reformulated as a multiplication by a reciprocal.

$$\frac{x_1}{x_2} \rightarrow x_1 \cdot \left(\frac{1}{x_2}\right) \tag{45}$$

For a similar reason, an arbitrary power can not be evaluated. Instead, this binary function must be converted to unary operations as in this example.

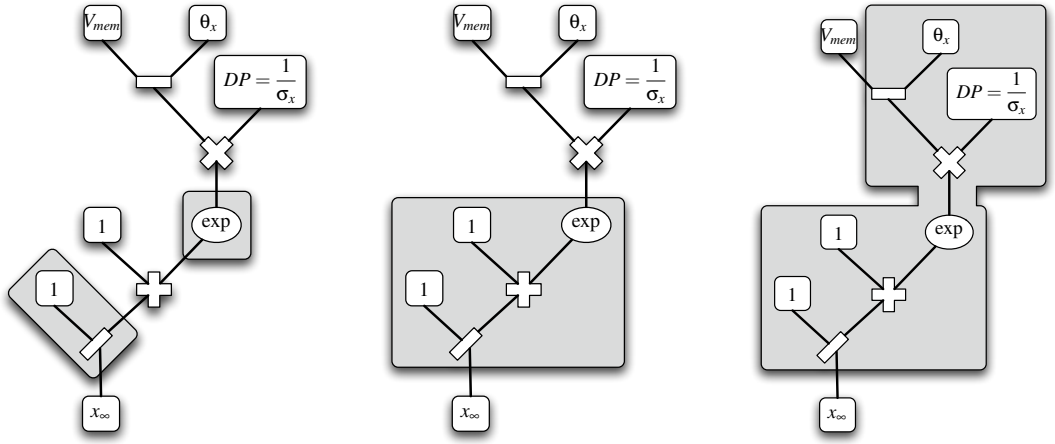
$$x_1^{x_2} \rightarrow \exp\left(x_2 \cdot \ln(x_1)\right) \tag{46}$$

The lookup-table algorithm can only replace sub-expressions that have one variable input.

In the example illustrated in Figure 25, the Boltzmann equation,

$$x_\infty = \frac{1}{1 + \exp\left(\frac{V_{\text{mem}} - \theta_x}{\sigma_x}\right)}, \tag{47}$$

contains both a reciprocal and an exponentiation. Since both of these operations are not computable in the hardware, they both require a lookup-table approximation. In Figure 25(a), the Boltzmann equation is shown with two shaded blocks, representing the functions that must be converted to lookup-tables. In this minimal algorithm, only the non-computable functions are approximated. In the next pane, a greedy algorithm is illustrated whereby one lookup-table is enlarged to cover as many operations as possible. Three operations are now encompassed by the lookup-table. This is preferred as the total resource requirements are dramatically reduced. Two lookup-tables, each with an adder and multiplier and the inner addition operation are required for the transformation shown in Figure 25(a). In Figure 25(b), one lookup-table is required with its one adder and multiplier. Two adders, a multiplier, and a *Block RAM* are saved. Finally, in Figure 25(c), we consider the case where θ_x and σ_x are constants and not parameters. In this possibility, an additional two operations, a subtraction and a multiplication, were absorbed. This illustrates how the resource cost of setting a value as a `CONSTANT` is reduced relative to a `PARAMETER`.



(a) Minimal Algorithm (b) Greedy Algorithm (c) Greedy Algorithm with Constants

Figure 25: Three possibilities for defining a lookup-table from a Boltzmann equation Eq. (47). The proposed lookup-tables will encompass the shaded blocks within the graph. a. This examples employs a minimal algorithm which requires two lookup-tables for this expression. b. In this case, a greedy algorithm reduces two lookup-tables to one lookup-table. c. If the parameters, θ_x and σ_x, are each defined as a `CONSTANT` instead of a `PARAMETER`, the expanded greedy tree covers an additional two operations. Note that the trees depicted in the figure are DFGs.

A future enhancement could execute the derived parameter routines for a second pass. The addition of the linear mapping circuitry in front of the lookup-table memory structure can potentially be combined with the parameters to form new derived parameters likely saving the subtraction and multiplication operations in Figure 25(b).

The algorithm, written by Christopher T. Church, starts by iterating through each node of the run graph. At each node, a sub graph is found by expanding out through its predecessor nodes to encompass as many operations as possible, according to the greedy algorithm illustrated in Figure 25(b). Each sub graph is evaluated into an operation cost. The costs are summarized, non-exhaustively, in Table 8. If the total sum of the costs per operations encompassed by the sub-expression exceed 29, than the table is generated. The value 29 along with the costs in Table 8 are fairly arbitrary and have been tuned to achieve our desired output. With these heuristics, raising a quantity to the 5th power would cause a multiplier-only implementation. Increasing that to the 6th power would switch it to a lookup-table. These can continuously be tuned as the requirements evolve.

The sub-expression that is targeted for a lookup-table is extracted and added as a tree

Table 8: Lookup-table heuristics

Operation	Cost
EQ	1
ADD	2
SUB	2
NEG	2
MULT	5
SQR	5
CUBE	10
DIV	30
EXP	30
SIN	1000

in a new lookup-table graph. The root of this tree becomes a new *write* node with a table ID value. In the run graph, the subexpression is replaced in this phase with a custom lookup-table function and an adder/multiplier for addressing during the precision analysis phase. At this point, the four graphs are passed on to the precision analysis phase.

Lookup-tables are not always the ideal solution to handling these non-computable functions. Table size increases exponentially as the precision requirement increases. With finite FPGA resources available for block memory, additional approaches need to be explored. Future enhancements to lookup-table generation can be found in Section 6.9.

6.4.2 Precision Analysis

The majority of neural models are simulated using the floating point number system. A floating-point number consists of three components, a sign, an exponent, and a mantissa. Commonly used in simulation tools such as MATLAB, the IEEE double precision format utilizes [1] a 64-bit word, where the first bit is the sign, the next 11 are the exponent, and the final 52 bits are the fraction. This number system provides an extremely large range of values, from $\pm\ 10^{323.3}$ to $\ 10^{308.3}$. Effectively, the exponent provides for the order of magnitude, while the relative precision on the value is determined by the size of the mantissa.

Computations with a floating-point representation are made efficient on general-purpose processors through dedicated hardware blocks capable of computing these operations at high

speed. These blocks are generally custom-designed integrated circuits specifically for high speed computation. While these arithmetic logic units (ALUs) can run at a very high frequency, performance is often lagging for a number of reasons. First, a processor is often limited to one to several arithmetic processors. Second, because of the memory architectures utilized in these processors, it is very difficult to maintain a high level of utilization for these execution units.

An FPGA processor can not run at the same clock rate as the specialized execution units in a general-purpose processor. What it lacks in raw throughput, it compensates by instantiating numerous execution units that can be executed in parallel. In addition, since DYNAMO builds what amounts to a unique instruction set architecture based on the model description, significant utilization can be achieved more readily.

DYNAMO has been designed to not use floating-point numerics because of the significant resources required to implement them efficiently. This would limit the total number of operations we can implement in the FPGA, effectively reducing the degree of parallelism. Instead, we have adopted fixed-point numerics to replace floating-point. Fixed-point operations are as computational intensive as integers with the capacity to represent fractional quantities.

6.4.2.1 Fixed-point Computation

In fixed-point computations, the operands are predefined to have a set number of integer bits and fractional bits along with an optional sign bit. For example, the `int32` type in C is a form of a fixed-point number with one sign bit, 32 total bits, and zero fractional bits. An 8-bit signed integer might have a range from -128 to 127. The same 8-bit number in the form of *Fix8_3*, referring to a signed number with three fractional bits and eight total bits, would have a range of -16.0 to 15.875.

Fixed-point numbers utilize a two's complement format to encode signed numbers defined as

$$\texttt{negation} \rightarrow (-x) = !(x) + 1 \tag{48}$$

where the ! operator performs a bitwise `NOT` operation on x. Overflow bits are ignored when

negating. The sign of a number is readily determined by the most significant bit (MSB).

This notation is very useful for generating compact hardware addition and subtraction units. A simpler cascaded adder, or a ripple adder, is made of a cascaded set of bitwise full adders. A full adder has three inputs, A, B, and C_{in}, and two outputs, S and C_{out}, representing the sum of the inputs and the carry-in bit, C_{in}. The carry-out bit, C_{out}, becomes the input to the carry-in bit of the successive full adder. The gate level logic is defined as

$$\begin{aligned} S &= A \oplus B \oplus C_{in} \\ C_{out} &= A \cdot B + A \cdot C_{in} + B \cdot C_{in} \end{aligned} \tag{49}$$

where $\oplus$ is the exclusive OR (XOR) operator, $+$ is the OR operator and $\cdot$ is the AND operator. All adders made from these full adder building blocks are sign agnostic. Because of the two's complement representation, both signed and unsigned numbers can be added with no additional circuitry. Subtraction requires minimal circuitry. The subtraction $A - B$ is evaluated as $A + (-B)$. In practice, B undergoes a bitwise negation and the least significant C_{in} bit is set to a one to complete the two's complement negation.

When two fixed-point numbers are summed or subtracted, the decimal places are pre-aligned with no changes other alterations to the structure. Floating-point addition is very costly in hardware since an alignment step must occur prior to the computation. This requires a addressable shift-register, which for wide bit-widths can be extremely costly and contributes to the majority of used resources. This is the primary reason for the use of fixed-point numerics. Since multiplication does not require any pre-alignment, floating-point multiplication units utilize marginally more resources than their fixed-point equivalents. A mixed floating-point, fixed-point design is not practical as conversion between the number formats will require the costly addressable shift-registers.

6.4.2.2 Precision Estimation

The DML specification requires each `PARAMETER`, `STATE`, and `INPUT` to be defined with the precision in the form of a low value, a high value, and a step size. For example, the DML statement

Table 9: Table of range propagation rules for n-ary operations

	low (L^*)	high (H^*)	step (S^*)
$x_1 + x_2$	$L_1 + L_2$	$H_1 + H_2$	$\min(S_1, S_2)$
$x_1 - x_2$	$\min(L_1 - H_2, H_1 - L_2)$	$\max(L_1 - H_2, H_1 - L_2)$	$\min(S_1, S_2)$
$x_1 \cdot x_2$	$\min(L_1L_2, L_1H_2, H_1L_2, H_1H_2)$	$\max(L_1L_2, L_1H_2, H_1L_2, H_1H_2)$	S_1S_2
$x_1?x_2 : x_3$	$\min(L_1, L_2)$	$\max(H_1, H_2)$	$\min(S_1, S_2)$

```
PARAMETER I_in (0 TO 40 BY 0.1) = 20;
```

defines an interval $[0, 40]$ with a step of 0.1. We define this precision to mean that each value defined within the interval of [all 401] value should be representably with a fixed-point resolution within a tolerance of 50% the step value. For literals of the system, we quantize the literal to minimum precision to preserve the value with 1% precision. For example, the number $x = 0.1$ can be represented as a *UFix9_9* number. When quantized, $x \simeq 0.099609375 = 9b.000110011$.

The range and step information is propagated in a single pre-order traversal through the run graph. This range and step information is determined by a unique algorithm per operation type and is based on operands. For example, an addition of a variable by a constant will result in a shift of the range by the constant and no change to the step. However, the addition of two variables will expand the range beyond both operands as well as adjust the step size.

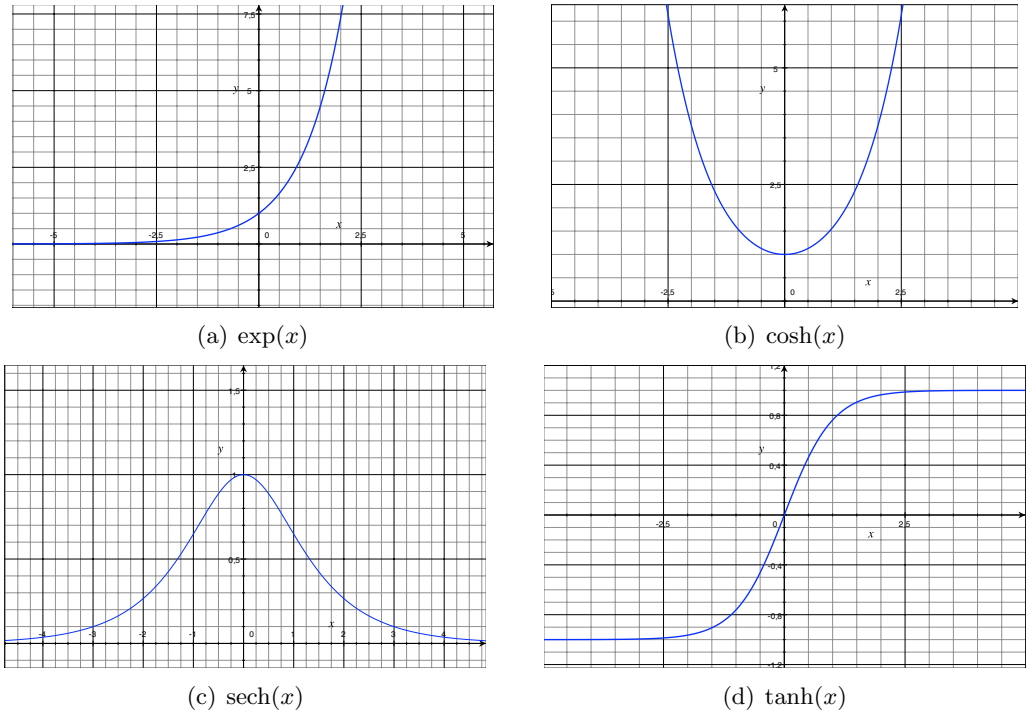
Rules for the propagation of range and step information through the run graph for binary and ternary operations are summarized in Table 9. These rules were generated primarily to remove the possibility of overflow. Overflow is condition by which the value exceeds the bounds of the representation. On the FPGA, the most significant bits are lost and the value is wrapped, causing systemic miscalculations. Overflows are catastrophic failures—the simulation can not continue after an overflow condition. Underflows occur when the fixed-point representation is insufficient to represent a very small number. In this case, the value reverts to zero. This is undesired as small values can be important depending on the particulars of the model. However, the absolute error of the misrepresentation is very small,

despite the relative error being 100%.

Overflows are also easier to determine and correct for. The addition of two numbers cannot exceed the sum of the maximums of the operand ranges. A similar rule holds for multiplication, the largest product of a multiplication is the maximum of the inner products of the ranges. It is more difficult to make a general rule for the step size through an operation. It can be argued that an addition operation should propagate the maximum of the operand step sizes. This would imply that the operand with the reduced fractional precision would negate any additional procession in the smaller value. However, in certain operations where small numbers are accumulated, such as in an integration, the step size must follow the more precise operand. The resulting rule uses the conservative approach and propagates the smaller step through an addition and subtraction. For multiplication operations, the full precision is preserved only when the new step size is the product of the step sizes of the operands. This is highly conservative. For example, the cubing of the m activation gate requires two multipliers. It can be argued that the precision of m^3 is no different than the precision of m, since both represent the same activation function. The compiler does not have the information to make that distinction. While this is not an optimal approach—excessive resources are required for larger multipliers, precision is preserved through multiplications.

The `IF` operation simply passes on the worst-case precision of its dual operands. Other operations, such as logical operators and comparison operators work solely on binary values and therefore do not propagate precision information.

Dynamo defines numerous unary operations each of which must propagate precision. These operators are illustrated in Figure 26 and in Figure 27. The functions depicted in these plots are commonly found in neural models or are representative of other functions that might be used. For example, a shape of a hyperbolic cosine/secant function can be used for modeling the slowing of a time constant around a gate activation. Exponentials are often used for modeling activation functions and synapses. Functions where the range is bounded, such as in a hyperbolic secant (see Figure 26(c)) and hyperbolic tangent (see Figure 26(d)), are preferable for conversion into fixed-point as they span few orders of magnitude. A



(a) $\exp(x)$ (b) $\cosh(x)$

(c) $\text{sech}(x)$ (d) $\tanh(x)$

Figure 26: Plots of the four continuous functions that are supported and processed through the precision analysis phase.

hyperbolic cosine function (see Figure 26(b)) or an exponential (see Figure 26(a)) with a large domain will quickly span many orders of magnitude requiring excessive precision to describe the function.

The equations representing the rules for propagating ranges through the unary functions are summarized in Table 10. The negation operator simply inverts the domain where the low becomes the high and the high becomes the low. The step remains unchanged. The square root operation is currently set to simply take the square root of the elements of the domain to determine the output range. This is an approximation based on being a one-to-one function for all positive real inputs. Additionally, since a square root grows sub-linearly, we have found that less precision is required for its representation.

This can be contrasted with an exponential function, $f(x) = \exp(x)$ which can quickly grow by orders of magnitude as x increases. For $x < 0$, the exponential function approaches zero, which potentially requires significant fractional precision. This makes determination of precision through an exponential particularly difficult to reason through. When the

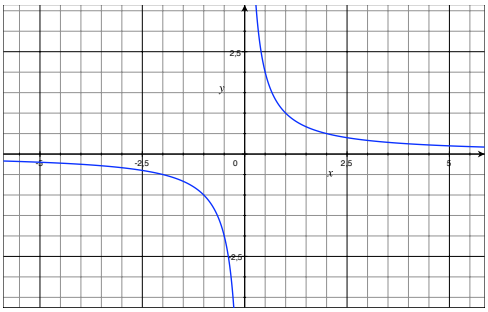


Figure 27: Plot of a reciprocal partial function. The discontinuity of at $x = 0$ makes this is a special case as it is not a total function across the real domain.

minimum input domain is small as determined by the exponential of the minimum compared with the step size, the function is approximated to have an output range leading with zero. The max output range is readily found by $\exp(H)$ while the step size remains constant. This is to balance the need to increase the step size for $x > 0$ and decrease the step size for $x < 0$. It is a compromise but has worked sufficiently well to date.

The hyperbolic functions in the table are all processed in different ways based on their varied classifications. All three functions have all real numbers as their domain. The range for a hyperbolic cosine function is $[1, \infty)$. If the input range crosses zero, the minimum is set to 1, otherwise the largest and smallest hyperbolic cosine evaluations make up the output range. The step size is set such that the number of unique inputs equals the number of unique outputs. A similar process is followed for range determination of a hyperbolic secant function which has a range of $(0, 1]$. In this case, the maximum output range is set to 1 based on the zero crossing. The minimum output, L^*, and step, S^*, are computed by a similar method as described above. The hyperbolic tangent function as a one-to-one function is readily determined by the same method as the square root. However, for this operation, the output step size does not have to be computed since the hyperbolic tangent function is approximately linear for small inputs.

The reciprocal function, or a constant divided by a variable, is valid across two domains, the set of all positive real numbers or the set of all negative numbers. The domain cannot span across zero. When this happens, it becomes no longer possible to reason on the output range as it spans $(-\infty, +\infty)$. This underlying condition occurs when processing the

Table 10: Table of range propagation rules for unary operations. The O-times symbol, $\otimes$, is true when the input range of the function crosses zero. The comparison operator, $(cond)?(iftrue) : (iffalse)$, is borrowed from C for brevity. Each of these are functions of one variable except for the division operation whereby the numerator is an arbitrary constant, k, and the denominator is the unary operand.

	low (L^*)	high (H^*)	step (S^*)
$-x$	$-H$	$-L$	S
$\sqrt{x}$	$\sqrt{L}$	$\sqrt{H}$	$\sqrt{S}$
$\exp(x)$	$(\exp(L) < S)?0 : \exp(L)$	$\exp(H)$	S
$\cosh(x)$	$\otimes?1 : \cosh(\min(\lvert L\rvert, \lvert H\rvert))$	$\cosh(\max(\lvert L\rvert, \lvert H\rvert))$	$\cosh(\max(\lvert L\rvert, \lvert H\rvert))\frac{S}{H-L}$
$\operatorname{sech}(x)$	$\frac{1}{\cosh(\max(\lvert L\rvert, \lvert H\rvert))}$	$\otimes?1 : \frac{1}{\cosh(\min(\lvert L\rvert, \lvert H\rvert))}$	$\frac{S}{H-L}$
$\tanh(x)$	$\tanh(L)$	$\tanh(H)$	S
k/x	$\min(k/L, k/H)$	$\max(k/L, k/H)$	$S\frac{H^*-L^*}{H-L}$

Hodgkin–Huxley model as described in Section 6.4.2.3.

After the range information is propagated through all operations within the run graph, the data structure adjusts to include precision information. Each range is converted to a precision based on the 50% of the step rule described above. Lookup-tables defaults, in this version, to utilize a maximum of 18-bits. Other operations can grow to arbitrary bit widths unless constrained by the `bitwidth` compiler option. This option can cap the total number of bits for any operation. This is helpful and often required as bit-widths tend to grow rapidly as the ranges propagate through the graph structure. A maximum bit-width is a useful heuristic that can partially offset the conservative slant utilized in estimating precision. The final step adds the addressing circuitry and appropriate bit-widths to the graph in the new data structure denoted as the precision run graph.

6.4.2.3 Precision Challenges

The precision algorithm implemented to date is not a perfect algorithm. There has yet to be an ideal or proven algorithm to convert a floating-point simulation to a fully equivalent fixed-point simulation (see Chapter 8 for more detailed analysis of this issue). Despite being able to readily convert the majority of simple mechanistic models (ion channels, for example), the sum of the parts does not necessarily imply the production of an identical model. This can be because of an oversimplified model definition or might even require

numerical imperfections.

For an example of the latter, consider the generally accepted form for an α activation function for a sodium channel in a Hodgkin–Huxley model. The α function often has the form of

$$\alpha_m = \frac{\frac{V_{\text{mem}}+40}{10}}{1 - \exp\left(\frac{-(V_{\text{mem}}+40)}{10}\right)} \tag{50}$$

which describes the forward rate of m-gate activation as a function of V_{mem}. This expression should be valid for all physiologically-relevant membrane potentials, but instead exhibits a discontinuity at $V_{\text{mem}} = -40$. This discontinuity is caused by the denominator of Eq. (50) crossing zero when $V_{\text{mem}} = -40$. The α function does not approach infinity as the denominator approaches zero since the numerator also approaches zero. This 0/0 condition is known as a removable singularity and refers to a point, x^*, where the $\lim_{x \to x^{*-}} f(x) = \lim_{x \to x^{*+}} f(x)$, but where $f(x^*)$ is undefined.

This condition is often not a problem because it is unlikely that $V_{\text{mem}} = 40$ given that these simulations often uses double precision floating point. It becomes a significant concern for fixed-point simulations where the minimum precision is utilized to maximize simulation performance. It is likely that the original formation is not tied directly to the physiology as a removable singularity can not occur in cell, but is instead an undesired artifact of the curve fit function.

Precision determination remains an open area of research and will likely remain that way for some time to come. There appear to be no fool-proof methods to consistently convert floating-point to fixed-point. It is further likely that it might never be possible to fully convert between representations. Despite this, fixed-point is a advantageous number system and might find itself as a target at the onset of future model development.

6.4.3 Timing Analysis

In a sequentially executed architecture, such as on a general-purpose computer, instructions or commands are evaluated serially. If a particular operation requires additional time, or clock cycles, to complete, the processor pipeline feeding the execution engine stalls. This

causes no issues as a processor is generally not a real-time[1] device (certain software platforms can provide some real-time capability to a processor). When multiple execution units are present, such as in a reconfigurable device, it is critical to properly time operations to maintain synchrony.

All operations instantiated on the FPGA have a sample period of 1 cycle. This means they are able to produce a new output every cycle of the simulation. Each operation also has a latency associated with it, defined as the number of cycles in which a change in the input would cause a change in the output. The delays within an operation are known as pipeline stages. More pipeline stages allow a greater number of simultaneous computations propagating through the operation although a maximum of one output per cycle is possible. In this analysis phase, the number of cycles of latency per operation is determined based on the properties of each operation.

Previous work has been done in estimating optimal latencies through operations. In one study, we generated a six-deep cascaded operation chain where we incrementally increased the latencies and recorded the post-place & route critical path time. We found that increasing latency linearly improved the maximum clock frequency until a point where the curve flattened and the maximum frequency was achieved. This suggested that if, for example, 2 cycles of delay were sufficient to get 90% of maximal performance, utilizing additional FPGA resources to increase the latency will yield marginal benefit. This work was repeated for four operations, addition, subtraction, multiplication, a constant multiplication, or a scaling operator. More information can be found in Chapter 4.

Operations over larger bit widths require more resources to implement and therefore often require additional cycles of latency to maintain a high throughput rate. The previously described study utilized bit widths of 14-bits and 28-bits. Many precisions grow to larger than 28-bits using our precision estimation algorithm. Therefore, we embarked on a more exhaustive study of performance as a function of precision of its operands.

We utilized the Xilinx Core Generator (7.1i) software package to generate operation macros for addition, subtraction, addition and subtraction dual-use, and signed/unsigned

[1]Real-time is defined here to refer to a definite time evaluation.

multiplication. The core generator was configured to create a performance optimized core for a given bit width. A performance optimized core will have a sample period of one clock cycle (*i.e.* will utilize no internal overclocking), and have sufficient pipeline stages to maintain high clock throughput. Exactly how this algorithm chooses the number of stages is proprietary to Xilinx. Nonetheless, we were able to use this information as guidelines for our own latency selection analysis.

We ran the Core Generator tool for all combinations of inputs between 2 and 64 bits. We assumed that the operands are commutative, for example, a 10-bit input summed with a 20-bit input is equivalent to a 20-bit input summed with a 10-bit input. We found that the results for adders, subtracters, and mixed-use adders/subtracters, have approximately the same performance. For this reason, only the adder/subtracter combination is illustrated, although the results are similar for the single-user operations.

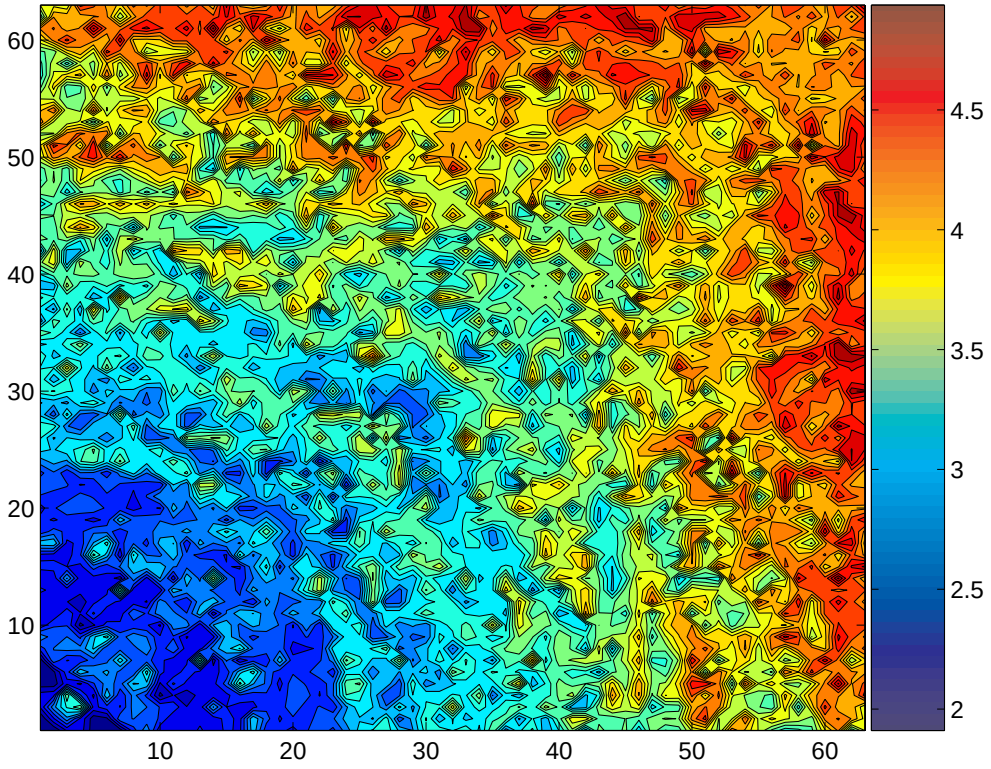
Each adder was set to have a delay of 2 cycles to generate baseline data across all adder sizes. This allowed for a chain of registers at the inputs and the outputs. For addition and subtraction, we found that consistently fast cores were generated for all bit widths (see Figure 28(a)). Maximum clock frequencies, defined as the inverse of the critical path delay, went from 200 MHz to almost 500 MHz corresponding to approximately 5 ns down to 2 ns, respectively.

For the DYNAMO timing analysis routines, we chose to implement summing operations with one cycle of delay if the output bit width is less than or equal to 32-bits and two for the remainder of cases. We use the following expression to define the lat as a function of the bit-width, n.

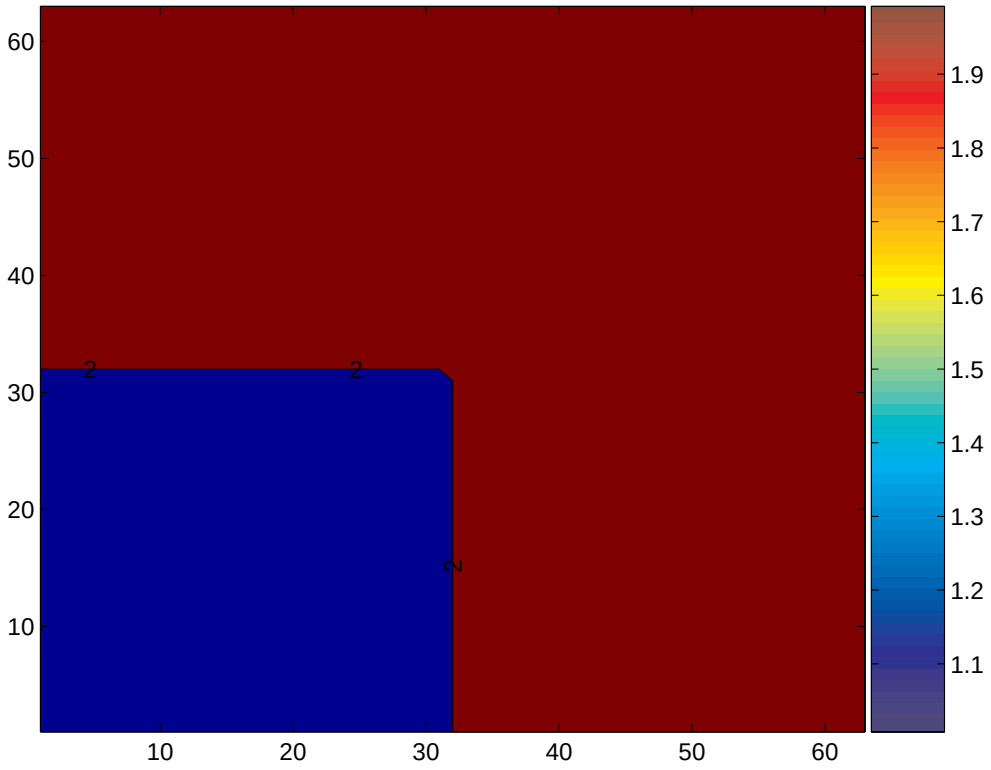
$$lat = \lceil n/32 \rceil \tag{51}$$

The cut-off, 32, is chosen arbitrarily and is currently used as a place holder. The resulting latencies are plotted in Figure 28(b). The extra delay was primarily added in deference to longer wire delays that can exist as operations become larger.

It should also be noted that this timing analysis is performed for the case where an adder is generated using Xilinx slices and not the pre-built Xilinx DSP blocks, which consist of a multiply–accumulate operation and three multiplexers. Future enhancements can take



(a) Critical path delay (ns)



(b) Pipeline stages

Figure 28: Adder/Subtracter contour plots showing a) minimal critical path delay and b) number of internal pipeline stages, as a function of bit widths of the operands. Plot a) is in units of ns and plot b) refers to the total latency of the operation as it is implemented in Dynamo. All analysis are performed with two pipeline stages.

advantage of these internal adders.

Multipliers are not built from slices but instead from DSP blocks. Each DSP block can compute an 18-bit by 18-bit signed multiplication generating a 35-bit signed product. Alternatively, a 17-bit by 17-bit unsigned computation will generate a 34-bit unsigned product. Core Generator was tasked to produce each combination of $2-64$ bit wide inputs with a latency of one cycle. Both signed and unsigned multiplications were tested with similar results. The resulting critical path time is shown in Figure 29(a) where the maximal operating frequency ranges from approximately 40 MHz to almost 160 MHz. This result is not typical as additional pipeline stages can be added to reduce the latency. The purpose instead was to ascertain the role of increased bit widths on period without compensation.

Core Generator suggests a pipeline depth to maximize performance. The number of cycles of latency was found to follow the following relationship

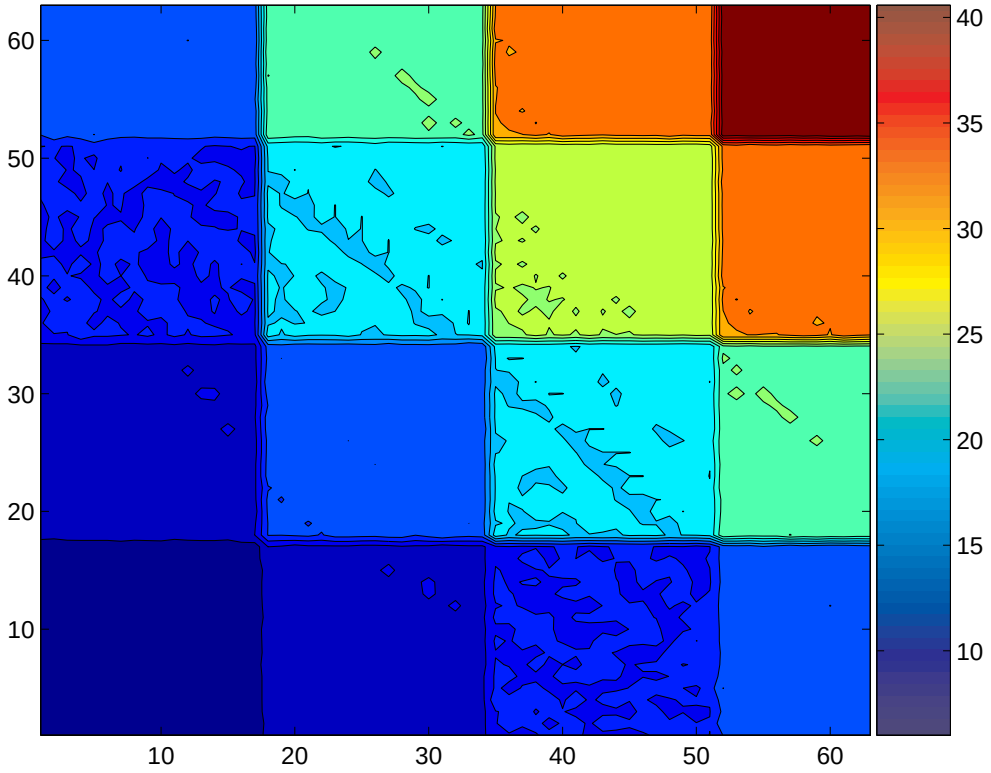
$$\begin{aligned} op_{\mathrm{A}} &= \lfloor (a-1)/17 \rfloor \\ op_{\mathrm{B}} &= \lfloor (b-1)/17 \rfloor \\ lat &= 4 + op_{\mathrm{A}} + op_{\mathrm{B}} + op_{\mathrm{A}} \cdot op_{\mathrm{B}} \end{aligned} \tag{52}$$

where a and b are the bit widths (minus the sign bit) of the A and B inputs, respectively (see Figure 29(b) for a graphical depiction). A multiplier with 64-bit inputs would require 19 latencies according to Eq. (52). Since the DYNAMO precision analysis component routinely forces multipliers to this size, register usage can get fairly constrained. Instead, we use a more conservative relationship described by

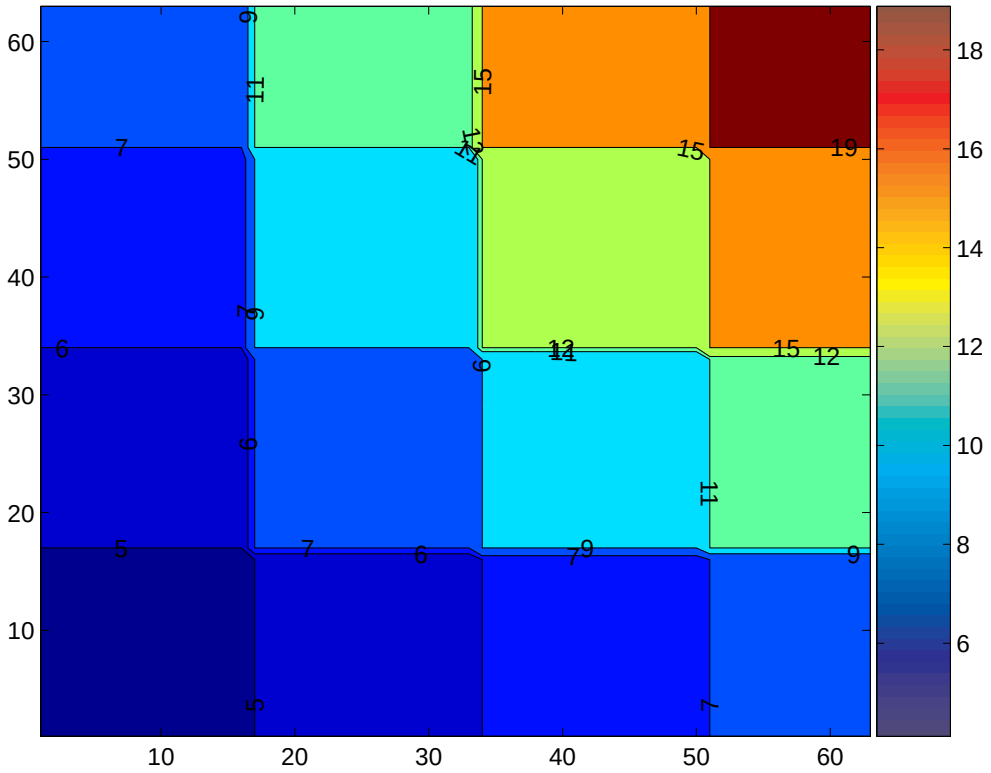
$$lat = \lceil a/17 \rceil + \lceil b/17 \rceil. \tag{53}$$

This removes the extra latencies caused by the cross products of op_{A} and op_{B} and reduces the base 4 stages required for all multipliers. The ceiling function is used to compensate for small multipliers. In practice, this is sufficient to remove the critical timing path from multipliers.

Other operations are determined by a table of operations and latencies as summarized by Table 11. Most operations require relatively small footprints and therefore include only one level of registers. This is less for the performance of operations but more to mitigate



(a) Critical path delay (ns)



(b) Pipeline stages

Figure 29: Multiplier contour plots showing a) minimal critical path delay and b) number of internal pipeline stages, as a function of bit widths of the operands for a signed multiplication. Plot a) is in units of ns and plot b) refers to the total latency of the operation as suggested by the Xilinx Core Generator. All analysis are performed with one pipeline stage.

Table 11: Table of pipeline stages for each operation type.

Operation	Latency
ADD	see Eq. (51)
SUB	see Eq. (51)
MULT	see Eq. (53)
NEG	1
NOT	1
GT	1
LT	1
GE	1
LE	1
EQ	1
NE	1
AND	1
OR	1
IF	1
SQR	see Eq. (53)
CUBE	2× Eq. (53)
TABLE	1

wire delay and provide the synthesis tool with additional registers to maximize overall performance.

The timing information is compiled per operation and appended to the graph structure to create a new precision and timing expression graph (PTEGraph). Timing analysis is the final stage of processing prior to reaching the target back-ends. The output, the PTEGraph is therefore the final graph output and is utilized throughout the software and hardware back-ends.

6.4.4 Operation Correlation Table

While many of the neural models of interest show significant regularity, the back-end graph representation is devoid of structure. The hardware back-ends can take advantage of regularity if there was a way to propagate this type of information without the structure in tact. We employed a operation correlation table as a means to encode regularity for the benefit of the back-end.

This table contains correlations between each operation within DYNAMO. For operations

Table 12: Correlations by operation over the four passes of the correlation determination algorithm. The first pass is the intial pass, followed by a forward pass, a backward pass, and an additional forward pass. This example only shows correlations between the two equations in Eq. (54). Correlations within each equation are zero and therefore ignored in this example.

Operation Pair	Before	Pass #1	Pass #2 (F)	Pass #3 (B)	Pass #4 (F)
$(\times_1, \times_2)$	0	1	1	3	3
$(\times_1, +_2)$	0	0	0	0	0
$(+_1, \times_2)$	0	0	0	0	0
$(+_1, +_2)$	0	1	2	2	5

that are of different types, for example an addition and a multiply, the correlation is zero. Due to the large number of zeros, the table was built as a sparse matrix where only the non-zero correlations were specified. The correlation table was built along three passes, an initial pass, a forward pass, and a backward pass.

The initial pass iterates through each node in the graph. For every other node with the same operation type, an initial correlation of one was set in the table. At the end of this pass, all addition–addition pairs had a correlation of one, all multiplier–multiplier pairs had a correlation of one, etc. The forward pass sums correlations of predecessor nodes. If predecessor nodes are correlated, the current nodes are increased by the same amount. In the following two expressions,

$$\begin{aligned} a_1 \times_1 b_1 +_1 c_1 \\ a_2 \times_2 b_2 +_2 c_2 \end{aligned} \tag{54}$$

after the initial pass, the multiply operations $(\times_1, \times_2)$ and the addition operations $(+_1, +_2)$ are correlated with a value of one. In the second pass, the addition operations are each preceded by correlated multiplier operations. Therefore, the addition correlations are increased to a value of two. In the third pass, the correlations are backward-propagated such that the multiplication correlations are increased to three. The final forward pass increases the addition correlation to five.

With the *corr_csv* DYNAMO command line option, a comma-delimited file is generated that contains cross-correlations for the *write* nodes. At each *write* node, all predecessor

nodes to the leaves of the graph are extracted. Each combination of two *write* nodes is evaluated by computing the inner cross-correlations of each node within each tree. These cross-correlations are summed to determine a graph wide correlation.

We have plotted the *write* node cross-correlations for three models, a five neuron Hodgkin–Huxley model[40] (see Figure 30), a single Booth, Rinzel, and Kiehn[10] (see Figure 31), and a heterogeneous population consisting of two Booth, Rinzel, and Kiehn models, two Hodgkin–Huxley models, and three FitzHugh-Nagumo models[32, 58] (see Figure 32). Each of these plots were normalized such that the correlation is one for highly-correlated states and zero for non-correlated states. The diagonal, or a state's correlation with itself, is forced to be one.

Five Hodgkin–Huxley models are depicted in Figure 30. The strongest correlations are shown for the voltage terms. The gates are also highly correlated with like gates. It is also telling to see how dissimilar gates show some correlation. Despite being different gates, they share a significant amount of structure.

The Booth, Rinzel, and Kiehn model shown in Figure 31 is an example of a multi-compartment model with an assortment of non-regular ionic conductances. This model additionally contains two states that track Ca^{2+} concentrations. Despite each state having non-identical structure, correlations did emerge that show clear similarities. The N-type and L-type activation and inactivation gates in the soma and dendrite compartments (*mnD*, *hnD*, *ml*, *mnS*, and *hnD*) showed the most significant cross correlations as their structures are most similar. Very strong correlations emerged between the K^+ activation gate and the Na^+ inactivation gate. Weaker, but still significant correlations emerged between Ca^{2+} concentration in the soma and dendrite, but little correlation was found between the two measured membrane potentials, *Vs* and *Vd*. These correlations illustrate how the DYNAMO compiler can take advantage of regular structure in seemingly non-regular models.

In the final example, a heterogeneous model consisting of two Booth, Rinzel, and Kiehn models, two Hodgkin–Huxley models, three FitzHugh-Nagumo models, and a time state is illustrated in Figure 32. This plot shows almost no correlations exist between these unique models. This is not particularly surprising since the Booth, Rinzel, and Kiehn model uses

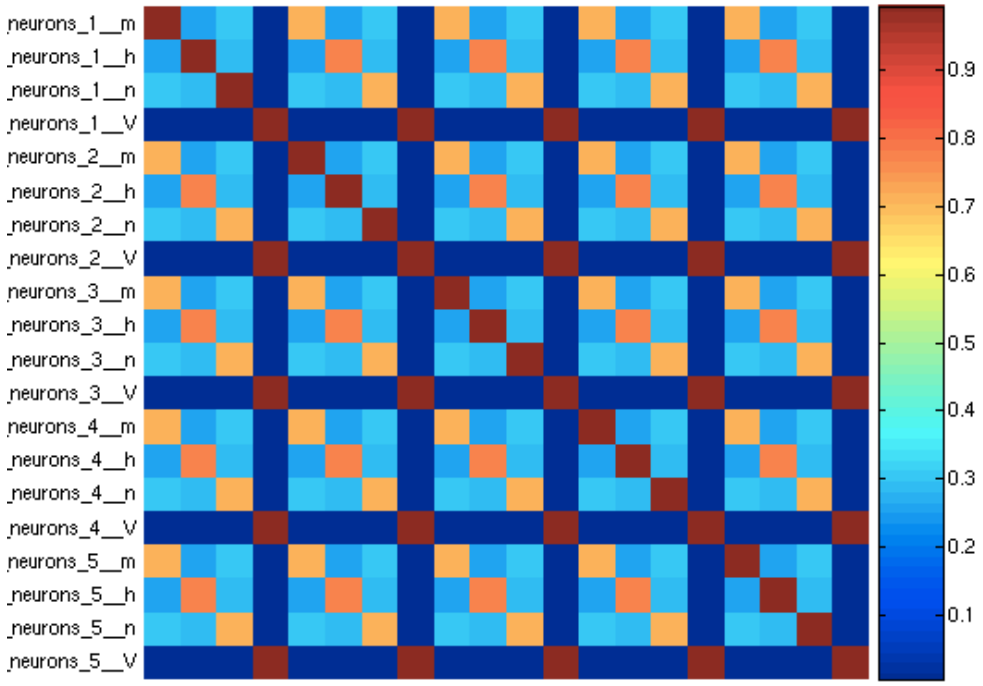


Figure 30: Plot of the correlation table for five Hodgkin–Huxley neuron model[40]. The plot is 20 by 20 consisting of each 4-state neuron model. Higher correlations are indicated by red and lower correlations by blue. Correlations are normalized to go between zero and one.

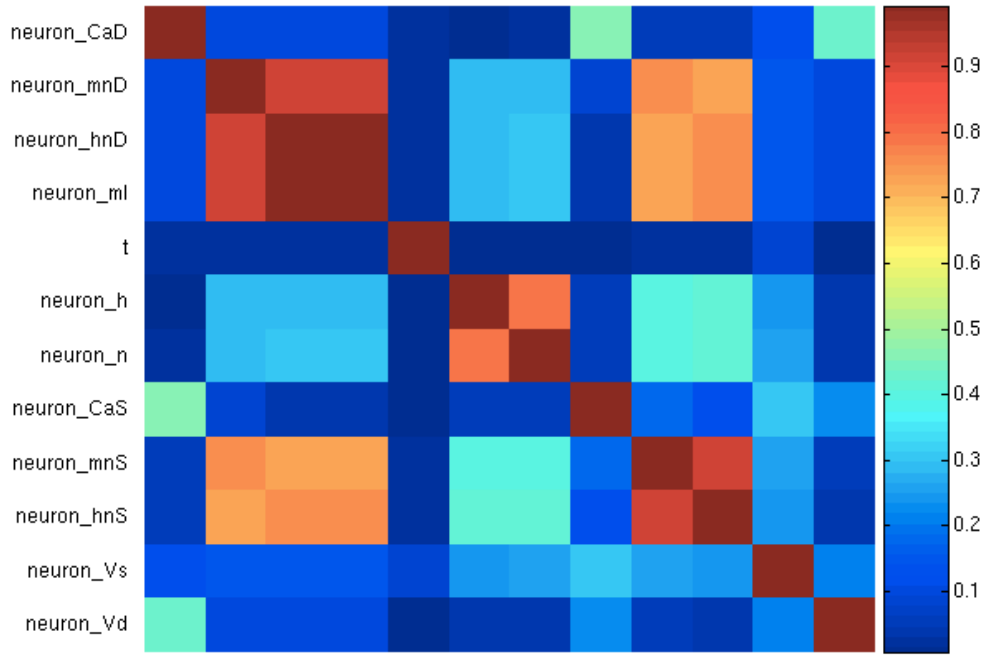


Figure 31: Plot of a single Booth, Rinzel, and Kiehn model[10] cross-correlations. Higher correlations are indicated by red and lower correlations by blue. Correlations are normalized to go between zero and one.

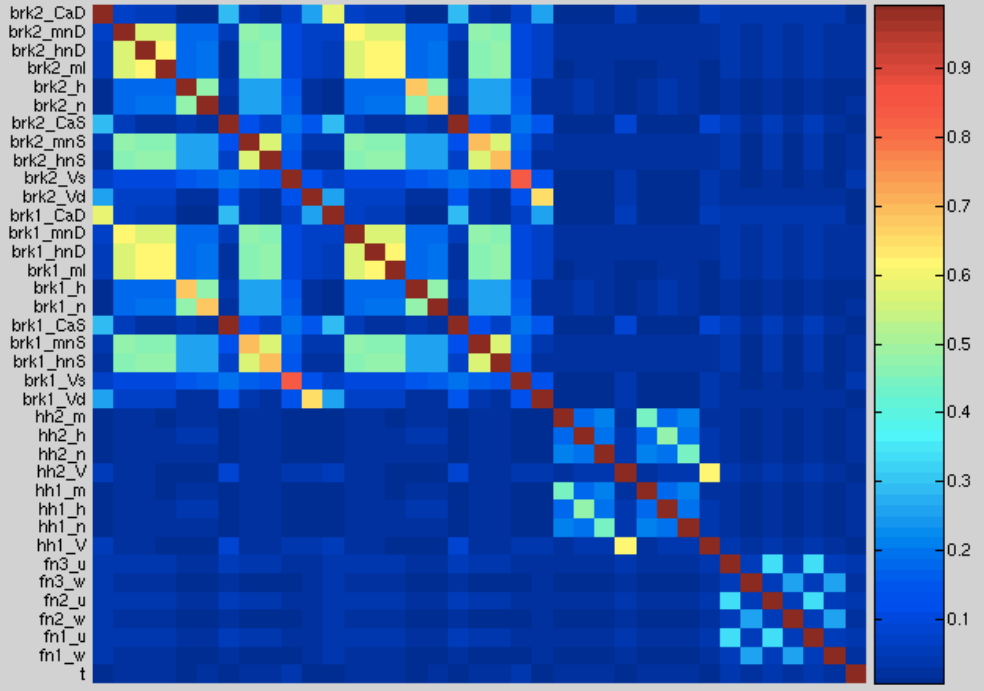


Figure 32: Plot of the correlation table for two Booth, Rinzel, and Kiehn models, two Hodgkin–Huxley models, and three FitzHugh-Nagumo models[32, 58] and a time state. Higher correlations are indicated by red and lower correlations by blue. Correlations are normalized to go between zero and one.

Boltzmann equations for computing ionic conductances verse α–β terms in the Hodgkin–Huxley model. Cross-correlations within each model type clearly emerge as indicated by the parallel lines to the diagonal that are present.

6.5 Software Back-end

The DYNAMO compiler began as an off-shoot of MRCI [65], a compiler for the translation of dynamical systems to a C-based, real-time Linux platform. While the compiler has been completely rewritten at least twice since MRCI, the concept of a software back-end has always been present. We have developed the software back-ends for two primary reasons: 1), to provide an easy to use and fast-compile floating point execution engine for our models written in the DML and 2), to provide a debugging platform for the hardware back-end and the internal components of the compiler.

The software back-end has two primary flavors, a MATLAB back-end that exploits the mathematical capabilities already present in that development environment and a more

recently developed C-based back-end to allow for high-performance, platform-independent software simulations for interaction with our Java-based modeling toolkit.

In general, the software back-ends generate a test bench with three main functions. First, the test bench provides an interface to set model parameters (defined in the DML) and simulation parameters (such as duration, input sequences, output selection). Next, the test bench performs the numerical approximation of the evolution of differential equations making up the model definition. Finally, the test bench provides the means to save data from the simulation for post processing.

The input portion and the output portion are particular to the back-end target, while the core simulation engine is fairly similar (less slight semantic differences). The simulation engine is built by iterating through each of the graphs rooted by the *write* nodes in the run graph. Each *write* node indicates a quantity that must be saved per iteration, such as a state or an intermediate value that is set in the DML as an output. Each graph is traversed post-order such that the preceding nodes are evaluated prior to evaluating the current node. Evaluation of a node consists of converting the operation, value (if literal), or variable name (if *read* node) to a string representation based on the particular back-end's syntax.

Both the C and the MATLAB back-ends execute their simulations using double-precision, floating-point numerics. At the time of this publication, the MATLAB back-end can generate a fixed-point simulation engine with optional implementation of lookup-tables. This back-end has been critical as a tool for studying the implication of fixed-point precision algorithms on model output. The fixed-point back-end utilized the Fixed-Point Toolbox in MATLAB, which provided an extremely flexible environment for simulation, but very poor performance relative to the floating-point simulation engine. The C fixed-point back-end is currently in development and will utilize arbitrary length integer libraries to perform fixed-point simulation.

6.5.1 Matlab

The MATLAB development environment (Mathworks, Natick, MA) provides a powerful platform for analyzing, processing, and visualizing large sets of data. As a general-purpose

Table 13: Information stored in DIF file fields as a function of the type of number system.

DIF Property	Floating-Point	Fixed-Point
Inputs	Name	Name, Range, Precision
Outputs	Name	Name, Precision
Internal Params	Name, Equation	Name, Equation, Precision
External Params	Name	Name, Range

programming language, models can be developed in the MATLAB language and readily simulated numerically. The combination of a full programming language suited for mathematical evaluation and a suite of high-level analysis tools makes MATLAB an ideal platform for a software back-end target.

We have additionally exploited System Generator, a custom blockset for Simulink, as a target for the FPGA back-end. Maintaining a common platform for the hardware back-end and a software back-end has allowed us to develop a substantial library of tools to interact with both targets. This infrastructure that we developed consists of a custom file format containing relevant DYNAMO information from the generation phase and from the original DML model description, a *dynmodel* class to coordinate the interfacing to the software and hardware model, and a set of graphical interfacing tools for user control of the model.

6.5.1.1 Dynamo Information File (DIF)

The DYNAMO Information File, or DIF, is generated by DYNAMO along with the simulation engine and contains all the information required for interacting with the simulation engine above. This includes global model information, such as model name, platform (software or FPGA), output rate, and time-step. The DIF file additionally compiles information on inputs, outputs, internal parameters, and external parameters. The data per type is dependent on whether the target simulation is to be performed in fixed-point or floating-point as shown in Table 13.

Inputs are the basis for protocols (a methodology by which a model is evaluated and metrics are deduced) and provide the means for stimulating the model. Neural models typically have inputs that can be injected currents, command voltages (for a current clamp),

or synaptic events. For floating point simulations, inputs are referenced by name only. For a fixed-point simulation, the input must be coupled with a range to avoid an overflow condition from occurring, and a fixed-point precision to properly quantize the input signal.

Outputs can either be a state or an intermediate value to be saved by the system. The DIF format enumerates the outputs and specifies precision information in the case of a fixed-point simulation engine. Outputs (and inputs) are also influenced by an output rate property in the DIF which downsamples (and upsamples) by a specified factor.

Parameters are all those quantities defined in the DML to be adjustable by the modeler. Parameters are split into external and internal parameters as a consequence from generating derived parameters (see Section 6.3.3). External parameters are those quantities that are adjustable directly by the modeler, they have external relevance. Internal parameters are evaluated as a function of the external parameters and provide for direct input into the FPGA. The separation of internal parameters from external parameters removes static calculations from the execution loop. In the case of membrane capacitance, C_{mem} would be the external parameter while $\frac{1}{C_{\text{mem}}}$ is the corresponding internal parameter.

For fixed-point simulations, the DIF file has additional fields for both internal and external parameters. For internal parameters, since they are user defined, a range is required to keep the model within normal operating conditions. External parameters require a precision and are quantized since they directly feed the software or hardware simulation.

6.5.1.2 "Dynmodel" Object Framework

To communicate with low-level hardware or a basic software interface within a user-approachable modeling environment requires the use of a driver. A driver is a software interface to translate domain-specific commands, *e.g.* inject a 5 second current ramp, into the low level commands, *e.g.* prepare a 50,000 point quantized vector and assign to the I_{app} input. In the MATLAB interface, a common driver is utilized for software and hardware simulations.

The driver is written as MATLAB class, *dynmodel*. The *dynmodel* class is instantiated with a argument specifying the DIF file. An additional command loads the simulation into working memory. Additional class methods are used for querying the inputs, outputs,

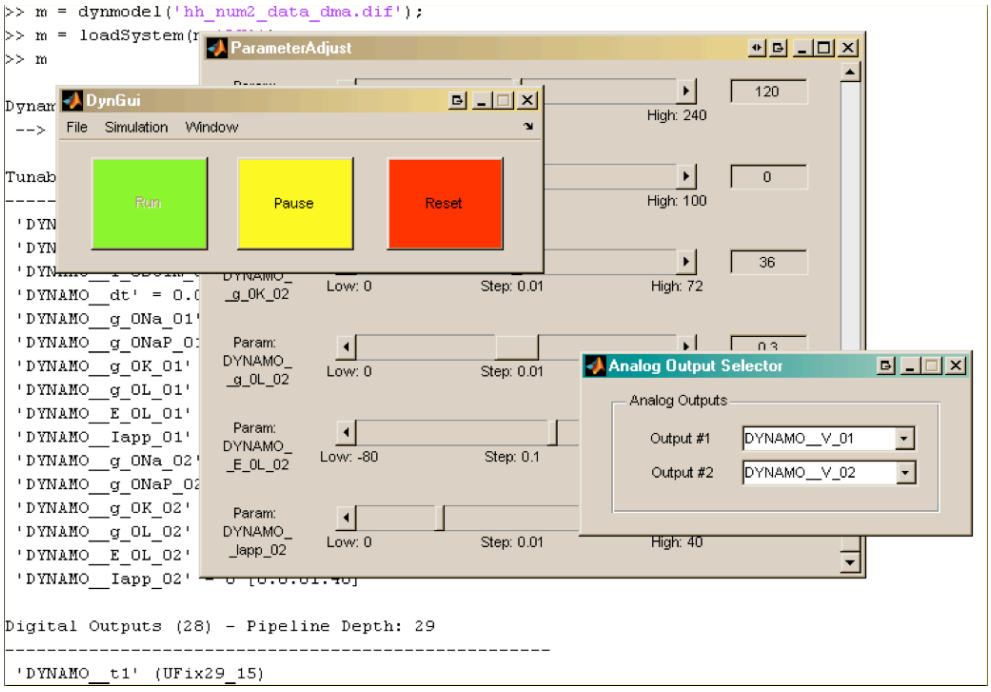


Figure 33: Screen capture of the *DynGui* package depicting the *DynParam* parameter adjustment tools and the analog input and output system. The analog interfaces are for hardware simulation only and will be described in more detail in Section 6.7).

parameters, ranges, etc. and to set new values for parameters, define vectors for inputs, and select outputs for logging. Simulations are executed by an additional command which goes through all processing necessary for running the simulation in software or hardware. Output data is then made accessible in the class for plotting or further analysis.

6.5.1.3 MATLAB *Graphical Interfacing*

Once a driver was constructed in MATLAB, we developed a prototype graphical interface to interact and begin model development using DYNAMO. Graphical interfaces were first developed for a version of the hardware back-end that utilized the analog interfaces on the Xilinx XtremeDSP-II/IV development boards. This version multiplexed all desired outputs on the two embedded analog data converters. This is depicted in the screen capture shown of *DynGui* in Figure 34.

This interface was the first to demonstrate the use of slider bars to modify all the parameters of the system interactively. The slider bars and edit box were bounded by the parameter ranges. This same parameter adjustment interface was incorporated into the

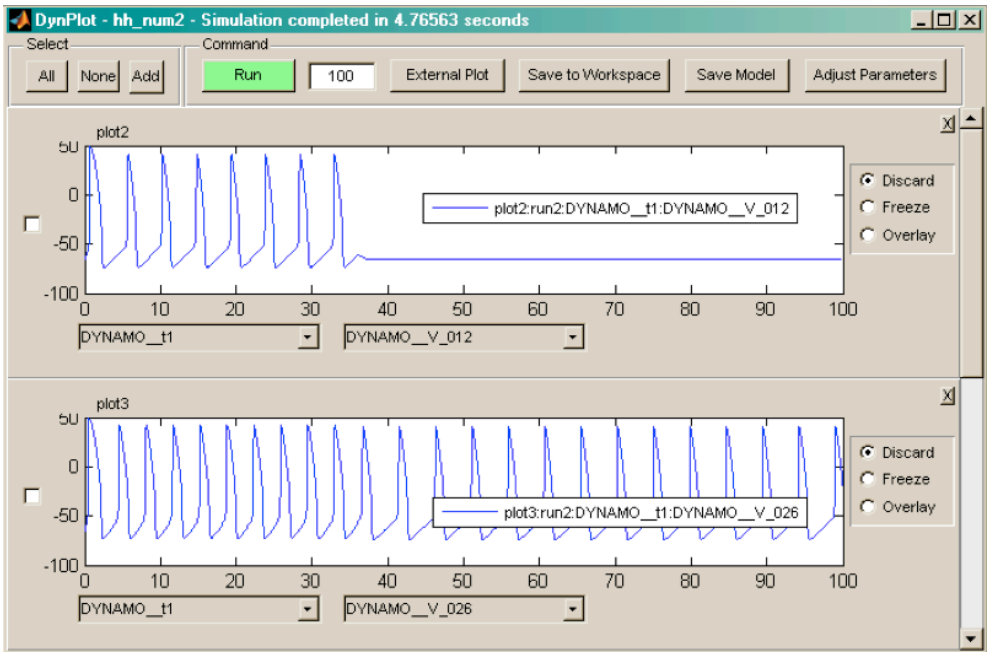


Figure 34: Screen capture of the *DynPlot* package showing a capture of a spiking two neuron Hodgkin–Huxley model. This data was captured using the digital FPGA interfaces. This tool is identical to what is used for software-based simulations in MATLAB.

digital version of *DynGui* called *DynPlot* (see Figure 34).

DynPlot interacts with both the software and hardware back-ends as a traditional modeling environment. Simulations are run according to a set duration after which the selected outputs (via the pull-down menus) are plotted on each graph window. Plots can be held frozen, added to with additional traces, or replaced following each re-execution of the simulator. An arbitrary number of graph windows can be added to investigate numerous outputs. Plots can also be further analyzed by opening a separate plot window or by saving to the workspace.

These tools provided a useful base environment on which to develop and refine the DYNAMO compiler. The latest features have not been ported to *DynPlot*, namely the use of inputs. Due to its close interaction with the *dynmodel* class, those models without inputs should be fully accessible to *DynPlot* with minor modifications. Despite this, development has shifted to the next generation interfaces based on C and Java, as described in the following subsection.

6.5.2 C/Java

Based on the original MATLAB interface and the development of a prototype C back-end, we have embarked on a quest to improve the C back-end to the equivalent feature level of the FPGA back-end. An equivalent feature level would include the ability to perform numerically identical simulation as on the FPGA. This C back-end is intended to be a replacement for the MATLAB back-end. Primary development for the C back-end has been undertaken by Steve Si Jia Feng, but is included here for completeness.

There are several motivating factors for moving to a C-based back-end. First, C is known as a high-performance computing environment with little overhead. It is considered an unsafe programming environment due to the propensity of run-time errors to occur, however autogeneration mitigates much of this risk. Next, fixed-point simulations can be performed efficiently utilizing integer types for calculations under 64-bits. For larger fixed-point operations, 3rd-party public-domain libraries are readily available that optimize method based on size of operands. Third, while MATLAB requires approximately 8 bytes of storage space per value (discovered via inspection within MATLAB), large vectors can be efficiently pre-allocated saving substantial memory and improving performance. Next, compilation of C code can be performed using the freely-available GNU C compiler (gcc) and does not require any commercial products to generate nor execute the simulation engine. Finally, the C back-end can be readily incorporated into a larger framework to develop user interfaces, such as the .NET framework on the Windows platform, XCode for an Apple platform, or Java for platform independence.

We have chosen to develop the C back-end to interface with a Java user application (see Figure 35 and Figure 36). Java is a flexible, object-oriented language with an extensive library of built-in functions and a robust framework for graphical application building. Additionally, it is platform independent. This feature is very important as we develop on a Linux platform while the user base would be primarily on a Windows platform.

The Java to C interface is generated via the Java Native Interface (JNI). This provides the ability for native execute of machine code within a Java application. Once native code is incorporated into Java, some of the benefits of Java are compromised. For example,

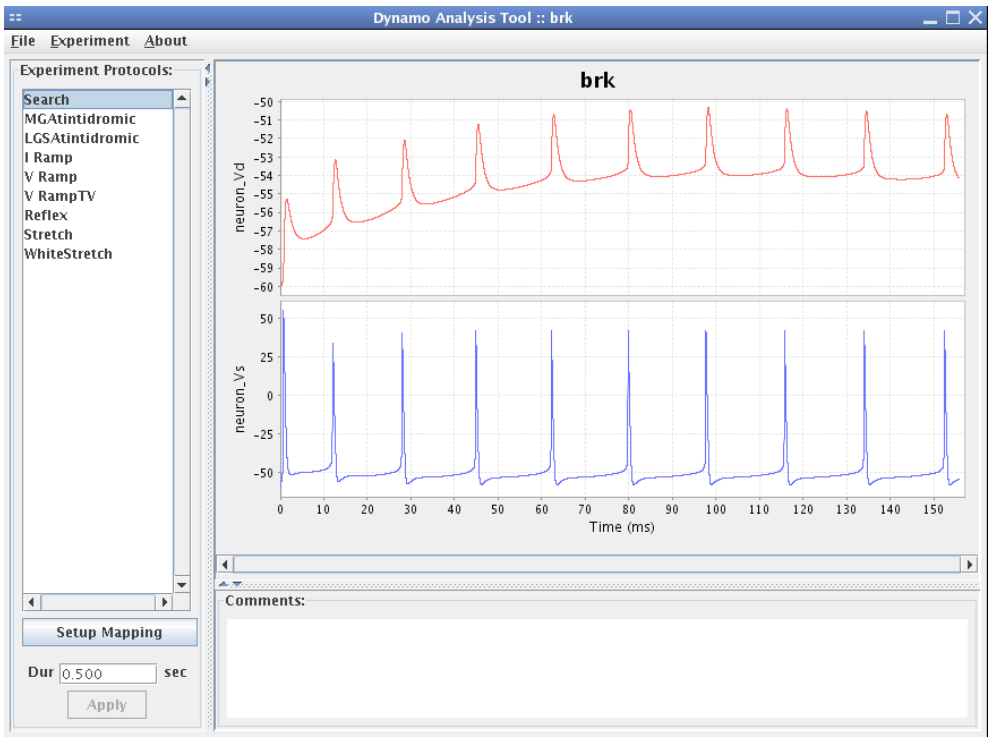


Figure 35: Screen capture of Java GUI depicting the somatic and dendritic voltage trajectories in a Booth, Rinzel, & Kiehn model [10] given a constant input current. The interface, written in Java, utilizes a C-based auto-generated C simulation engine and JNI-based interfacing.

platform independence is lost, but this is mitigated by the fact that the C output can be compiled cross platform. Garbage collection is not possible across the JNI, instead buffers must be released on the heap following usage. Finally, common C runtime errors such as pointer overruns can cause the Java application to crash. This becomes an additional challenge for the development of this C back-end.

The C back-end is in the prototype/development stage. More will be known as the development progresses including performance of the C back-end through the Java interface. It is expected that this JNI-based interface can be extended to other applications beyond simple plotting and parameter tuning. For example, high-level analysis tools such as parameter search routines or sensitivity analysis tools can utilize the same interface independent of Dynamo.

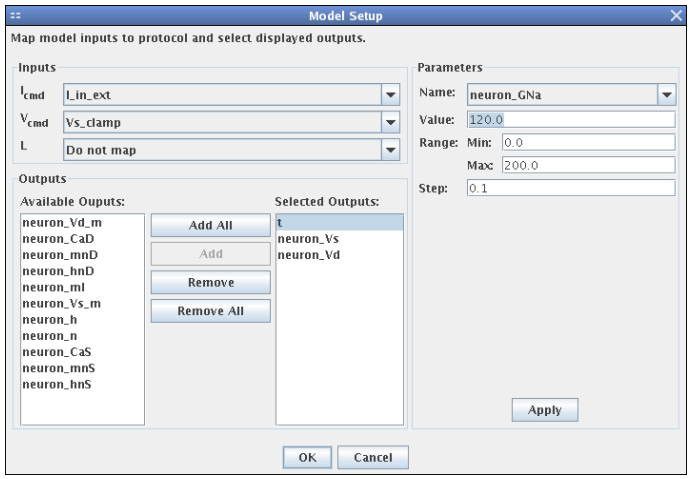


Figure 36: Screen capture of a three-pane dialog box for configuring models' ports and parameters. This screen allows inputs to be mapped to protocol inputs, *e.g.* a voltage-command signal mapping to the somatic voltage state. Outputs are selectively set to be logged while all tunable parameters are accessible through a pull-down menu for run-time tuning.

6.6 Hardware Back-end

The DYNAMO compiler was researched and developed to automate the translation of dynamical systems to a tightly-coupled, high-performance hardware-targeted implementation. The previous sections of this chapter dealt with general compiler transformations, optimizations, and analyses. This section is instead focused on the particular transformations, algorithms, and heuristics that are used in the hardware generation phase.

Our current hardware back-end has undergone three full rewrites. The first version was a simple translator, *i.e.* each operation represented in the run graph was translated to an equivalent System Generator block. This back-end was known internally as the *tree* back-end since it was a direct translation from the graph/tree data structure. For models that exceed the number of available resources, the compiler fails. For one Hodgkin–Huxley model, this method is sufficient, but in general, as models grew in size, this back-end became inadequate.

The first-generation implementation took the form of a multi-cycle processor, each operation was identical where the instruction takes a set number of cycles to compute one iteration of the numerical solution. A multi-cycle architecture is not efficient as only one

cycle of the total cycles per sample period is used for computation. Pipelining is one method processors use to improve the efficiency of a data-path. In a pipelined processor, multiple instructions are processed simultaneously as if moving along an assembly line. As an instruction moves from one stage to the next (via a clock edge), the previous stage is occupied by the next instruction. If a pipeline is fully utilized, the overall efficiency rises by factor equal to the number of stages.

Our previous modeling efforts described as the manual art, manual engineered, and assisted flow methodologies all employ fully utilized pipelines as their core execution engine. In each case, the model chosen had a highly regular form. In the manual art example, a ten-compartment motoneuron model was constructed with a ten-stage pipeline. Likewise, in the assisted flow example, a 40-neuron population model emulating the pre-Bötzinger Complex was constructed with a 40-stage pipeline. For the redesign of the hardware back-end, the question was, how close to a pipelined architecture is possible when the implementation is auto-generated?

There are three main reasons why we would not want to fit the model into a pipelined architecture. First, there is no guarantee that the model will contain enough regularity to form a pipeline of sufficient depth to build a data-path containing all the desired operations. Next, even if there was regularity, the conversion from the front-end language format through the IR removes all hierarchy in the system. The hierarchy of the system would have to be rebuilt accurately to form identical sets of operations that can be pipelined. Finally, for smaller models that are replicated many times, it is not always advantageous to form a single, conventional pipeline. Instead multiple pipelines would provide enhanced parallelism and performance.

We instead worked towards a hybrid multi-cycle/pipeline approach. This approach attempts to find similar structure in sub-expressions such that parts of the data-path would be internally pipelined wrapped in a multi-cycle architecture (more on these two architectures can be found in Chapter 4). We termed this approach *partial pipelining*. Pipelining in the past has been limited to multiple models or multiple compartments. With *partial pipelining*, regularity can be found between ion channels or other model mechanisms. Heterogeneous

networks of neurons can be combined in ways that are not necessarily obvious to the digital designer developing via the manual art approach (see Chapter 3). This new methodology provides a means to handle arbitrary models with arbitrary levels of regularity.

The next version of the hardware back-end was the first attempt to implement the *partial pipelining* algorithm. A resource table was initially generated such that there was one resource for every operation in the described in the run graph. The operations were scheduled to the resources and an area estimation was performed. If the design fit within the target device, the schedule was returned, otherwise, like resources in the resource table were folded and the an additional scheduling was attempted. This was iteratively performed until the resulting schedule was complete. Algorithms for scheduling operations onto appropriate resources were simple and often did not follow an engineer's intuition when manual scheduling.

While the performance of the scheduler was a significant improvement over a software simulation, there was room for improvement. The blue diamonds in Figure 37 are monotonically increasing which is to be expected. As more models are simulated, the possiblity for additional parallelism and reuse will drive up the per model performance. The real-time (RT) performance data only shows part of the picture. Additional information can be garnered by looking at the resulting pipeline depth. This represents the total number of clock cycles required for one iteration of the model and provides a helpful metric to characterize the quality of the schedule.

The pipeline depth for one, two, three, and five neurons is equal to 22, 22, 28, and 36 cycles, respectively. We can see that from one to two neurons, there was no required increase in the number of stages. This implies that two full representations of the Hodgkin–Huxley model fits within the area resources of the FPGA. The real-time factor does not quite double as would be expected. This is due to a slight increase in the critical path time as is often the case when FPGA resources are heavily utilized.

The deficiencies in the schedule become evident when the third Hodgkin–Huxley model is added. Here, the addition increased the number of stages to 28, a 6 stage increase. Theoretically, third model could have utilized a previously instantiated model offset by

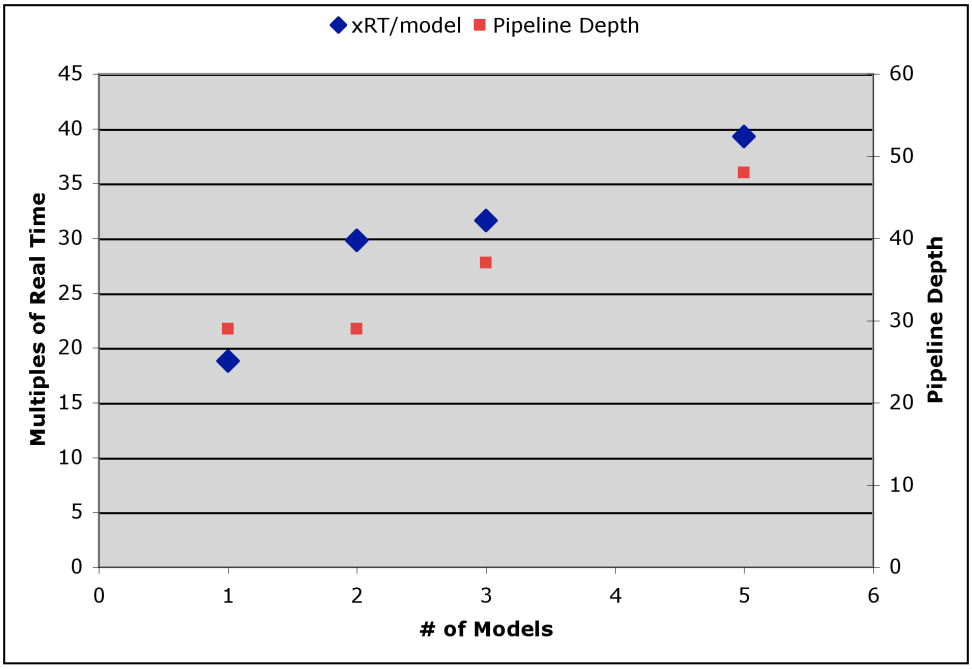


Figure 37: Plot of the real-time performance factor as a function of the number of Hodgkin–Huxley models implemented. Real-time performance is normalized per model. The time step is set at 0.01 ms. Real-time performance is calculated according to the methodology described in Chapter 5. The secondary y-axis depicts pipeline depth as the number of models increase.

one cycle yielding 23 total cycles. Similarly, the fourth model could have used the second instantiated model requiring no increase beyond 23 total cycles. The fifth would have had to been scheduled after the first and third or the second and fourth to increase to a total of 24 cycles. Instead, the jump from three models to five models required 8 additional stages, not the one predicted. This one stage per additional model increase is based on the sample period of one cycle in a pipelined architecture, and the lack of this increase implies a poorly performing schedule algorithm.

This scheduler attempted to use heuristics to decide whether two operations should be scheduled on the same resource. In practice, many more factors must be taken into consideration if the scheduler is to have the "intelligence" to generate an optimized output. Since the *pipelined simulink* back-end did not meet expections, a full redesign was undertaken. At the center of this new scheduler lies a cost function that balances 14 suitability metrics to find the best fit for each operation. Each of these metrics has been adjusted by a weighting, which was tuned by trial and error to provide the best results. The details of which are outlined in the following five subsections. This new and improved scheduler, called the *cost-function scheduler* has been successful in dramatically increasing the size of models that can be auto-generated.

The new scheduler is separated into five components. First a dynamically adjusting resource table is generated based on a profile of the model. The model is then scheduled in the resource table according to the cost function. Multiple optimizations are executed across the scheduler reducing the overall hardware footprint. The resulting design is fully netlisted with the entire infrastructure (see Section 6.6.4 for details). This netlist format is not unlike a typical hardware description language but is used internally as an intermediate representation. The netlist is evaluated a second time for area utilization. If the design exceeds the available resources, the process is repeated by adjusting the parameters of the initial dynamic resource table. The final schedule is then translated into System Generator blocks for generation within Simulink.

6.6.1 Dynamic Resource Table

The previous generation hardware back-end based on the *pipelined simulink* scheduler utilized a large resource table with unique resources per operation. If the resulting schedule went beyond the area resource limitations on the FPGA, the resource table was "folded" back on itself, reducing the size and attempting again. In the latest scheduler, the resource table is preconfigured with resources representative of the model such that the resulting scheduler should fit within the constraints of the FPGA.

The first step involves generating statistics of the model. Each operation of the model is placed into bins based on the operation type (ADD, SUB, MULT, etc.) and latency/delay (z^{-1}, z^{-2}, etc.). The total operations per type of a Booth, Rinzel, and Kiehn (BRK) model [10] are summarized in Table 14. Adders/subtracters will have a latency of either one or two cycles and multipliers can range from 2 to 8 cycles, depending on size. Most other operations will have a latency of one cycle. Latencies are used for binning as they provide comparable bit width operations and contribute directly to the overall performance of the design.

A simple area estimation algorithm is used to count the number of multipliers, block RAMs, and what we term as general-area primitives (GAPs, which include registers and 4:1 function lookup-tables) per resource bin. This is not as exhaustive as the post-scheduling area estimation routines, but provides an approximation. The BRK model was estimated at its original form to consume 12 block RAMs, 6587 GAPs, and 305 multipliers. Since the area estimation is only based on the operations and not on the additional control logic and routing circuitry that are required, a area scaling function is used to reduce the number of available resources. The default scaling factors are 0.7 for RAMs, 0.8 for GAPs, and 0.95 for multipliers. These scales are user adjustable on the command line and are automatically refined if the scheduled design exceeds the area constraints of the FPGA. With these scaling factors, the FPGA has 135 RAMs, 25,600 GAPs, and 183 multipliers.

The design is multiplier-constrained as 305 multipliers blocks are required when only 183 are available. It should be noted that while a total of 85 multiplier operations are in the model, multiplier primitives on the FPGA are limited to 18-bit by 18-bit signed operations,

Table 14: Counts of each operation sorted by type.

Node Type	Count	Latency
READ	45	z^{-0}
WRITE	12	N/A
MUL	85	$22@z^{-2}$, $31@z^{-3}$, $6@z^{-4}$, $17@z^{-5}$, $7@z^{-6}$, $2@z^{-8}$
ADD	28	$13@z^{-1}$, $15@z^{-2}$
SUB	37	$17@z^{-1}$, $20@z^{-2}$
NEG	0	z^{-1}
GT	0	z^{-1}
LT	2	z^{-1}
EQ	0	z^{-1}
GE	0	z^{-1}
LE	0	z^{-1}
NE	0	z^{-1}
IF	3	z^{-1}
AND	0	z^{-1}
OR	0	z^{-1}
LITERAL	15	z^{-0}
TABLE_0	8	z^{-1}
TABLE_1	1	z^{-1}
TABLE_2	1	z^{-1}
TABLE_3	2	z^{-1}

often requiring several multiplier blocks per operation. In the BRK model, multipliers are the constrained resource. The number of multipliers required must be reduced. An equitable algorithm was chosen to reduce proportionally across all the bins that utilize the constrained resource. Both multipliers and block RAMs (whereby the width is greater than 18-bits in the Virtex architecture) require the use of the 305 DSP blocks. The number of multipliers at each latency value along with block RAMs are reduced. This will allow a broad range of resources to be available to the scheduler proportional to its needs within the model as opposed to a only small bit-width or large bit-width multipliers.

The multipliers and RAMs are slowly rescaled over multiple iterations until a new resource table is built that fits within the constraints. The final resource table utilizes 9 RAMs, 6,803 GAPs, and 182 multipliers as summarized in Table 15. Those operations that were reduced in the final resource table are indicated with parenthesis showing the original total. Operation types with more instantiations are preferentially reduced in this algorithm. This is done as those operations are more likely to be ones that follow a regular structure. This is an unintended benefit from rounding up fractional resource counts during each compression cycle (eight compression cycles were required to reduce the table to one that fits within the resources).

This resource table is termed a dynamic resource table (DRT) as the resources have yet to be configured. At this stage, no precision information has been assigned as this will be determined during scheduling. Additionally, operations might take on multiple purposes. For example, an ADD operation can readily perform additions or subtractions or just subtractions. Flexibility is built into this stage to morph as necessary according to the scheduler algorithm.

6.6.2 Hardware Scheduler

This subsection described the process by which the DYNAMO compiler schedules the operations comprising the neural model into the resource table. Briefly, the scheduler iterates through each node of the design, determines the compatible resources, and assigns based

Table 15: Counts of each operation sorted by type and latency in the dynamic resource table. Previous counts before compression of the DRT are shown in parenthesis.

Node Type	Count	Latency
MUL	9 (22)	z^{-2}
MUL	11 (31)	z^{-3}
MUL	4 (6)	z^{-4}
MUL	9 (17)	z^{-5}
MUL	5 (7)	z^{-6}
MUL	2	z^{-8}
ADD	13	z^{-1}
ADD	15	z^{-2}
SUB	17	z^{-1}
SUB	20	z^{-2}
LT	2	z^{-1}
IF	3	z^{-1}
TABLE_0	5 (8)	z^{-1}
TABLE_1	1	z^{-1}
TABLE_2	1	z^{-1}
TABLE_3	2	z^{-1}

on a multi-metric cost function. This scheduler operates on a single pass, it does not backtrack, de-assigning operations. Instead, it is designed in such a way for the schedule to be optimized on additional passes, described in the next subsection.

The decision making process of the scheduler requires the algorithm to decide if node n should be assigned to resource r. To the circuit designer, the answer is often, "it depends." There does not appear to be an algorithm that can deterministically produce the optimal scheduled result. The optimal result occurs when the model simulates at the highest rate possible on the hardware itself, an extremely difficult quantity to directly reason about. For example, the fewest number of clock cycles per iteration, or sample period, is one metric by which to judge performance. However, complex routing and multiplexors to achieve that might substantially increase the critical path, forcing a slower clock period. A balance has to be found between many different criteria when developing a schedule.

The scheduler is purposely absent of heuristics based on rules. This is simply because we do not know the rules, nor do we believe that these rules can be easily expressed. Instead, we treat the scheduler as a multicriteria optimization problem. As such, we evaluate all the

criteria that goes into a scheduling a resource such as compatible bit-widths, decimal point alignment, latency, successor and predecessor nodes, etc. These criteria are weighted and summed as a cost function. The lowest cost resource r is then assigned the node n.

We believe this approach is preferred over rules since the cost function enables tuning from the outside of metric weights. Different classes of models (isolated vs. population, homogeneous vs. heterogeneous, etc.) might require vastly different heuristics to perform suitably. This approach pulls customization of the scheduler from internal to the compiler into a series of numerical weightings.

6.6.2.1 Determining Compatible Resources

The expression graphs are scheduled in order that they appear in the list of *write* nodes, *i.e.* unsorted. We initially believed that larger trees should be scheduled prior to smaller trees, where the size of the tree was determined to be the longest latency path from a leaf node to the root node. This had the unexpected consequence of assigning all the resources prematurely. For example, for ten Hodgkin–Huxley models, the largest trees were the ones rooted at the write of V_{mem}. In this example, all the voltage expression graphs were scheduled leaving few resources available to the gate variables, m, h, and n. By leaving the graphs unsorted, each neuron is fully scheduled before moving onto the next, enabling a more fair distribution of resources per unique expression.

The graphs are traversed pre-order, such that the *read* nodes are scheduled prior to the *write* node. The earlier computed operation is always before the later. The rules for selecting compatible resources varied per node type, where a compatible resource consists of both a resource entry in the DRT and a time point. Multiple compatible resources can be possible for a given resource entry if it is compatible at different time points.

There are two criteria that must be true for a node to be compatible to be assigned to a particular resource. First, the operation type must either be identical or convertible. If a node is a multiply operation and the target is a multiply resource, than it is compatible. The scheduler also considers operations that can have dual-use functions, for example, an adder. Addition resources can readily be converted into dual-purpose addition/subtraction

Table 16: Example resource table showing six resources and ten cycles. Assigned nodes are depicted with the • symbol. The ? marks those resource/time pairs where a multiplication with operands available in cycle three can be scheduled.

		Cycle Time									
Resource	Operation	0	1	2	3	4	5	6	7	8	9
1	+	•	•		•	•		•			
2	+		•			•					
3	−			•		•		•			
4	×	•	•	•	?		•	?			
5	×	•	•		?	•	?				
6	×	•			•	•	?				

resources. Therefore an addition operation would be considered compatible with a subtraction resource, although not necessarily a good choice. The second criteria requires the latency of the node to be less than or equal to the latency of the resource. A multiplication that requires four cycles to compute a result should not be assigned to a resource that only allows two cycles. If it were, all operations would need to acquire the longer latency which could potentially disrupt a previously assigned node.

Multiple time points can be considered for a given resource. In the example schedule shown in Table 16, a multiply is to be scheduled in the resource table. Assigned resources/-time pairs are denoted with a • symbol. There are three compatible resources numbered $4-6$. Assuming the operands are generated no later than at time three, the question marks (?) denote those positions in the table that the operation can be scheduled. The algorithm chooses all available times directly following a previously assigned node if the times are greater than the minimal times. The algorithm also always returns the minimal time if available. In this case, a total of five resource/time pairs will be available to the scheduler as compatible resources.

Future enhancements can take into account additional convertible resources, such as comparison operations and subtracters. Additionally, commutativity of operations can also be considered such that the addition of $a+b$ can be considered along with $b+a$ as separate compatible resources. Other changes to operation analysis (see Section 6.3.2) can mitigate the benefits of this feature.

6.6.2.2 Scheduler Cost Function

Each compatible resources is evaluated for fitness. For example, in a model of 10 neurons, a multiplication operation that is identical in each neuron would be a good fit to share the same resource. The fixed-point bit-width would likely be identical, and if the predecessor and successor nodes were assigned to common respective resources, fan-in and fan-out between operations would be minimized. The challenge is in how to determine that the multiplication operation is the same across operations. Since no heuristic or rule is considered fool-proof, a weighted cost function is utilized to balance many, often competing factors.

The cost, C_j, of a given resource, r, is found as the sum of the weighted sum of N metrics as shown in Eq. (55). Each metric returns a fitness value such that lower values imply improved compatibility. Each metric is multiplied by a weighting that is user-adjustable via the DYNAMO command line. The current weightings are listed in Table 17. Metrics are prefixed by either *Inp*, *Out*, or *Op* for input-specific metrics, output-specific metrics, or general operation-specific metrics, respectively. Output- and operation-specfic metrics are computed once per compatible resource while the input-specific metrics are computed per operand. For a unary input operation, 14 metrics are summed, while a binary operation will have 20 metrics. The minimum cost C^* is returned from the function and the node is assigned the DRT entry with the most compatible resource.

Only operations and *write* nodes are scheduled. When the DRT is generated, literals and *read* nodes are pre-populated. These nodes are considered static across every cycle of the resulting pipeline. Therefore, the *read* nodes and literal nodes are pre-assigned to cycle zero.

$$\begin{aligned} C_r &= \sum_{i=1}^{N} W_i \cdot M_i \\ C^* &= \min(C_r) \quad \forall r \in R \end{aligned} \tag{55}$$

If a resource is not yet assigned, it is flagged as such for the metrics to evaluate. A resource does not acquire a fixed-point precision until the first node is assigned to it. This

Table 17: Listing of each metric and its default weighting

Metric	Weight
InpWidth	1
InpFrac	1
InpSign	1
InpType	1
InpUniqueSrcs	15
InpDly	1
OpCycleTime	4
OpScheduled	1
OpLat	10
OpType	1
OpLimRsrcs	0
OpCorrelation	5
OutNumSinks	1
OutSinkType	1

way, the scheduler has maximum flexibility to initially populate a resource at any bit-width. A special *isAssigned* flag is propagated to the metrics such that special rules take hold. If additionally scheduled resources require more precision, the resource is updated to reflect the increase. Convertible operations also dynamically change along with assignment. The adjustment of the resource table during scheduling gives the table its dynamic distinction.

InpWidth This metric compares the total number of bits in the predecessor node to the corresponding input pin of the resource. The ideal case occurs when both widths are equal. For that case, there is no cost. When the predecessor node bit-width is less than the resource, the resource is not fully utilized. The cost in that scenario is equal to the number of bits of difference. When the resource must expand by, for example, n bits, to accommodate the node, this action is discouraged by assigning a cost of $2n$. If the resource has not been assigned, there is no cost for the first assignment.

InpFrac Similar to the **InpWidth** metric, this metric charges no cost of an unassigned resource or a resource with an identical number of fraction bits. Additionally, like the **InpWidth** metric, there is a unit cost for each fraction bit that is not utilized in the resource and a two unit cost for each bit that the resource must expand to

accommodate the node.

InpSign This metric compares the signedness of the inputs to the resource with the inputs to the node. It is not ideal to use a signed resource when an unsigned resource will suffice and vice-versa. Therefore, there is a unit cost for dissimilar signedness and no cost for the same signedness or for an unassigned resource.

InpType This metric assigns a cost if the predecessor operations are dissimilar. For identical operations or if the resource is not assigned, there is no cost. If any of the predecessor resources are of the same type, but not identical, there is a unit cost. This would occur if, for example, a *read* node has been assigned to an input and the current node's input is also a *read* node. Predecessors that are convertible operations are additionally assigned a one unit cost. Completely dissimilar inputs are assigned a cost of two. For example, if the current node is a multiply, the metric is evaluating the first operand, and if the multiply from $(a_1 + b_1) \cdot c_1$ is already assigned to the resource, than a compatible node with structure $(a_2 + b_2) \cdot c_2$ will have no cost, $(a_2 - b_2) \cdot c_2$ will have one unit cost, and $(a_2 \cdot b_2) \cdot c_2$ will have a cost of two units. This metric is important if there is a significant amount of regular structure in expressions.

InpUniqueSrcs This metric counts the number of sources presently assigned to a node that are from unique resources. In a perfectly scheduled implementation, an input would only have one source despite being active throughout the pipeline. If the additional source is unique, a larger multiplexer is required to choose the appropriate input. In the Virtex architecture, one LUT is required for a 2-input multiplexer, one slice for up to a 4-input multiplexer, two slices for an 8-input multiplexer, 1 configurable logic block (CLB) for a 9- to 16-input multiplexer, and 2 CLBs for multiplexors between 17 and 32 inputs. Because the additional resources do not scale linearly, the cost for additional unique sources parallels the increases in area. When the input has already been assigned or the resource has yet to be assigned, the multiplexer does not need to grow or be created and thus the cost is zero. For the second unique input, a multiplexer is required and the cost is 4 units. The third input have a cost of only

2 units due to the still small size of the multiplexer. For the fifth inputs, the cost increases again to 4 units since the size of the multiplier is now getting relatively large. The ninth and seventeenth input cost 8 units and 16 units respectively due to the excessive size of the multiplexers. When the multiplexer resources do not change, the cost is one unit for the fourth, sixth, seventh, eighth, tenth, etc. inputs. This metric is exponential in its cost to discourage multiplexers from being used, instead allowing deep pipelines to emerge.

InpDly Similar to the **InpUniqueSrcs** metric, this metric is designed to discourage the use of delays whenever possible. If the scheduling of a node at a particular time point requires a long delay from the predecessor node, this metric will assign a high weight to that node. As in the previous metric, increases in cost are not linear and instead follow increases in area utilization. Delays are built using SRL16 primitives in the Xilinx architecture which utilize the LUTs in each slice[2]. These are shift registers that are up to 16 elements deep. SRL16 primitives require a register at the head such that a 17-cycle latency shift register requires half the primitives of a slice. A 33-cycle latency shift register can be built in a single slice. If the resource is already assigned with the same source, any delay can be reused, yielding zero cost. The first register required has a cost of three units. The first SRL16 required adds an additional three units, with one unit of cost per additional delay. After 17-cycles of delay, the second SRL16 costs an additional eight units, with now two units of cost per additional delay. Even though no additional hardware is required for certain increases in delay, a small cost is incremented to keep the schedule as packed as possible. Scheduler optimizations can correct long delays as will be explained in the following section.

OpCycleTime Ultimately, performance is tied to the sample period of the overall datapath generated. This metric works to reduce the sample-time by preferentially selecting resources and cycles that occur earlier. The cost returned becomes exponentially larger as the cycle time increases. The relationship, determined empirically, is set to

[2]Half of available slices, the *SLICEM* primitives, are capable of becoming an SRL16

$cost = \text{round}(0.2 \cdot cycle^{1.5})$.

OpScheduled All metrics up to now do not penalize for using a resource that has not been assigned. If a resource has not been assigned, it can be used in later scheduler iterations when another operation is not a good fit for any other resource. This metric assigns a unit cost if the resource is not yet assigned. For highly regular small footprint models, this metric can be set to a low weighting to encourage heavy use of available parallel resources. For large, heterogeneous models, a high weighting can aid late-to-schedule trees by preserving untouched resources.

OpLat This metric compares the latency of the resource with that of the node. The compatible resources list has already been constrained such that only resources with equal or longer latencies were included. A cost of one unit for the difference in the latency between the resource and the node is returned by this metric. Equal latencies return a zero cost.

OpType This metric compares the operation type of the resource with the node. If they are identical, the metric returns a cost of zero. If the operation types are different, which would occur only for a convertible operation (ADD/SUB), a unit cost is returned. A high weight on this metric can effectively stop these convertible operations from forming.

OpLimRsrcs This metric is not currently implemented but is kept as a place-holder for future enhancements. When the number of resources of a given type that are not assigned is dwindling and when there are classes of nodes that have not been scheduled, this metric should increase the weight of assigning a new resource. This is somewhat redundant to the **OpScheduled** metric with more global knowledge on the structure of the trees that still must be scheduled. Given that this global information is required, this metric will have to wait until the scheduler is enhanced to produce this data.

OpCorrelation This metric utilizes the correlation matrix generated in a previous step and described in Section 6.4.4. The correlations are determined between the node to be

scheduled and the previously assigned nodes. The maximum of the these correlations is scaled such that the largest and smallest correlation in the table is equal to a cost of ten for the least correlated pair of nodes and zero for the most correlated pair of nodes. This value between zero and ten becomes the cost of this metric. This is the one metric that takes into consideration global properties of the model.

OutNumSinks While the **InpUniqueSrcs** metric attempts to reduce fan-in to operations, this metric reduces fan-out. Extensive fan-out from an operation can lead to extensive wire delays reducing performance. Every unique successor node to the current node to be scheduled is counted to have a cost of two units. This is summed with the number of successors of the already assigned nodes. However, if there are no additional sinks, there is no additional cost. When comparing successor nodes, to be exact, the successor node and the pin has to be unique. The pin is not considered in this computation based on the assumption that the pins of a successive resource will be co-localized and contribute little to overall delay.

OutSinkType Perhaps more significant than the previous metric, this output sink type metric assigns a high cost when the successive node is of a different type than the previously assigned nodes' successors. For example, if all of the assigned nodes drive a multiplier resource, then an additional node that drives the write of a state would be return a high cost. The cost is 4 units for each additional type of sink.

Following the scheduling of all nodes, the data structure based around the DRT is converted into a new data structure for easier analysis. While the DRT was based around the resources and a list of assignments, the new data structure, termed the schedule is based on a matrix with one axis representing the resource and the other axis representing the cycle time.

6.6.3 Scheduler Analysis

The scheduler data structure built following the scheduling of each node of the run graph now undergoes a series of optimizations. In this new matrix data structure, movement of

nodes is trivial and does not require the generation of a new data structure after every change. The schedule undergoes four optimization passes, with additional passes planned. After each optimization pass, the following statistics are gathered across the schedule: number of entries, percent utilized, pipeline depth, total delays, and percent resources over utilized.

The number of entries is simply calculated as the number of assignments across the entire resource table. This includes every *read* node, *write* node, literal and operation in the graph. Percent utilized describes the portion of the schedule's total slots for operations that are assigned. Since *read* nodes, *write* nodes, and literal nodes only exist once by definition, they are not included in this calculation. A percent utilization of 100% implies a fully utilized resource schedule. Models with a high degree of regularity tend to have a larger percent utilized than models with a lack of regularity. For example, the Booth, Rinzel, and Kiehn model in one execution of DYNAMO had a pipeline depth of 71 cycles and a utilization of 2%. If instead, 40 Hodgkin–Huxley models are implemented in DYNAMO, the resulting pipeline depth is 62 cycles with a utilization of 8%. This is still not significant, and a source of future expected performance enhancements.

When the operands of a particular operation are generated prior to its use, delays need to be added for synchronization. Total delays are computed as the sum of the number of cycles of latency per delay required by the schedule. A smaller number of delays constitutes a "tighter" schedule. The currently implemented optimizations work to reduce the number of delays in the system, which directly affects area consumption.

If an input of a resource is driven from multiple sources for use in different cycles of the schedule, a multiplexer is required to properly route the appropriate input at the appropriate time. In a well implemented schedule, few multiplexers should be required as resources are reused across the regular structures. The percent resources over utilized statistic returns the percentage of resource inputs that require a multiplexer.

Scheduler optimization pass #1 reassigns and duplicates all parameter, state, and literal reads such that no delays are required for synchronization. Since state reads (which are ultimately implemented as registers) and literals are static throughout the schedule, there

is no requirement for delays. This optimization simplifies the netlister by not having to differentiate between inputs that are static and inputs that are dynamic. For parameters, this optimization is required for optimization pass #4.

Scheduler optimization pass #2 is tasked with reducing unnecessary delays between operations within the data-path. Starting with the latest cycle and working towards cycle zero, scheduled nodes are evaluated for delays from their predecessor nodes. If a predecessor node can be shifted forward in the schedule to reduce a delay taking into account other sinks for that node, the predecessor is reassigned to a later clock cycle. As this algorithm iterates through to cycle zero, the nodes undergo a forward in time shift in the schedule, freeing up open schedule slots in the early cycles. Following this, optimization pass #3 repeats the optimization in pass #1 to reflect the new positions of many operations. The combination of optimizations $1 - 3$ typically remove greater than 90% of the delays in the schedule.

The final optimization pass reassigns all parameters that source a pin of an resource into a special parameter resource. Multiple parameters sourcing a pin can be handled more efficiently by utilizing random-access memory (RAM) elements instead of registers and multiplexers. An additional pass can be done for literals in the same manner, however, modern synthesis tools are often able to reduce the logic by constant folding, so that optimization is not performed.

Additional unimplemented optimizations can effectively wrap operations performed later in the schedule to early cycles if there are no conflicts. This optimization can bring the scheduled back-end closer to the performance of manually-generated fully pipelined neural models. The utilization should increase dramatically as the pipeline depth decreases. Further work at this stage can also reassign reads and writes of states into larger granular memory elements such as distributed RAM and block RAM. This will have the benefit of increasing the size of models that can be implemented on the FPGA.

6.6.4 Generic Netlister

After the final optimizations have been completed, the scheduler undergoes a transformation into an internal netlist format. A netlist is a module that contains a listing of instantiations

of primitive operations (the base operations in the hardware) and other modules, nets describing the connections between the operations, and ports for top-level connections. Since a netlist can instantiate modules which are netlists, a hierarchical structure can readily be generated.

The internal netlist format is an intermediate representation and is not tied to any particular hardware description language. A common set of features make translation into Verilog, VHDL, or System Generator blocks trivial. As an intermediate representation, all hardware details are present, including bit widths on the ports of all modules and primitives, latencies for every block, and quantized initial values for all registers and memory structures.

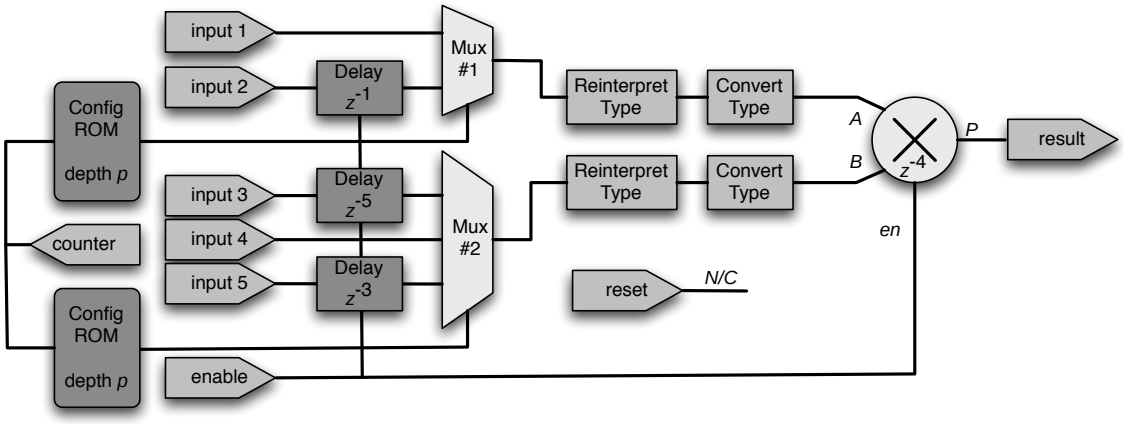
The netlister creates a top-level module consisting of a run block containing the data-path, a parameter block for tuning and setting of parameters for input into the data-path, an input block to receive streaming inputs for simulations, an output block to capture streaming data from the run block for digital transfer, and associated control logic. Each of these blocks are built independently and connected in the top-level module.

The control logic includes a cycle counter, enable logic, and a termination condition. First, the counter is count limited from 0 to $d - 1$, where d is the pipeline depth. This cycle counter is accessible to all blocks of the top-level. The enable logic logically ANDs an external enable set by the interfacing tools, a ready flag from the input block, and a ready flag from the output block. Finally, the termination condition utilizes a comparator to compare the required `STATE t` with an externally set stop time and an SR-register (set–reset flip–flop) to indicate a stopping condition has been met. Input ports include an external enable, an external reset, and the stop time. The termination condition and debug ports of output variables are designated as output ports.

6.6.4.1 Run Block

The run block contains the data-path, or the core execution unit, of the model. Generation of the run block requires both the DRT and the schedule data structures. The DRT contains the detailed information of each resource while the schedule is a convenient representation for manipulating operations. Since node assignments do not move from one operation to

Figure 38: Schematic illustration of a typical operation. The operation in this example is a 4-cycle latency multiplier where the first input is driven by a 2-input multiplexer and the second input is driven by a 3-input multiplexer. Ports 2, 3, and 5 have synchronization delays with specified latencies. All clocked elements are tied to a global enable signal. A synchronous reset signal is included in every component but not required in this example.



another, the operation portion of the DRT remains synchronized with the schedule.

The generation algorithm iterates across each resource in the DRT. The resource is converted into a module that incorporates all necessary multiplexing, delays, control logic, type conversion, etc. that is operation specific. There is a global counter signal available to all resources that counts through the cycles for use by any of the control logic. An additional clock enable and reset signal (both synchronous) are available to each operation module. A typical operation is shown in Figure 38.

In this example, a multiply operation is shown with all its associated circuitry. The multiply is configured with a precision for its input and output pins. To ensure inputs have the appropriate precision, two bit manipulation blocks are provided in series, a reinterpret block that casts the input with the appropriate signedness and fractional point while the convert block adjusts the number of bits by either truncating or sign-extending the inputs. These conversion blocks are added only when necessary to generate the appropriate input precision.

The conversion blocks are optionally driven by multiplexers when the source of input must change at different cycles. Both operands do not need to have multiplexers with the same number of inputs. In this case, the first operand has two possible sources while the

second operand has three unique sources. Some of the sources in this example require delays for synchronization. Control logic, in the form of select signals for each multiplexer, utilizes a read-only memory (ROM) addressed by the cycle counter. If the operation is convertible, an additional control ROM is generated to produce a function code.

All blocks with internal registers use a global enable signal to control the execution of the simulation. For protocols to be executed on the hardware and for data transfer with asynchronous interfaces, an enable signal allows the pausing or throttling of the model. For data to not be corrupted after a pause, all registers within the data-path must become quiescent. In the above example, the multiplier and the delays are all clocked and therefore have an enable signal.

For other blocks such as *write* nodes, a control ROM addressed by the cycle counter produces a write enable signal for the register storing the state. That write enable is combined with the global enable by a logical AND to produce the register enable signal. State *read* nodes act as a pass-through from *write* nodes while parameter reads are directly tied to input ports. Literals are implemented simply as constant blocks in the generic netlist format.

The port names for each operation module are built off unique resource ID numbers assigned in the DRT. The port names are automatically wired together by like names with a built-in function. The inputs of the run-block consist of each parameter resource and a global enable and reset signal. The outputs include all variables specified as such in the DML.

6.6.4.2 Parameter Block

The parameter block provides the capability for tuning and distributing the parameters to the run block. The structure has many similarities to the parameter block explained in Chapter 5. The parameters act as a memory mapped structure to the software driver. In the shared memory system, each parameter is stored as a 32-bit unsigned integer equivalent. On the FPGA, this parameter memory is implemented as a dual-port RAM where one port is accessible to the hardware while the other port is accessible to the software interface.

The parameters are not read by the run block directly from the parameter memory. Instead, the parameters are either loaded from registers or from a second tier of smaller RAM elements. When a single parameter is utilized by a operation throughout the entire sample period, a register is used to store the parameter value. When multiple parameters are required at various cycles, a RAM is configured with each parameter value. This RAM is addressed by an additional control logic RAM to select the appropriate address as a function of the cycle counter.

The registers and multi-parameter RAMs are updated continuously and asynchronously to the run block. The parameter memory is addressed by a counter that cycles across the entire depth of the RAM. A shift register of the same delay with a single propagating high bit provides a rotating enable signal to each register and multi-parameter RAM. This enable is used as a write enable signal to update the contents of the RAM. Additional logic is required to make this viable, including counters to generate the write addresses that are controlled by SR flip-flops, and conversion/reinterpretation blocks to produce the appropriate precision for each parameter. Registers and RAMs are continuously updated regardless of whether or not the values have changed.

While it might be simpler to update parameter values only after a change, limitations in System Generator motived our design decisions. System Generator does not allow direct access to the write enable and address lines on the driver portion of the parameter dual-port memory. Overall, very little area is required to implement the additional logic, relative to the remainder of the parameter system and data-path.

6.6.4.3 Input/Output Blocks

The primary means for data transfer in this iteration of the DYNAMO compiler is via high-speed digital interfaces. The current development board, the ML402, offers a co-simulation mode within System Generator whereby digital data can be block transferred via a custom protocol over Gigabit Ethernet. Simply, this digital interface is built from a single read FIFO (First-In, First-Out) for inputs and a write FIFO for outputs. These digital interfaces allow inputs and outputs to be selectively enabled or disabled. Additionally, resampling is

supported to reduce the data transfer requirements.

A FIFO is a memory element for implementing a queue. There is an input port and an output port where the input port is written to upon assertion of a write enable and the output port is read from when a read enable is asserted. Reading and writing of the FIFO is independent and can be driven off different clocks. However, a FIFO cannot be read when empty nor written to when full. The status signals *empty* and *full* are available from FIFOs to prevent those conditions from occurring.

Since a computer is not a real-time device, input streams can not be guaranteed to be available in the input FIFO for the data-path. Similarly, it can not be expected that the output FIFO will never become full. Therefore, the FIFO *empty* and *full* flags are important in providing an asynchronous interface to a general-purpose computer. An empty or full condition on the respective FIFOs tied directly into the global system enable, such that on assertion, the entire data-path stalls until a FIFO is either filled or emptied as needed. This can slow down an FPGA simulation, but it ensures that all data is transfered without corruption.

Performance is improved by upsampling inputs and downsampling outputs. Often, the time-step required for integration is smaller than that needed for processing. *In vivo* and *in vitro* electrophysiology experiments often sample at no more than 25 kHz (40 μs) while the time-step is often 10 μs or less. This discrepancy can accommodate up/down sample ratios of 10 or more.

The frequency content of the input and output is often orders of magnitude less than the frequencies required for integration. Therefore, simple resampling circuits can be utilized instead of complex and resource intensive interpolation and decimation filters. In practice, experimental electrophysiology protocols often make use of resampling over more complex methods.

Multiple signals are transferred per FIFO by interleaving. Interleaving, or time-division multiplexing, allocates periodic time slots to each input and output. Since a FIFO block size and granularity are one word, an arbitrary number of slots can be dynamically generated and extracted. This is in contrast to a memory with a set depth. For example, if 16 streams

are stored in a 1024 element RAM, there will be 64 elements per stream at any given time. If instead there were 17 streams, there will not be a uniform number of elements per stream, either requiring offsets to properly address or unusable elements in the RAM. Since a FIFO has no beginning or end, it is the ideal structure for variable length input streams.

Both inputs and outputs can be selectively activated through the user interface. A reduction in the number of inputs or outputs can have the affect of improving performance by reducing transfer overhead. Outputs can selectively be set to be streamed via the FIFO. This is performed by setting a one-bit enable bit in an additional *Shared Memory* structure. This memory has one element for every output in the system. For System Generator (v8.2) targeted models, undocumented internal restrictions require *Shared Memory* blocks must be at least 3 elements deep and no less than 2-bits wide.

All variables declared as "outputtable" are accessible to the output block from the run block. These n outputs are resampled at the output sample rate. The total number of outputs, N, possible in this architecture is equal to the product of the pipeline depth, d, and the output sample ratio, R. The most significant 16-bits (user configurable by a compiler flag) of each output is extracted per output and fed into an n-input multiplexer. A count-limited counter from 0 to $N-1$ with two comparators and SR-registers produces control signals to signify the counter is between 0 and $n-1$. This in-range count provides both the addressing for the output select *Shared Memory* block and for the select line in the multiplexer. The data out from the output select memory becomes a write enable for the output FIFO while the multiplexer provides the data.

Inputs are similarly processed with an added complication. While an output can be configured to not have a sink to drive, an input must have a source. The DML defines a boolean field, `Inp.isReady`, such that the modeler can define the behavior of the simulation with and without the presence of a particular input. This input feature not only provides the option for default values, but also for mode changes based on the particular command input. For example, a neural model can selectively be in a current-clamp mode or a voltage-clamp mode based on the activation of current-command or voltage-command inputs.

The active inputs are serially loaded into the FIFO. An identical counter, comparator,

and SR-register structure as described above is used to bin each input read from the FIFO. The FIFO's read enable pin is asserted according to the output of the corresponding input-select *Shared Memory* block. The output of the FIFO drives the input of a demultiplexer where the select line is the input-select memory block output. Each output of this demuliplexer is registered and available to the run block. Each input-select is also registered and accessible to the run block as a boolean variable.

6.6.5 Area Estimation

After the full implementation, including the run block, parameter block, input block, and output block, is netlisted, an area estimation algorithm verifies that the design fits within the constraints of the target device. The algorithm iterates across each component in the netlist, ignoring ports and nets. The total area returned is intended only to be an approximation, the final area is not unknown until after the design is placed by the implementation toolkit, in this case Xilinx ISE v8.2.

The resources of the FPGA are split into four bins: *luts*, *ffs*, *rams*, *mults*. A *lut* is a 4-input function generators that can implement a 16 element RAM, a 16-stage shift register, a 1-bit addition/subtraction (in theory but not in practice), a 2:1 multiplexer, or any other 2-input logic function. A *ff* is a single-bit register used either for storage of state information or internal stages in a pipelined operation. Each *RAM* is an 18 kbit memory element while the *mult* bin refers to the embedded 18x18-bit signed multipliers on chip.

These bins are loosely connected and are a simplification of the Xilinx architecture. In a Virtex-4 device, resources are split into three bins: configurable logic blocks (CLBs), XtremeDSP blocks, and Block RAMs[93]. Each CLB includes two *SLICEM* and two *SLICEL* blocks. Within each slice there are two lookup-tables and two registers which correlate to the *luts* and *ffs* bins above. The XtremeDSP block contains a multiply–accumulate (MAC) operation with built-in multiplexers and registers, but roughly correlates to the *mults* bin. The Block RAM is functionally identical to the *rams* bin.

While there are similarities between the bins used for area estimation and the actual logic resources available in the Xilinx architecture, there are differences that partially account

Table 18: Compares the performance estimation of DYNAMO with the built-in performance estimation tool in System Generator and the post-MAP phase following synthesis.

Model	Bin	DYNAMO Est.	SysGen Est.	Map
FN	*luts*	688	408	2375
	ffs	1501	1051	2533
	rams	3	0	15
	mults	10	10	10
HH	*luts*	4418	3055	3315
	ffs	9232	9334	5441
	rams	15	10	27
	mults	101	101	98
BRK	*luts*	10933	8216	9716
	ffs	17445	16134	8866
	rams	11	8	23
	mults	140	138	123

for discrepancies. For example, related logic is often grouped in the same slices and CLBs. Unrelated logic is not generally combined in the same slice unless resource limitations require it. No checking is done to verify related logic is grouped together. Another difference is that only lookup-tables in the *SLICEM* primitive support the alternate RAM or shift-register configuration while all *luts* can be configured as such. The additional resources within the XtremeDSP blocks are ignored with respect to area estimation. The synthesis tool might group an addition and a multiplication together, but it is not done automatically as a MAC is not an primitive operation in DYNAMO. Finally, co-simulation within System Generator adds overhead that is not included in the estimate as it is not generated by DYNAMO.

For certain operations, great care was taken to accurately represent the resource utilization. We performed parametric tests across all combinations of bit-widths from 2- to 64-bits for addition/subtraction blocks and multiplication blocks. The results in Figure 39 show that the *luts* and *ffs* utilization grow linear with the maximum bit-width, b of any input. The plots were curve-fitted and found to be

$$\begin{aligned} luts &= \lceil 1.500 \cdot b + 2.754 \rceil \\ ffs &= \lceil 2.000 \cdot b + 4.000 \rceil \end{aligned} \tag{56}$$

with an R^2 value of 0.99 or greater. The number of *ffs* is shown for a latency of one.

For each additional cycle of latency required, an output bit-width flip-flops are added to the estimate. This formula is used for all adders, subtracters, combined adder/subtracters, relational operators, and negation.

Multipliers are implemented differently depending on the target architecture. For a Virtex-4 architecture, arbitrary size multipliers are built exclusively in XtremeDSP blocks. The number of XtremeDSP blocks is a function of the bit-width, b, and sign, s, of each input.

$$XtremeDSP = \left\lceil \frac{b_{\mathrm{A}} - s_{\mathrm{A}}}{17} \right\rceil \cdot \left\lceil \frac{b_{\mathrm{B}} - s_{\mathrm{B}}}{17} \right\rceil \tag{57}$$

This expression is found empirically from executing a place and route on every combination of inputs as shown in Figure 40(a). Multiplications on an FPGA are done by summing partial products, similar to the way multiplication is performed manually. In a Virtex-4, the partial products utilize the internal adders in the XtremeDSP blocks. Registers utilize a portion of the logic space and is equal to the sum of the two input widths ($\mathit{ffs} = b_{\mathrm{A}} + b_{\mathrm{B}}$) as shown in Figure 40(b).

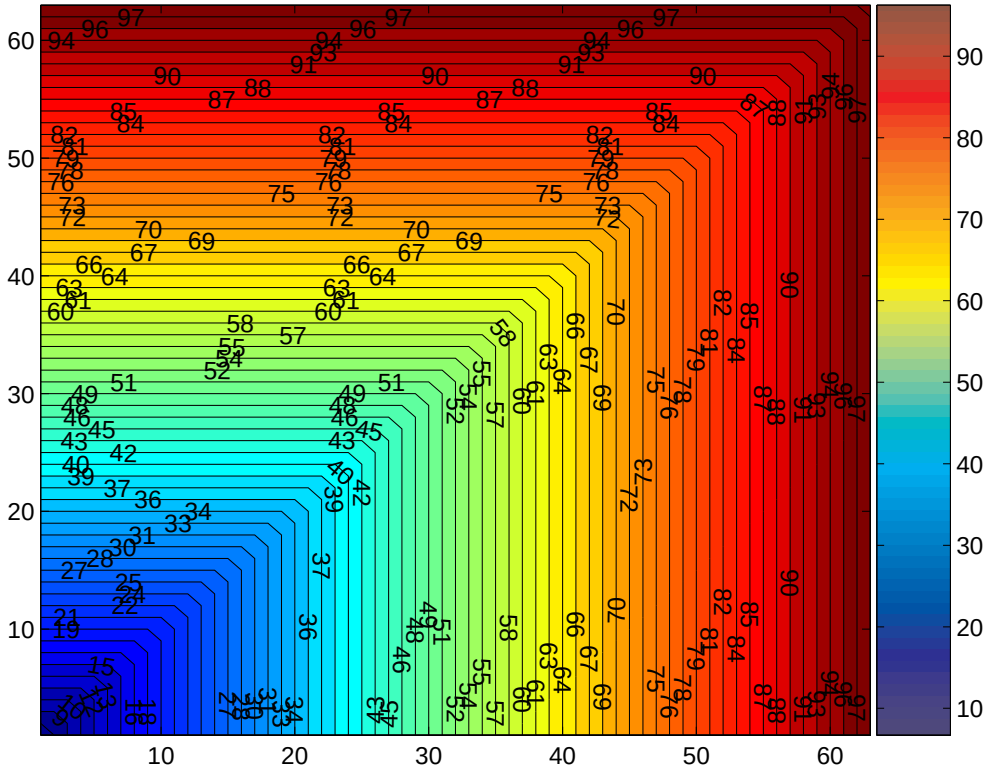
In a Virtex-II architecture, the adders used in multiplication are instead built out of logic elements on the device. The algorithms for determining the numbers and sizes of these partial adders is based on schematics from Xilinx[93]. The inner product adders,

$$N_{inner\_adds} = \left\lceil \frac{b_{\mathrm{A}} - s_{\mathrm{A}}}{17} - 1 \right\rceil \cdot \left\lceil \frac{b_{\mathrm{B}} - s_{\mathrm{B}}}{17} - 1 \right\rceil, \tag{58}$$

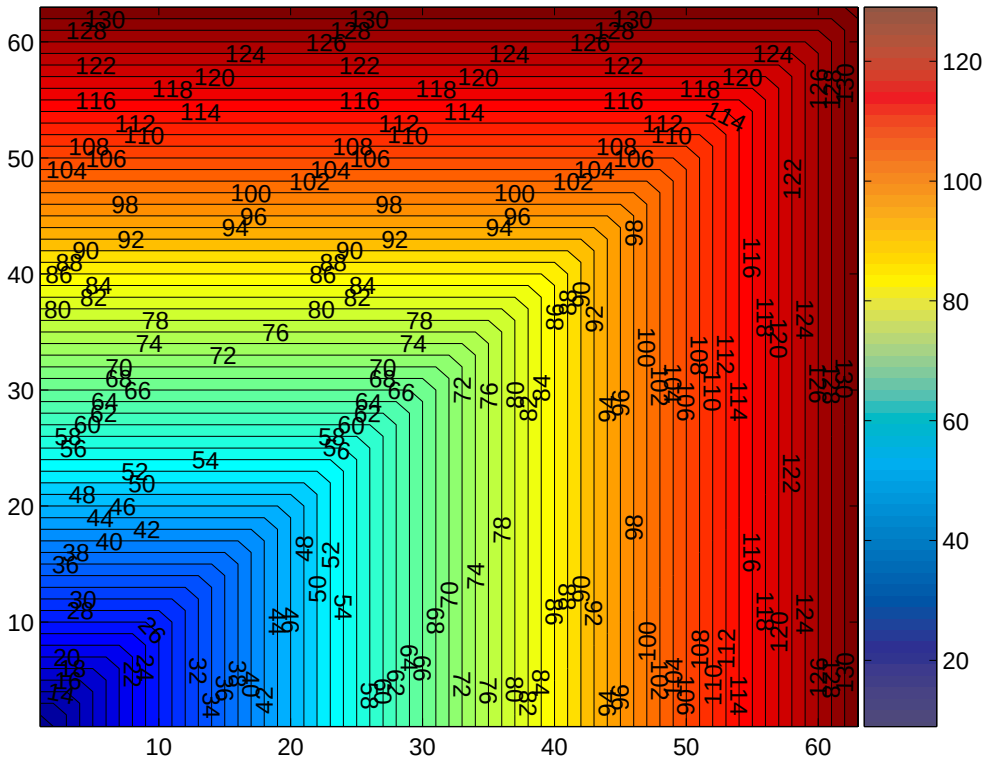
are built as 35-bit output width adders with inputs of 35-bits each. An additional adder large adder is built for the final outputs of each of the inner additions. This adder, the outer-adder, has a bit-width equal to the output-width of the multiplier.

Logical operations such as AND, OR, XOR, etc., each consume one *lut* per bit while bit-width multiplied by the latency yields the number of *ffs* consumed. Shift operations consume no *luts* but require *ffs* at the rate of bit-width times latency. Generally, shift operations should have no latency as they consume no logic. Multiplexers use *luts* at the rate of

$$luts = b \cdot \left\lceil \frac{N}{2} \right\rceil \tag{59}$$

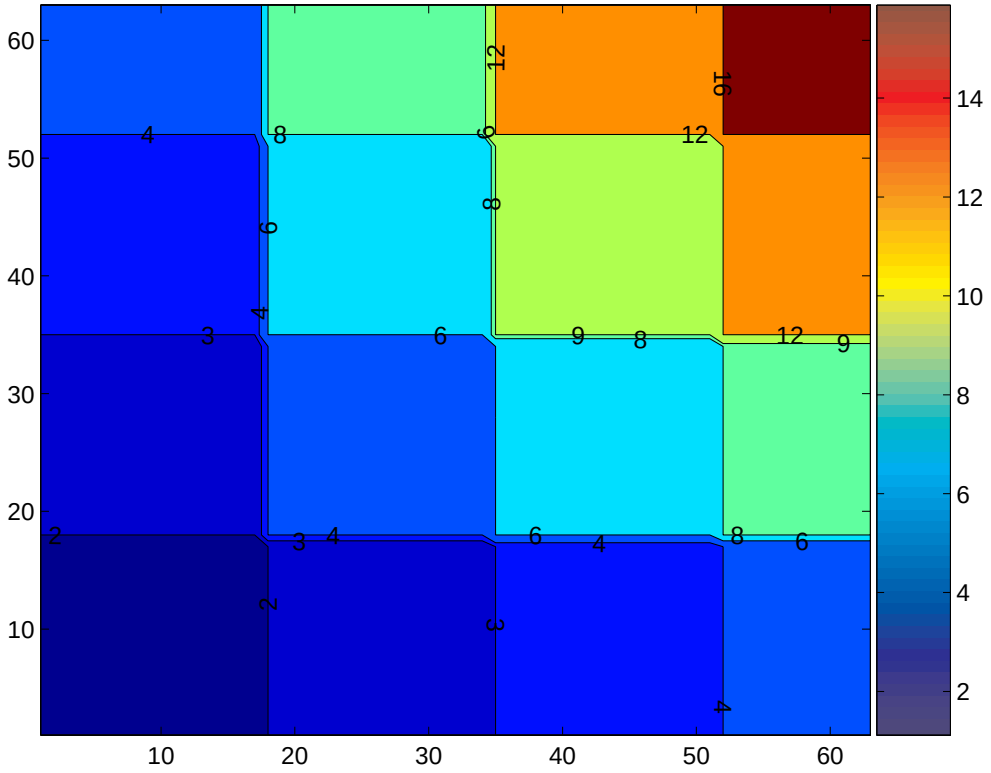


(a) Lookup-table usage

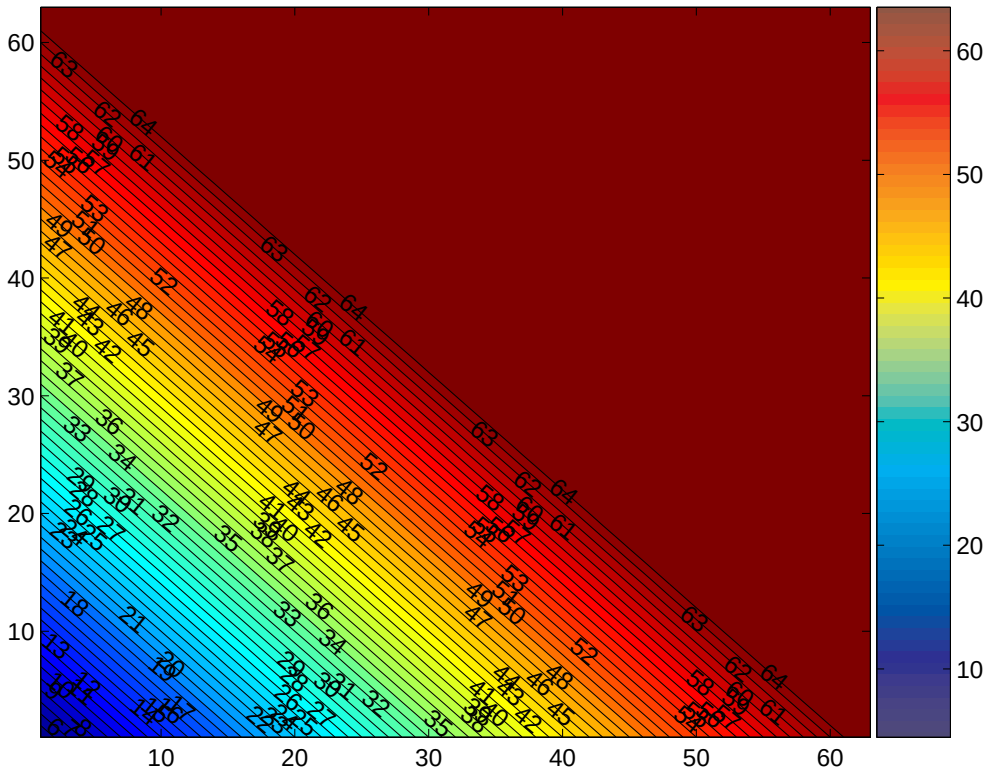


(b) Flip-flop usage

Figure 39: Adder/Subtracter contour plots showing the post place and route utilization of a) 4-input lookup-tables and b) flip-flops as a function of the bit-widths of the operands. Each combination was built by Core Generator and then synthesized in a minimal Verilog wrapper.



(a) XtremeDSP usage



(b) Flip-flop usage

Figure 40: Multiplier contour plots showing the post place and route utilization of a) XtremeDSP blocks and b) flip-flops as a function of the bit-widths of the operands. Each combination was built by Core Generator and then synthesized in a minimal Verilog wrapper.

where b is the output bit-width and N is the number of inputs. The *ffs* utilized are calculated identically as the above logical operations.

Registers consume the number of *ffs* equal to its bit-width. Set–reset flip-flops are by definition one-bit and consume only one *ff*. A down-sampling operation requires a register clocked at the lower clock rate. Therefore the consumption of a down-sampling operation is equal to that of a register. An up-sampling operation can be interpreted as being clocked at a higher rate, so therefore, it requires no resources. Similarly, type conversion and reinterpretation blocks are estimated to require no logic. Delays utilize the 16 element deep shift-register primitives (SRL16) for the excess delays greater than one. The total *luts* required is equal to

$$luts = b \cdot \left\lceil \frac{l-1}{16} \right\rceil \tag{60}$$

where b is the bit-width and l is the latency. A delay always utilizes one *ff* per output bit. Counters are approximated as a register of an equal number of bits and a logical operation of an equal number of bits with no latency.

Memory on a Virtex FPGA is split into distributed and block RAM. Distributed memory utilizes selectRAM on the FPGA, using one *lut* for each 16-bits of single-port RAM or 8-bits of double-port RAM. The utilization for a single port RAM of depth d, output bit-width b, and latency l is

$$\begin{aligned} luts &= b \cdot \left\lceil \frac{d}{16} \right\rceil \\ ffs &= b \cdot l. \end{aligned} \tag{61}$$

A dual-port RAM uses a similar calculation except for twice the number of *luts*.

Block RAMs are used for large tables of values, shared memories, or FIFOs and can be configured in the following ratios: 512×36, $1k \times 18$, $2k \times 8$, $4k \times 4$, $8k \times 2$, and $16k \times 1$. Multiple RAMs can be cascaded to create larger memories. For bit-widths greater than 18, a *mult* resource is required to obtain the additional routing resources. For the Virtex-4 architecture, the first cycle of latency is built into the block RAM while additional cycles each require *b-ffs*. A small number of *luts* are used for cascading and are ignored for simplicity.

If the total memory utilized is greater than the resources available on the target device, DYNAMO clears the entire DRT and schedule and begins the process again. In the next iteration, the DRT scaling factor is reduced to generate a smaller DRT. If the number of GAPs are exceeded, the initial GAP count is reduced by 40%. If MULTs are exceeded, this count is reduced by 5%. Finally, the RAM count is reduced by 20%, if exceeded. Up to five iterations of the scheduler are executed after which the compiler fails returning an error.

It is possible that the implemented model can not fit within the target device, *i.e.* the design was estimated incorrectly. In this case, the user can readily determine the final DRT scaling factors and adjust those factors from the command line, forcing the compiler to schedule with less resources. An iterative approach in that manner should enable most models to pass through unmodified to the FPGA.

6.6.6 System Generator Output

The resulting netlist that is estimated to fit within the target device is passed onto the last stage of the compiler, the generation of System Generator output. System Generator, as a blockset with Simulink, can be programmatically generated with `add_block` and `add_line` MATLAB commands. The System Generator output code generates a MATLAB script to build component-by-component and net-by-net the model.

The MATLAB script builds a module (over-writing if one already exists) of the model with all necessary blocks for generation. The includes a System Generator token in which the settings for the target architecture are predefined, as well as default inputs for the enable, reset, and stop condition signal. At this point, debug outputs are defined for all "outputtable" variables. This will be removed in a later version when all data is transferred through the FIFOs. Using a call-back defined within the System Generator token or a MATLAB command, synthesis and implementation scripts are automatically built by the tool, initiating the compilation sequence. Depending on the size of the model, performance of the machine, and available memory, this process usually takes between 30 minutes and 2 hours. A library is generated which includes a co-simulation Simulink block which then is manually copied into a user-developed test-bench with the corresponding FIFO blocks

and basic infrastructure to control the model. It is hoped that soon this entire process is automated, but in the meantime, the current implementation serves our purpose better as a debugging platform.

The build script as the final step defines a DIF file (see Section 6.5.1.1) for use with the *dynmodel* interface. This provides all the pertinent information on the system, including parameters, inputs, outputs, and all necessary fixed-point precisions and ranges. The time-step is also included to generate appropriate input sequences for the model. Identifiers for each of the shared-memory/FIFO structures is provided in the DIF file as well.

System Generator provides two means for transferring data. *Shared Memory* blocks implemented in the hardware use unlocked memory. These memory interfaces provide random access and transfer one word at a time, similar to an I/O interface. A MATLAB class called "shared_memory" will instantiate an object when passed the unique string identifier for the memory. Memory can be modified by indexing the object, which is zero-indexed unlike other vectors in MATLAB. The FIFO interfaces have two means for data transfer. Corresponding FIFO blocks are available with all the same controls available as in the hardware, including empty flags, full percentages, read/write enables, etc. While these blocks make for simple interfacing, they transfer data slowly, at a rate of one word at a time. A vector based transfer method is also available to transfer frames of data. For example, in a 1023 word FIFO (always one less element available than the size of underlying memory block), a block size of 600 words might provide a good balance between a large block transfer and minimal pipeline stalling if the FIFOs become full/empty. Unfortunately, these vector-based transfer methods are not particularly flexible and offer many challenges. Despite difficulties, we have used them for sustained transfer rates across a PCI interface of 8 MBps.

6.7 Results

This section summarizes the results of the third-generation DYNAMO compiler utilizing the cost-function based hardware scheduler. From a performance perspective, we define positive

results to include both a significant increase in simulation throughput and an architecturally-reasonable outcome. This application requires these metric terms to be notably vague. The DYNAMO compiler has already undergone drastic revisions with more planned. In each iteration, we have witnessed a performance increase and expose new ways of enhancing the capabilities and/or performance of the compiler. These results are a snapshot into the current workings of the compiler and do not reflect previously witnessed performance nor should they fully define future performance.

The first requirement was that the simulation must undergo a significant throughput enhancement. We are defining a significant performance enhancement to be at least an order of magnitude. We used the NEURON simulator[39] v5.8 distributed by Yale University as our control platform. It is the standard neural simulation package and with over 600 publications utilizing the tool at the time of this writing. The NEURON package was executed on a Pentium 4, 3.2 GHz with 2 GB RAM running Redhat Enterprise Linux WS3 with all graphics disabled. We utilized our C software back-end as a second control and correlated that to NEURON running with a fixed dt. Our C generated code was compiled using the GCC 3.2.3 optimizing compiler and executed on the identical machine as NEURON. The FPGA back-end produced default precision values and was compiled targeting the Xilinx ML402 development board with an XC4VSX35-fg676-10 device. The results of the simulations for a FitzHugh–Nagumo (FN), Hodgkin–Huxley (HH), a Hodgkin–Huxley with all-to-all except recurrent synaptic connections[90], and the Booth, Rinzel, and Kiehn model. Only the Hodgkin–Huxley models were implemented on NEURON, but all were generated and tested via the C back-end. Both the 10-neuron and 40-neuron Hodgkin–Huxley models were additionally tested with a fixed time-step in NEURON. The results are summarized as real-time performance in Table 19.

Real-time performance is generated based on the following relationship:

$$\times\text{real-time} = \frac{dt}{d \cdot t_c} \tag{62}$$

where d is the pipeline depth and t_c is the critical path latency. The critical path is the reciprocal of the maximum clock frequency. Actual clock frequency is dependent on any

Table 19: Real-time performance comparison between DYNAMO targeting an FPGA, DYNAMO targeting the C software back-end, and NEURON v5.8 using a variable time-step solver and fixed time-step solver.

Model	Total Ops	FPGA	C	Neuron	Neuron (fixed dt)
FN	16	521,105	1,315,800		
HH	91	24.3	5.2	2.65	
HH10	900	13.6	0.51	0.89	0.48
HH20	1800	10.5	0.23	0.51	
HH30	2700	8.28	0.15	0.36	
HH40	3600		0.10	0.28	0.13
HH10X	1980	8.97	0.26	0.42	
BRK	183	12.8	5.52		

available clocking circuitry and but can often be generated to be close to the maximum clock frequency. All real-time factors are in units of s/s where values greater than one indicate greater than real-time performance.

In Table 19, it is evident that not all models enjoy significant performance increase. Smaller, non-repetitive models are often hampered in their ability to produce impressive performance results. In the case of the FitzHugh–Nagumo model (FN), the performance was reduced over 50% from the C back-end to the FPGA implying the model was too small to take advantage of the parallelism offered on an FPGA. The single Hodgkin–Huxley model offered a 4.7× performance over the C back-end and a slightly more impressive 9.2× boost over the comparable model in NEURON. The Booth, Rinzel, and Kiehn model also produced sub-optimal performance improvements of 2.3× the C back-end. When multiple models were evaluated in the case of the Hodgkin–Huxley model, more dramatic improvements were witnessed. For ten, twenty, and thirty Hodgkin–Huxley models, performance gains over NEURON equaled 15.3×, 20.6×, and 23×, respectively. Finally the interconnected ten-neuron population (HH10X) with 90-synapses enjoyed a 21.4× improvement.

The second criteria for a positive results is an architecturally-reasonable outcome. This implies that the model in its implementation was "intelligent," *i.e.* the DYNAMO compiler algorithms produced an expected, suitable output. For example, the number of cycles in the data-path pipeline should be commensurate with the complexity of the model and should

scale along with the size of the model. Area utilization will ideally be close to maximal to ensure the most parallelism. Multiplexers should be used sparingly as a optimally developed pipeline would not require multiplexers as sources and sinks of operations will overlap. These are fuzzy criteria and necessary as such since the cost-function does not necessarily produce the optimal result in all conditions. Since the performance as shown in Table 19 has satisfied the first criteria, this is meant more as an internal progress report as to how the DYNAMO compiler performs internally and if there is still room for future growth in performance.

Pipeline depth is a key determinate of overall performance as it is equivalent to the sample period of the system. For every increase in the depth of the pipeline, an additional clock cycle is required. As the model increases in complexity, it is expected that the pipeline depth slowly increase to handle the additional computational overhead. A successful algorithm will have either a linear response with a slope representing a small nominal increase in time per additional neuron simulated. In the following plots for a FitzHugh-Nagumo population (Figure 41), a Hodgkin–Huxley population (Figure 42), and a fully-interconnected Hodgkin–Huxley population (Figure 43), the depth rises linearly with the increase in units simulated. For the former two, the unit is the number of neurons. In the latter model, as the number of synapses increases, the pipeline depth scales less with the population size and more with the total number of synapses. Linear fits are performed for each plot and the R^2 values are included in the plot.

To further analyze the results of an increase in pipeline depth as the complexity increases, the cycle count was averaged over the total population size or synapse count. As the number of models increases, it is expected that the average cycles per model will asymptotically approach a limit at which performance is maximized. The optimal outcome occurs when where every resource is assigned corresponding operations across the models (as in the architectures demonstrated in [37, 88, 89, 90]). An additional model to a pipeline should result in only one additional cycle. If the model is small enough such that multiple pipelines can form, than the an additional model will on average result in less than one cycle. If the model is too big to be instantiated as a flat model, such as in the Booth, Rinzel, and Kiehn model, an additional model will require more than one additional cycle.

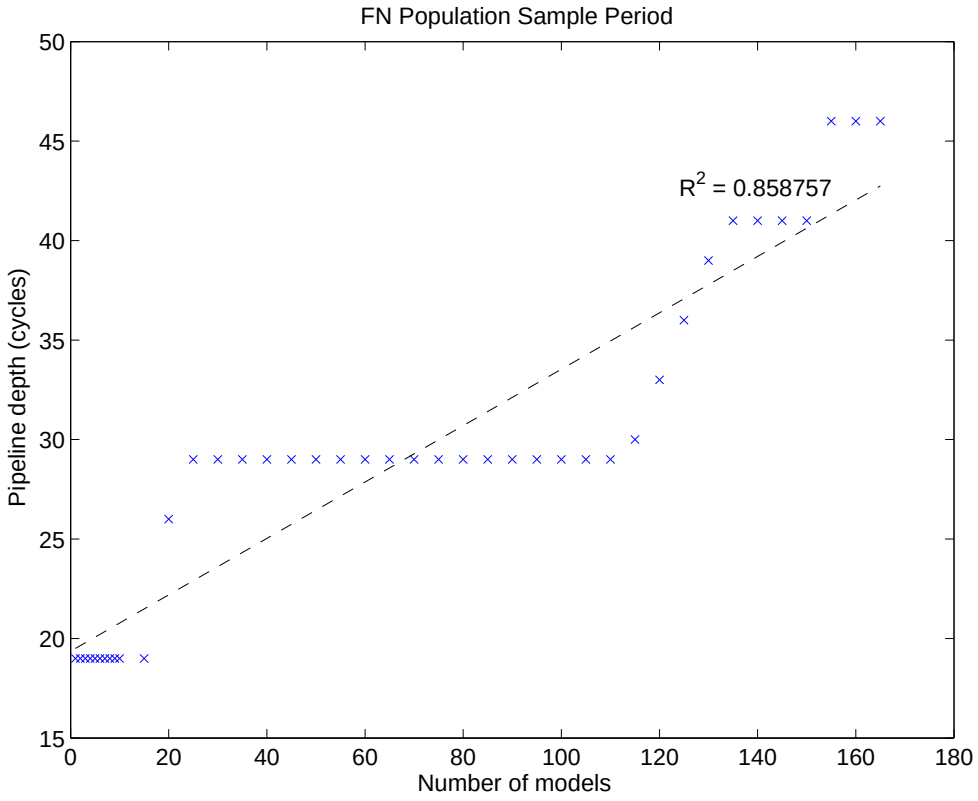


Figure 41: Plot of the pipeline depth for increasing numbers of FitzHugh–Nagumo models.

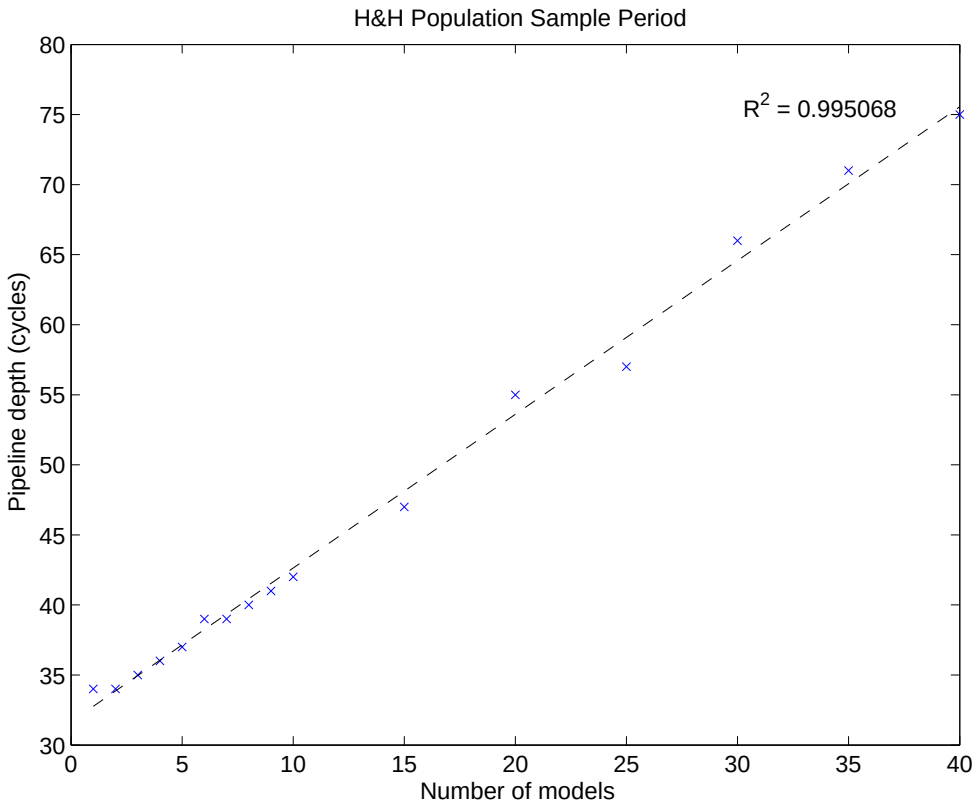


Figure 42: Plot of the pipeline depth for increasing numbers of Hodgkin–Huxley models.

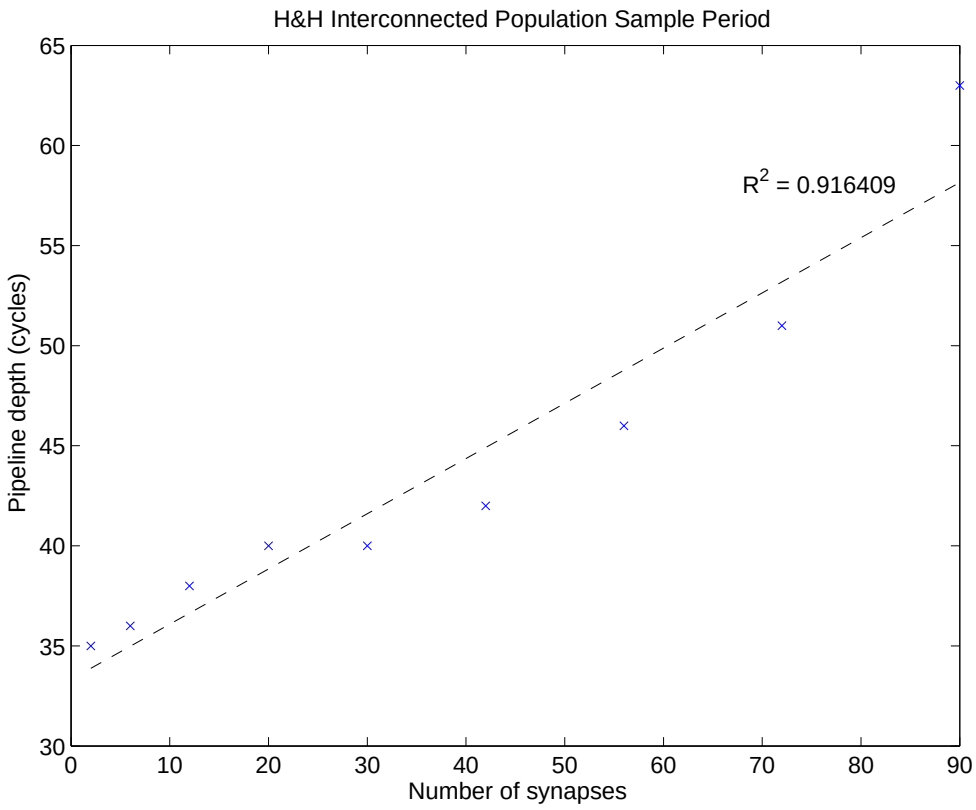


Figure 43: Plot of the pipeline depth for increasing numbers of interconnected Hodgkin–Huxley models.

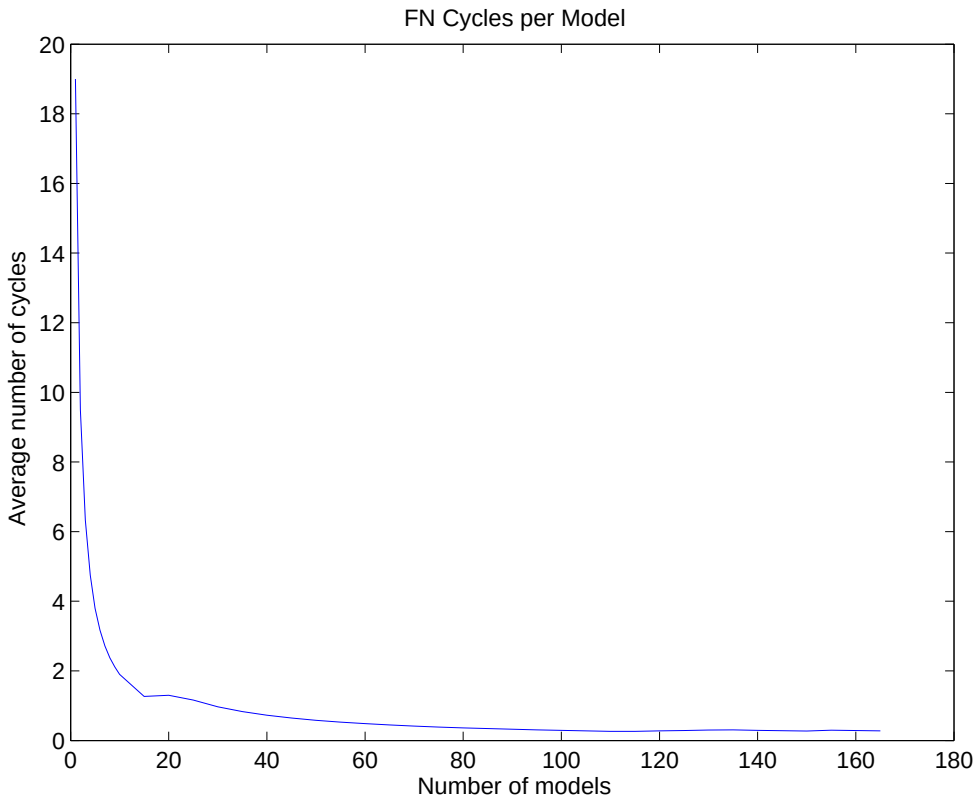


Figure 44: Plot showing the asymptotic average cycle count as the number of FitzHugh–Nagumo models increased.

We analyzed the average cycles per model for the FitzHugh-Nagumo model (Figure 44), the Hodgkin–Huxley model (Figure 45), and an interconnected Hodgkin–Huxley model as a function of the number of synapses (Figure 46). In each plot, a characteristic reciprocal form emerged implying a steady-state increase in cycles for each additional model.

As described in Section 6.6.3, a number of statistics are gathered by the DYNAMO compiler after each schedule optimization. These statistics provide a useful window into the operation of the scheduler, helping to understand how close the auto-generated result is to an equivalent manually-generated model with a fully-utilized pipeline. In particular, the pipeline depth is compared against the percent utilized (proportion of the scheduler operations' time-slots that are assigned) and the percent multiplexed (proportion of operation inputs requiring multiplexors). In a manually-generated pipeline, according to our previously published methodologies, the percent utilized would be 100% and the percent multiplexed would be 0%. As evident in the following figures, these optimal cases are

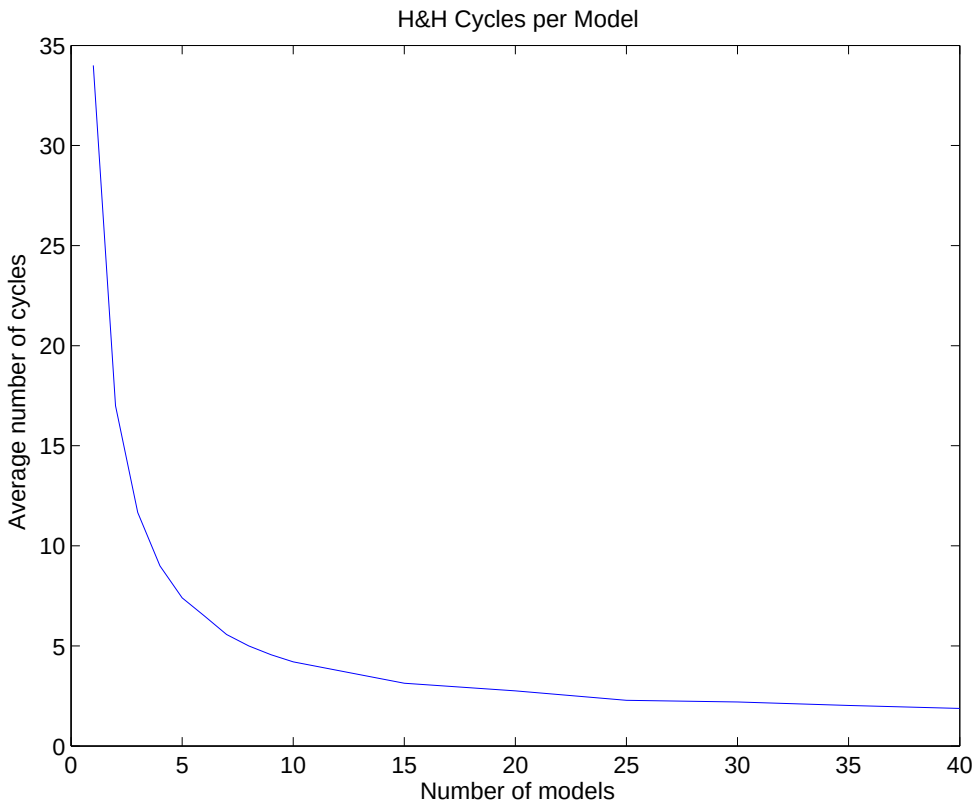


Figure 45: Plot showing the asymptotic average cycle count as the number of Hodgkin–Huxley models increased.

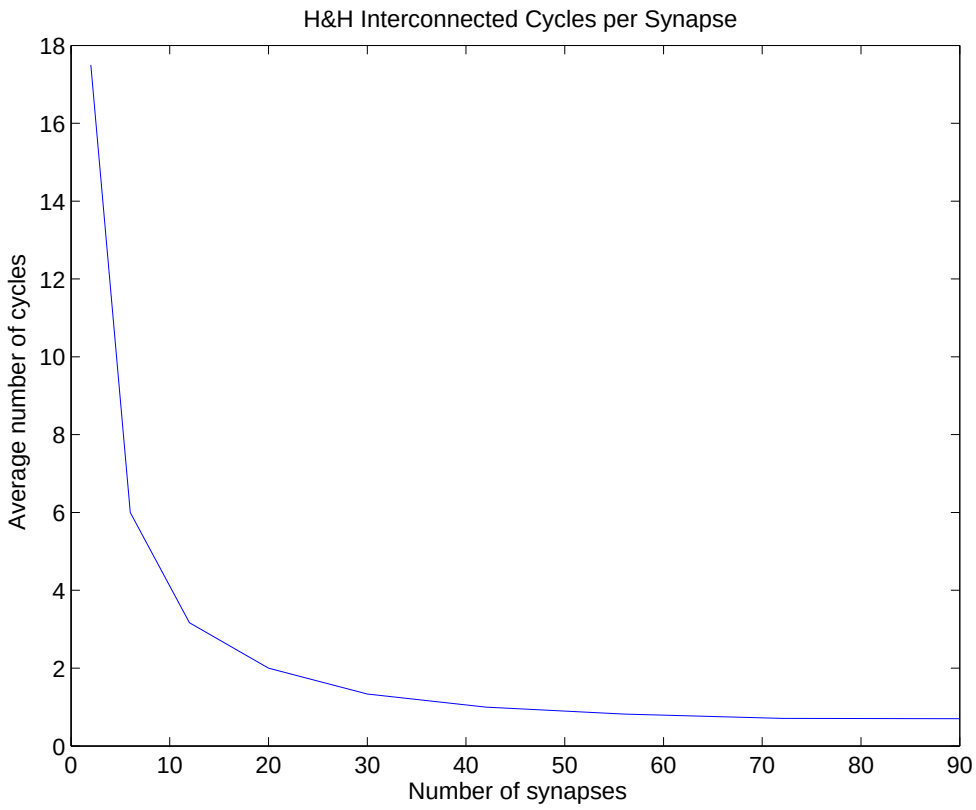


Figure 46: Plot showing the asymptotic average cycle count as the number of interconnected Hodgkin–Huxley models increased.

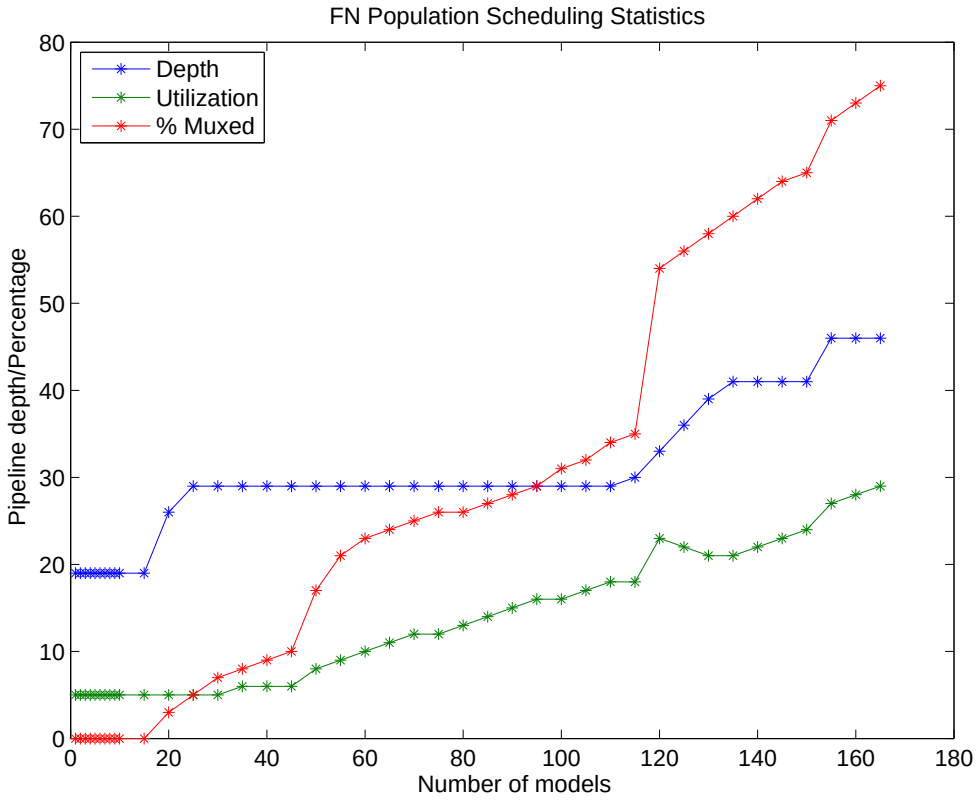


Figure 47: Statistics retrieved from the scheduler showing percent schedule utilized and resources over utilized (% muxed) along with pipeline depth as the number of FitzHugh–Nagumo models increases. For resources over scheduled, % muxed refers to the percentage of operation inputs that require a multiplexer to select among multiple signal sources.

not reached, yet the compiler is clearly making good decisions to produce positive results. For the three model types demonstrated, schedule statistics are illustrated in Figure 47, Figure 48, and Figure 49.

Further clues can be garnered from the resource utilization statistics gathered from the area estimation analysis. The DYNAMO compiler fails when a particular resource gets constrained to the point where it can not be reduced in any way. This often happens with register resources. As the number of state variables increases, many of the available registers in the system are reallocated to store state values and not for internal pipelining stages in the operations. This has the negative affect of reducing the number of operations that can be instantiated freeing up *ffs* and *luts*. A reduction in the number of operations causes the creation of many multiplexers to handle the increased utilization per operation.

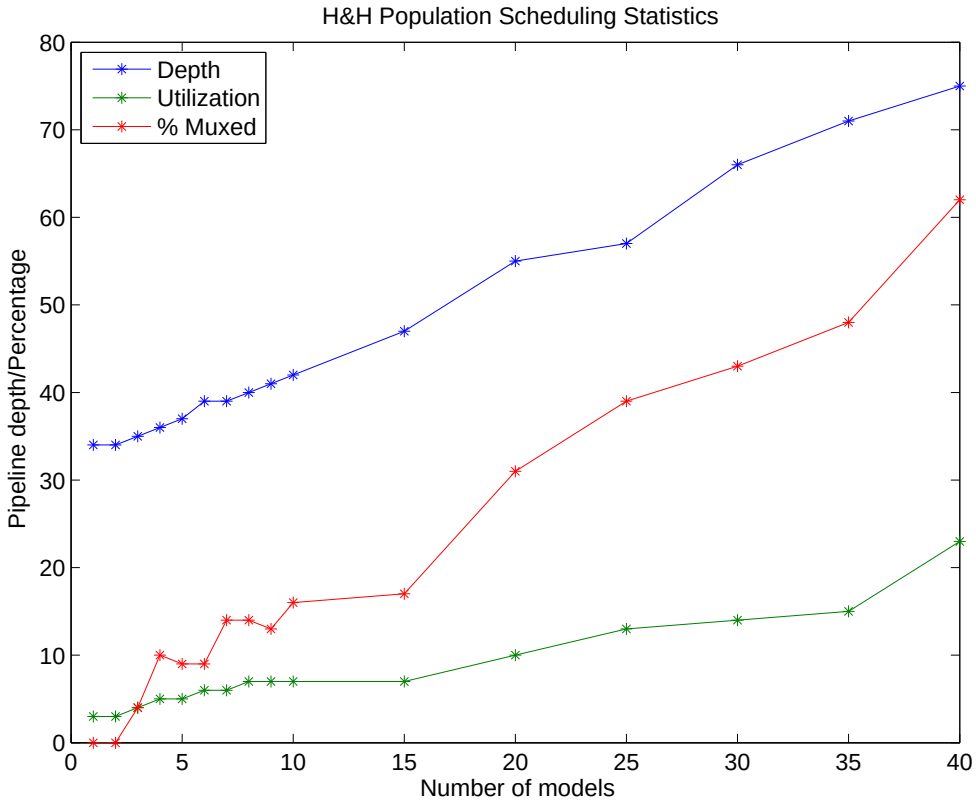


Figure 48: Statistics retrieved from the scheduler showing percent schedule utilized and resources over utilized along with pipeline depth as the number of Hodgkin–Huxley models increases.

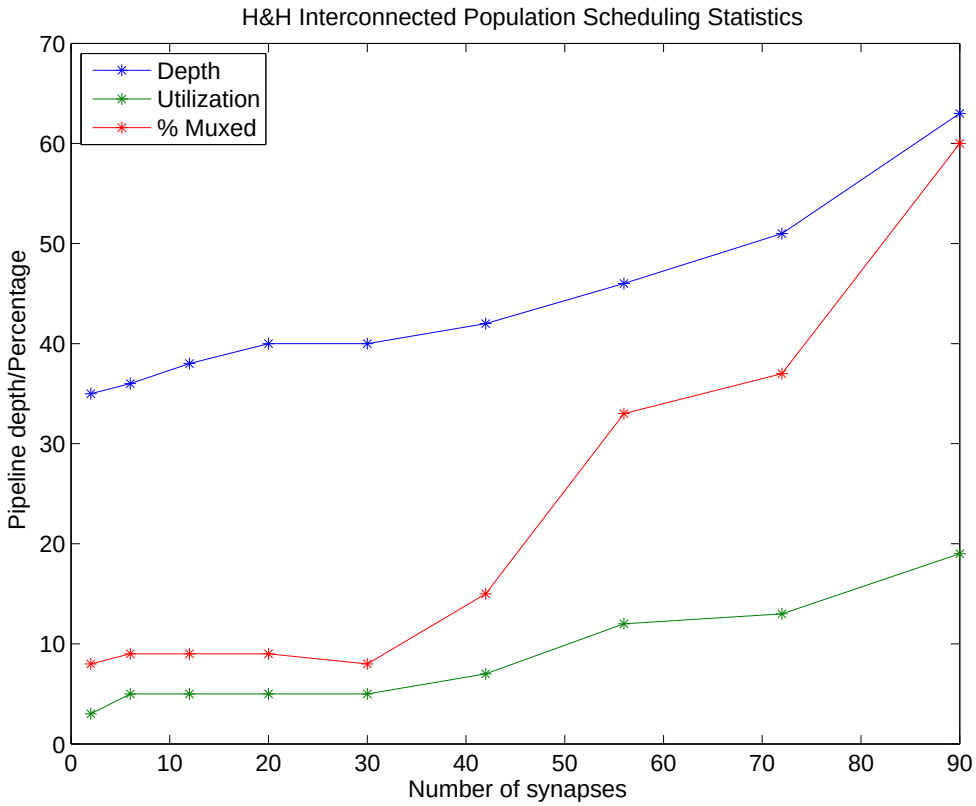


Figure 49: Statistics retrieved from the scheduler showing percent schedule utilized and resources over utilized along with pipeline depth as the number of interconnected Hodgkin–Huxley models increases.

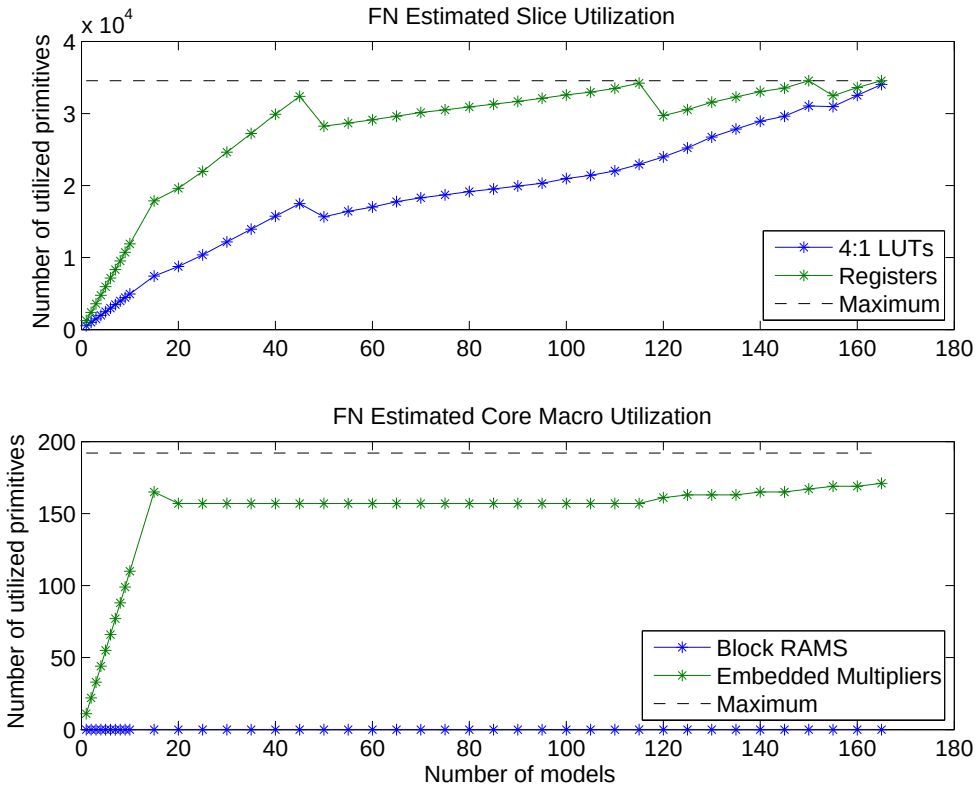


Figure 50: Plots showing increases in *luts*, *ffs*, *mults*, and *rams* as the number of FitzHugh–Nagumo models increases. The dashed line indicates the maximum resources for the target FPGA.

Multiplexers consume large portions of the newly available *luts*. Eventually, the device's *ffs* and *luts* reach a critical level and the compiler must abort without a solution.

Disjointed segments in the resource utilization plots indicate where the compile has been forced to reallocate resources based on over utilization. In the FitzHugh–Nagumo plot (see Figure 50), *mults* were initially the constraining resource, but later shifted to *ffs* until the lack of available *luts* eventually caused the compiler to cease after 165 models. For the Hodgkin–Huxley population (see Figure 51) and interconnected model (see Figure 52), both were initially constrained first by *ffs* and then by *luts*. This highlights that a major improvement can be possible through the use of more granular memory structures for state variables.

The DYNAMO compiler reached a maximum number of 165 FitzHugh–Nagumo models. The pipeline depth was 46 cycles to evaluate all 165 models. The average cycle count,

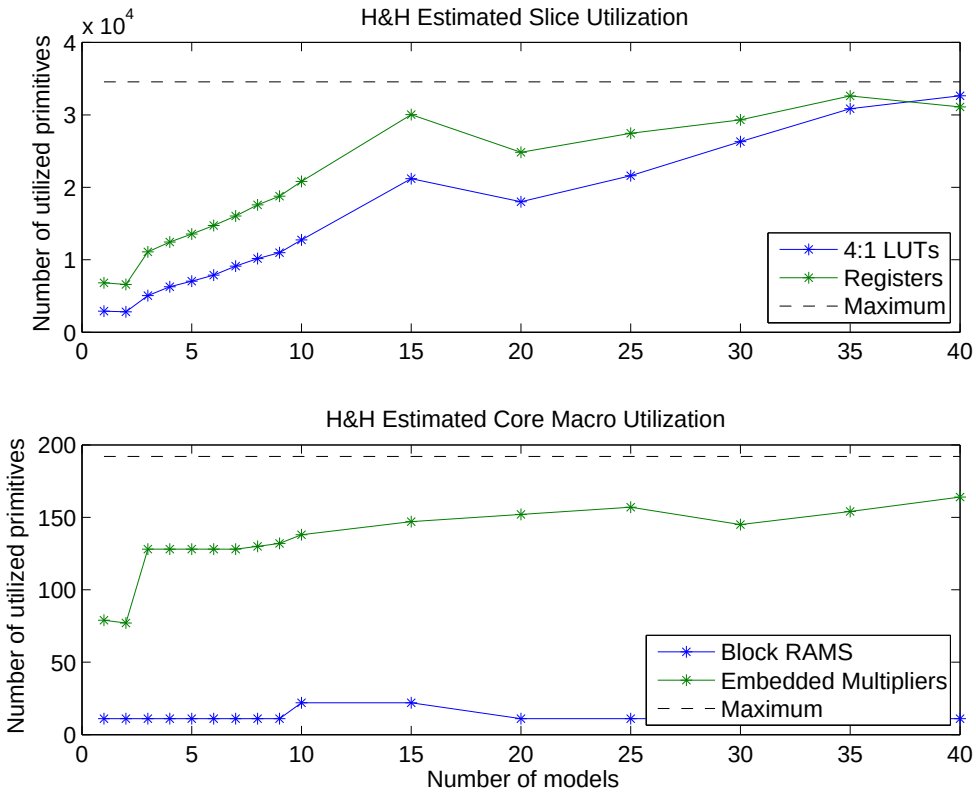


Figure 51: Plot showing increases in *luts*, *ffs*, *mults*, and *rams* as the number of Hodgkin–Huxley models increases. The dashed line indicates the maximum resources for the target FPGA.

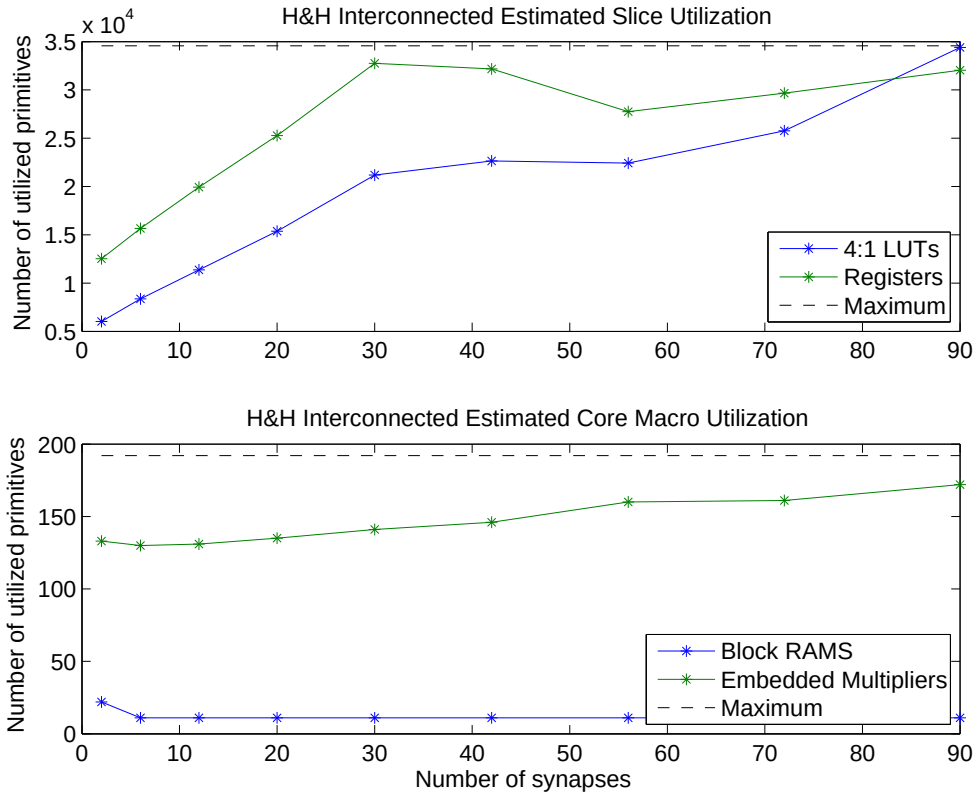


Figure 52: Plot showing increases in *luts*, *ffs*, *mults*, and *rams* as the number of interconnected Hodgkin–Huxley models increases. The dashed line indicates the maximum resources for the target FPGA.

0.28, meant that approximately four simultaneous pipelines were developed, although not exactly. From looking at Figure 41, it is evident that up to 15 models were instantiated in a non-overlapping configuration. We know that no resources were used more than once since in Figure 47, the multiplexer utilization remained zero up through 15 models. Therefore, 15 pipelines should have been able to emerge. With 15 pipelines, an additional 150 models should have required 10 additional stages, reducing the pipeline depth to 29 cycles. In practice, this is not possible as *ffs* became constrained as shown in Figure 50. This is a minor criticism of the scheduler showing room for future performance enhancement. Despite the sub-optimal outcome, a hand-generated version would have likely required 165 cycles for computing all models, almost four times the cycles as in the auto-generated case.

The Hodgkin–Huxley population model without synapses enjoyed a near linear rise in depth for an increase in the number of models (see Figure 42). The first two models were able to fit flat in the FPGA (as evidence by the 0% multiplexer utilization and that the pipeline depth and overall percent utilization stay constant across one and two models. An increase to 40 neurons should have required 53 cycles if the two pipelines were always instantiated. Similar to the FitzHugh–Nagumo case, limitations in *ffs* led to a rescaling of the DRT size at 20 neurons and two rescalings at 40 neurons. At 40 neurons, the percent multiplexed reached 62%, a value high, but understandable as the number of resources available for the models were shrinking (see Figure 48). Overall, the addition of 39 models after the first caused an increase in 41 cycles, or almost 1 cycle per model. This is similar with what would be accomplished in a manually-generated Hodgkin–Huxley population model.

The interconnected Hodgkin–Huxley population model could be built with up to 10 neurons before exhausting available resources. The pipeline depth was not linear with respect to the number of neurons, but was linear ($R^2 \geq 0.9$) with respect to synapses as shown in Figure 43. For an n neuron model, $n(n-1)$ synapses linked each neuron with every other neuron. The number of pipeline stages asymptotically averages to less than one per each additional synapse. Percent utilization rose continuously, but the percent multiplexed reached a high of 60% (see Figure 49). The limiting resource were the *luts* where the limit

was reached following a rescaling based on *ff* overage (see Figure 52).

The results show that models of a sufficient complexity can benefit from execution on an FPGA utilizing the DYNAMO compiler. Models that are too small do not benefit from the parallelism of the FPGA. Very large models, at this point, are beyond the reach of DYNAMO as the storage of states in registers is the limiting constraint. For a large class of neuron models (approximately 100 to 3000 operations), an auto-generated FPGA design flow has been shown to provide at least a 10× improvement in performance. Future growth in performance is evident by careful examination of the results from compiling populations of neural models.

6.8 Validation

For the DYNAMO compiler to find utility amongst neural modelers, the compiler must be able to produce comparable results to standard sofware-based neural modeling tools. We know that FPGAs can be used to successfully develop and utilize models [37, 87, 81, 89]. This section describes our approach to validating models produced with DYNAMO. A validated model implies that the FPGA implemented version produces qualitatively comparable results as a software simulation.

A robust validation process would ideally extract a series of behavioral metrics from both the FPGA implementation and software implementation with a statistical analysis to show correlating behaviors. This is not to suggest that parameters and output metrics are quantitatively equal, but rather that general behaviors are replicated. For example, a bursting neuron population must exhibit bursting on the FPGA and in software while the specific bursting period need not be exactly equal. Results are not expected to be equal as simulations are different with respect to fixed-point precision, integration algorithms, and function estimations.

DYNAMO utilizes conservative algorithms for precision determination (see Section 6.4.2). This provides a reasonable first-pass attempt to build a functional implementation. If the DYNAMO compiler does not produce desired results, there are a number of options built-in to enable manual interaction with the tool. For example, the MATLAB software back-end

optionally produces a fixed-point back-end with statistics logging. These statistics include overflows and underflows per operation and minimum and maximum step per integrated state variable. Overflows are considered a fatal error and must be corrected as an overflow will cause wrapping. Underflows might not be fatal and have to be individually investigated. States where the minimum integration is equal to zero might imply an underflow is inhibiting the state from properly evolving over time. The statistics option produces a table of results along with fixed-point precision settings. Options within DYNAMO enable a two-pass compilation mode whereby a precision file is generated on the first-pass which includes each expression and fixed-point precision. On the second-pass, DYNAMO can optionally read in an annotated version of that file adjusting precision values internal to the compiler to reflect the changes. This way, the user can hand-tune underflow errors from the system.

We have anecdotal evidence to suggest that a majority of precision errors are a result of under-sized lookup-tables. This is often manifested by visible stair-stepping in an output state implying limited precision at one or more operations within the expression graph. Maximum depth and width of lookup-tables defaults to 1024 elements deep and 18-bits wide, respectively. These limits are adjustable on the command line. Increases in table size have resulted in improved results, such as in the Booth, Rinzel, and Kiehn (BRK) model. When implemented with standard memory configurations for lookup-tables, calcium concentration in the dendritic compartment remained constant at zero. When an additional 4-bits of addressability were added (increasing depth to 16384 elements) and by allowing up to 36-bit width, the BRK model integrates its states as expected.

The DYNAMO modeling language (DML) provides extensive flexibility with respect to describing models. For example, bounded iterative approximations described in the DML can be a substitute for automatic lookup-table generation. Adjustments to the lookup-table heuristic on the DYNAMO command line can increase or decrease the propensity for tables to form. Additionally, the precision analysis routines return warnings for inappropriate ranges, such as a denominator of a division crossing zero. These warnings often precede functional errors in simulation. Algebraic reformulations in the DML can help to alleviate

these warnings.

When the auto-generation algorithms are not valid across all cases, DYNAMO provides a wide range of configuration options to tweak the resulting model implementation. This way, the user can take part in adjusting and tweaking to ensure their neural model is appropriately validated.

6.9 Future Work

While the DYNAMO compiler is highly capable in its current state, future work can enable larger models, enhanced performance, and improved usability. An improved modeling language with possibly more neural-specific constructs can aid the modeler in developing their models. The basic assumption of one execution engine forming the run tree to the run graph to the run block can be enhanced to allow for multiple execution engines, implementing a state machine. Additional numerical integration algorithms can be considered in future iterations including higher-order solvers as well as variable time-step solvers.

Fixed-point generation is a constant focus of research and will likely continue to be for some time. There is yet to be a general algorithm to predict precision requirements for highly nonlinear, coupled systems of differential equations providing identical results. This algorithm might never exist. Incremental improvements are nonetheless possible with additional research effort. Anecdotal evidence has shown that lookup-tables in particular are a major source of precision error in the system. Inclusion of additional estimation constructs, such as bipartite tables [27] and polynomial approximations can offer improved precision at a cost of additional logic.

The previous iteration of the DYNAMO compiler incorporated the generation of the this System Generator build-script directly into the scheduler. By developing this clear delineation between the System Generator output and the scheduler in the form of the generic netlist format, additional back-ends are possible. Preliminary, exploratory work has gone into building a purely Verilog back-end. Verilog was chosen over VHDL based on familiarity, its flexible type system, and its simplified syntax. Synthesizable parameterized modules have been built for delay, adder, subtracter, and multiply blocks. For a complete

Verilog back-end to emerge, the remaining blocks need to constructed as parameterized modules. Additionally, a infrastructure that does not rely on System Generator to provide the co-simulation interface has to be developed.

These and other enhancements can make the DYNAMO compiler even stronger as an auto-generation platform for neural modeling applications. In its current incarnation, DYNAMO is a capable platform for translating and compiling neural model descriptions through an intermediate representation into an FPGA-based execution engine.

CHAPTER VII

CONCLUSION

The overriding goal of this thesis has been to develop a methodology to utilize an FPGA for neural modeling applications. FPGAs have incredible parallel processing capabilities which bodes particularly well for the iterative numerical evaluations required when solving systems of differential equations. Their drawback has typically been in ease-of-use, often requiring dedicated electrical engineers to develop for FPGAs. This thesis bridges the gaps of understanding between engineers, physiologists, and modelers, allowing the scientific community to embrace FPGA technology without the required domain knowledge.

There have been four phases, as described in the previous four chapters, in generating this thesis. First, FPGAs had to be shown to not only be capable of describing neural models, but also to provide a substantive benefit. Two publications [37, 88] demonstrated the successful implementation of neural models including a Hodgkin–Huxley squid giant axon model[40], a Booth–Rinzel model simplified motoneuron model [9], and a multi-conductance, multi-compartment motoneuron model. In each case, performance increases from 10× to 72× were demonstrated. These performance increases mark a revolutionary increase in computational capability; however, the implementation was time-consuming and difficult to tune or rework. Despite of these difficulties, these models served as a proofs-of-concept for the project.

The next phase was a study into the "how" of model construction. We knew that model implementation on FPGAs could be done with positive results, but in this step, we looked for design rules, algorithms, and methodologies to aid modelers in developing their own neural models. The first efforts were aimed to develop candidate architectures for simple models (*i.e.*, single compartment, single neuron) to multi-compartment or neural populations. A clear design approach was developed whereby the integration of the differential equations was computed using a standardized approach, and the data-path, or the calculation of the

flow field was custom built according to a set of simple rules. Guidelines for conversion to fixed-point were explored and suggestions offered. Timing considerations were described, and simple algorithms for properly synchronizing a design were developed and explained. This set of algorithms makes up the core design guide for neural model development on FPGAs and was subsequently published in the *Neural Engineering* journal [89].

Following the development of the model construction methods as described in Chapter 4 suggesting a canonical form for neural models, we investigated means to ease interaction with the model. This third phase result was a set of tools to aid the modeler in designing for and interacting with an FPGA. These tools allowed the auto-generation of the integration module, run-time parameter tuning, and multiplexing of two analog outputs for debugging. These tools provide assistance to modelers by allowing them to focus on the data-path portion (the equation component) of the model without focusing on the FPGA particulars. This design approach enabled a collaborative effort with Michael S. Reid that resulted in the development of a 40 neuron population model implementing an all-to-all synapse network with a total of 1600 synapses. The system was a reduced Pre-Bötzinger Complex (PBC) network and provided Reid with a high-performance platform for his modeling studies. The assisted flow methodology with applications to the construction of the PBC network has been published in *IEEE Transactions on Neural Systems & Rehabilitation* [90].

The final phase of this thesis has taken the development of neural models one step further towards a fully-automated design flow. The assisted-flow methodology still required significant domain knowledge in digital design principles. We required an automated design flow was capable of providing the most accessible and simplest process to translate a model from a dynamical system description to a high-performance execution core on an FPGA. The project, known as Dynamo, has consumed the bulk of effort within this thesis. Dynamo has been built from over 50,000 lines of code in a dense functional language and has enlisted the efforts of multiple developers. As a prototype environment, Dynamo has clearly demonstrated the feasibility of automated FPGA development by providing a full front-to-back translation and compilation.

CHAPTER VIII

DISCUSSION

8.1 Contributions of this Thesis

The overriding goal of this thesis has been to investigate the use of FPGAs for neural modeling. Prior to this research, we found that FPGAs were heavily utilized for high-performance applications, particularly as reprogrammable DSP processors. We further recognized that numerical solutions of differential equations, the base operating means for neural model simulations, share very similar mathematical characteristics to the difference equations that underlie many DSP algorithms. We hypothesized that neural models in the general case can be implemented on FPGAs with substantial performance gains.

Developing example models for an FPGA would not be enough to create a significant impact on the neural modeling community. FPGAs had several drawbacks that made modeling difficult, and unless they were overcome, use of these devices would be minimal. If we were successful in the implementation of these FPGA models, this research would still be constrained by several requirements. First, as models can grow in size, a means for handling large models must be found. Next, accuracy concerns for utilizing fixed-point numerics must be minimized. Lastly, this whole process must be easy to use for the modeler. Without these three conditions, FPGA neural simulations would not be practical for many modelers.

The real significance of this thesis is that we were able to overcome these drawbacks. These drawbacks were gradually overcome through each iteration as described in chapters 3–6. We solved the size issue for specific cases through our pipelining methods described in Chapter 4 and for the general case in Chapter 6. We diminished the numerical precision concerns of modelers through our floating-point to fixed-point algorithms and successfully demonstrated our approach on a numerically-stiff Pre-Bötzinger Complex model (see

Chapter 5). We engineered a fully automated design flow for the conversion of neural models into a simulation engine running on an FPGA, introducing the first platform for neural modelers to utilize FPGAs as a generic simulation coprocessor. This final contribution, the DYNAMO compiler, encompasses the previous two contributions, a host of additional novel technologies, and innovated applications of existing technology (provisional patent filed covering the internal representation of the compiler, the dynamical system specific scheduler, and the front-to-back application of technology for neural modeling). This integrated system, we believe, will likely enjoy the largest impact in the neural modeling community.

8.2 Discussion of the Dynamo Compiler

In particular, the DYNAMO compiler has several key capabilities that separate it from other solutions that warrant more discussion. First, the DYNAMO modeling language (DML) has been designed as a language for describing both the construction and the mechanism of models. Inspired by a wide range of programming languages and modeling environments including Mathematica, XPP, NEURON, and MRCI, the DML has a rich feature-set including objects (encapsulation and inheritance), modularity, functional/mathematical paradigm, and syntax for domain-specific features including differential equations, integration routines, sampling rates, parameter manipulation, embedded range information, and untyped data-types. As the third-generation modeling language, a primary goal has always been flexibility as we cannot predict today what form future models might take.

These language features are unprecedented in the realm of currently available direct-to-hardware description programming languages. A variety of languages are available for describing hardware, such as Verilog and VHDL, yet both operate at an RTL (Register Transfer Logic) level. This mode of operation requires the modeler to be fully aware of bit-level operations and register timing. Higher level tools abstract the RTL effectively creating a short-hand means of hardware design. This category includes Xilinx System Generator, Mentor Graphics FPGA Advantage, and National Instruments LabView FPGA. These tools aid experienced hardware engineers in realizing and/or prototyping their designs in less time; however, underlying domain knowledge is still required. Still other tools currently exist for

Matlab (Xilinx AccelChip), C (Celoxica), or Simulink (Mathworks HDL Coder) direct conversion. These tools are challenged from the onset as these languages were not designed for a hardware target and thus have to make performance trade-offs to accommodate software assumptions. Additionally, none of these have the language features specific to modeling thus requiring the modeler to build a custom harness to interact with their model.

DYNAMO does not merely translate the equations into an FPGA output, but instead refactors it by performing various optimization routines. For example, DYNAMO automatically propagates constant values, removes static computations from the execution loop, and removes redundant calculations. These features allow the modeler to express the model in a convenient form while ignoring performance implications without consequence. Few tools, with the exception of optimizing C compilers, have the capability to algebraically manipulate the equations for performance benefits.

Floating-point to fixed-point conversion is built-in to the DYNAMO compiler in a partially automated fashion. This step has proven itself to be the most challenging aspects of producing an automated flow. The systems of non-linear differential equations that typically characterize neural systems are often stiff, lack analytical solutions, and are victims of the simulation environment's choice of integration algorithm and numerical precision. Accordingly, neural models cannot be expected to perform identically when the simulation platform changes drastically. In particular, a continuous-time numerical solver with double floating-point precision on a computer and a fixed time-step solver with fixed-point precision executing on an FPGA are sufficiently different such that modelers should expect different results. An additional source of error, approximation functions, complicate the validation further.

The different sources of error can be controlled and mitigated in unique ways. First, fixed time-step solvers often utilize time-steps that are simple to represent in decimal, such as 0.1, 0.01, or 0.001, but cannot be represented in binary. On a double-precision computer processor, 52-bits are utilized for the mantissa providing a suitable approximation. While in hardware, 52-bits can be allocated to the storage of this constant, it is fairly wasteful and not generally necessary. If a full 52-bits are not used, the solution will invariably drift in

a fixed-point system. A work-around would be to use a value that is exactly representable with a minimal number of bits, such as 0.125 (2^{-3}), 0.0078125 (2^{-7}), or 0.0009765625 ($2^{-}10$).

Second, quantization error is a significant source of discrepancies in simulations and is difficult to fully quantify. One class of quantization errors, underflows, occurs when a non-zero value is quantized to zero. When this happens for the error term of an integration, this can inhibit slow time-scale processes from proper emulation. Quantization errors can generally be avoided with sufficient precision, but high-precision is costly in terms of area utilization. Low-precision operations are preferred for performance, at a cost of increased quantization error. As such, the minimization of quantization error is still an active area of research. Shi and Brodersen at University of California, Berkeley, have lead the field in automated algorithms to determine fixed-point precision. Their algorithms utilize a multiple objective optimization that is specific to the algorithm [76, 78, 79]. For example, a QAM (quadrature amplitude modulation) modulator might be able to use bit-error rate (BER) to quantity error. An algorithm for a neural model would likely need specific metrics to evaluate fitness, for example, spike rate or bursting period. Requiring a particular fitness function from the modeler deviates from the fully-automated design philosophy but might be required. Even with this fitness function, identical behavior cannot be assured. According to Shi, removing quantization effects is not possible for all cases, especially in non-linear systems with feedback, such as differential equation modeling [79].

The third class of precision errors occurs when approximations are used in place of accurate iterative calculations. DYNAMO utilizes look-up tables for all difficult-to-compute expressions, including exponentiation, trigonometric functions, and reciprocals. The first FPGA-based physiological models utilized look-up tables of 1024 elements [37, 88] which appeared sufficient for those models. The PBC network required look-up tables of 4096 elements (12-bit addressable) [90]. Current work with a Booth, Rinzel, and Kiehn model [10] suggests tables with 14-bit addressability might be required to enable full functionality. This implies that future work should explore alternatives to static look-up tables such as multi-partite tables, variable-order approximation functions, or other pipelined techniques

to reduce the memory footprint required [70, 82, 27, 28]. Some of these techniques, such as bi-partite table approximations, are possible to implement in the DML, though a compact and optimized form would require DYNAMO enhancements.

The hardware scheduler is one of the key differentiating features of the DYNAMO compiler. When designing a system in a typical hardware description language such as Verilog or VHDL, care has to be taken to ensure the resulting implementation fits within the available resources of the FPGA. Larger designs require more expensive FPGAs with additional resources. The target Virtex-4 FPGA (XC4VFX35) could accommodate no more than two Hodgkin–Huxley models. The typical modeler is often interested in models of significantly more complexity. The hardware scheduler has been the key innovation employed in DYNAMO to handle substantially larger models. By effectively rebuilding hierarchy in a flat expression graph, DYNAMO intelligently groups similar expression sub-graphs on common resources not unlike manually generated FPGA models described in this thesis.

The hardware schedule works closely with an FPGA resource estimation, another unique feature of DYNAMO. The internal resource estimation tools provide the scheduler with the physical constraints of the target device. By iteratively producing a schedule until a design is estimated to fit, the likelihood of a resulting implementation passing through place & route is dramatically improved.

The technology in DYNAMO is not limited to a System Generator design flow. We use System Generator because it provided built-in support for fixed-point data types and featured a co-simulation mode for digital data-transfer. We have spent considerable effort in exploring alternatives to System Generator. We built a limited library of Verilog-based fixed-point operations and successfully built a FitzHugh–Nagumo model solely using Verilog. A replacement co-simulation platform was more difficult and would have required considerable engineering effort beyond the scope of this thesis.

8.3 Impact of this Thesis on Neural Modeling Applications

Prior to the generation of this thesis, FPGAs were no more than a curiosity for neural modelers. Today, FPGAs are a realistic platform for high-performance, complex neural

modeling applications. We believe that FPGA usage is highly correlated with ease of use. As the DYNAMO compiler becomes accessible to the general modeling community, we expect to see a surge in FPGA designs. Initially collaborative works are beginning to come to fruition.

We expect DYNAMO to be embraced across within multiple fields of neuroscience. The dynamic-clamp community utilized neural cellular or mechanistic models along with a traditional "wet" electrophysiology experiment to study a hybrid system. This community can benefit from FPGAs real-time characteristics and high-performance for complex models.

For smaller models, such as the Hodgkin–Huxley model, parameters were gleaned from experimental data and mechanisms were based on curve-fit approximations. As the size and complexity of models continues to increase, there is a growing realization that parameters can not fully be specified by experimental data alone. This was found through the study of the non-uniqueness question: can multiple, unique parameter sets produce similar output behavior? Foster *et al.* utilized a stochastic search to find numerous ionic maximal conductances to support a target firing range [33]. Prinz *et al.* expanded this field by generating a large scale database of parameters for a six conductance model [64], her results supported previous work that suggested averages of experimentally determined conductances do not necessarily produce the desired output [35]. This growing disconnect between experimentally determined parameter sets and functional output suggests that parameter sets found through iterative searches of the parameter space can lead to physiologically-relevant models. For larger models, the parameter space becomes under-determined, whereby exact values for parameters based on experimental results is no longer a precondition to constructing a model. This "backward" approach to modeling, starting with the behaviors and working back towards the parameters requires sophisticated searching algorithms and often thousands of simulations. Executing numerous iterations of a complex model is an ideal application of a high-performance, FPGA-based modeling platform.

The ability to execute a model at real-time (1 second per second) gives an experimental quality to the simulation. Contrary to the automated search algorithm described

above, manual manipulation of the model with instantaneous feedback allows the neuroscientist/neurophysiologist to perform "what if" analysis. For example, the blocking of an ion channel can be accomplished by zeroing a conductance value, not unlike pharmacological channel blockers. The intuition of an experimentalist can be used to tune model parameters to produce a desired output. Personal computers are not fast enough except for the simplest models and computer clusters can not provide low-enough latency for real-time tuning.

FPGAs are positioned to make a substantial impact in the neural modeling community by their high-throughput, low-latency, and real-time characteristics. Models from simple two-state oscillators to more complex sets of hundreds of differential equations can be generated by manual-means or automated with Dynamo. While some effort still remains for the modeler to work within the fixed-point confines of the FPGA, the benefit is significant, providing a one to two orders-of-magnitude performance improvement. Planned incorporation with the NEURON modeling tool [39] will instantly make Dynamo accessible to the wider neuro-modeling community.

APPENDIX A

DYNAMO MODELING LANGUAGE BNF

```
dynamomain ::= topleveldeflist [main]

topleveldeflist ::= {topleveldef}

topleveldef ::= 'IMPORT' string ';'
              | constdef
              | funcdef
              | systemdef

main ::= 'MAIN' maindeflist 'ENDMAIN' ';'

systemdef ::= 'DEFSYSTEM' id '(' sysarglist ')' deflist 'ENDSYSTEM' id ';'

sysarglist ::= {sysidlist}

sysidlist ::= sysargtype id {',' sysidlist}

sysargtype ::= 'CONSTANT'
             | 'DYNAMIC'
             | 'SYSTEM'

deflist ::= {def}

maindeflist ::= maindef {maindef}
```

```
maindef ::= def
          | outputratedef
          | inputdef
          | outputdef

inputdef ::= 'INPUT' ridlist ';'

ridlist ::= rid {',' rid}

rid ::= id '(' lambda 'TO' lambda 'BY' lambda ')'

outputdef ::= 'OUTPUT' outputlist ';'

outputratedef ::= OUTPUTRATE real ';'

outputlist ::= output {',' output}

output ::= outmask

outmask ::= string

def ::= systemdef
   | funcdef
   | pardef
   | constdef
   | statedef
   | sysintdef
   | equation
```

```
funcdef ::= 'FUN' id '(' idlist ')' '=' lambda ';'

pardef ::= 'PARAMETER' rasgnlist ';'

constdef ::= 'CONSTANT' asgnlist ';'

statedef ::= 'STATE' rasgnlist ';'

sysintdef ::= 'SYSTEM' asgnlist ';'

equation ::= 'd' '(' id ')' '=' lambda ';'
        | id '=' lambda ';'

asgnlist ::= asgn {',' asgn}

rasgnlist ::= rasgn {',' rasgn}

asgn ::= id '=' lambda

rasgn ::= id '(' lambda 'TO' lambda 'BY' lambda ')' '=' lambda

lambda ::= lambdaapp
         | 'IF' lambda 'THEN' lambda 'ELSE' lambda
         | lambda 'AND' lambda
         | lambda 'OR' lambda
         | 'NOT' lambda

lambdalist ::= lambda ',' lambda {',' lambda}
```

```
lambdaapp  ::= lambdaapp aexp
           | lambdaapp '(' lambdalist ')'
           | aexp
           | lambdaapp '[[' lambda ']]'
           | lambdaapp '.' 'isReady'
           | lambdaapp '.' id
           | lambdaapp '+' lambdaapp
           | lambdaapp '-' lambdaapp
           | lambdaapp '*' lambdaapp
           | lambdaapp '/' lambdaapp
           | lambdaapp '^' lambdaapp
           | lambdaapp '%' lambdaapp
           | lambdaapp '::' lambdaapp
           | lambdaapp '<' lambdaapp
           | lambdaapp '<=' lambdaapp
           | lambdaapp '>' lambdaapp
           | lambdaapp '>=' lambdaapp
           | lambdaapp '=' lambdaapp
           | lambdaapp '!=' lambdaapp
           | '{' conditions '}'
           | '-' lambdaapp

conditions ::= lambda 'WHEN' lambda ',' conditions
           | lambda 'OTHERWISE'

aexp ::= real
     | integer
     | string
```

```
    | '#t'
    | '#f'
    | '(' lambda ')'
    | id
    | '(' 'FN' '(' idlist ')' '=' lambda ')'
    | '(' 'RFUN' id '(' idlist ')' '=' lambda ')'
    | 'LET' vals 'IN' lambda 'END'
    | 'RLET' vals 'IN' lambda 'END'
    | '[' lambdalist ']'
    | '[' lambda ']'
    | '[' ']'

vals ::= {value}

value ::= 'VAL' id '=' lambda

idlist ::= id {',' id}
```

APPENDIX B

BOOTH, RINZEL, & KIEHN DYNAMO MODEL

```
/*
 Booth, Rinzel, Kiehn - 1997 - Compartmental Model of Vertebrate
 Motoneurons for Ca2+ Dependent Spiking and Plateau Potentials Under
 Physiological Treatment

 $Id: brk.dyn,v 1.1 2006/10/19 13:45:25 randyweinstein Exp $
 $Source: /opt/cvs/RandyThesis/Supporting/brk.dyn,v $
*/

DEFSYSTEM brk (DYNAMIC dt, DYNAMIC t, CONSTANT integrate)

FUN sqr (x) = x * x;

// The time range (s) during which Iapp will be applied to the system
PARAMETER t_on (1 TO 500 BY 1) = 1;//     "Beginning time for Iapp";
PARAMETER t_off (1 TO 20000 BY 1) = 2;//    "Ending time for Iapp";
PARAMETER Istep (0 TO 30 BY 0.01) = 11;
PARAMETER Iconst (0 TO 30 BY 0.01) = 5;
PARAMETER Iramp (-40 TO 40 BY 0.01) = 0;

// Conductances (mS/cm^2)
PARAMETER GNa (0 TO 200 BY 0.1) = 120;
PARAMETER GK_dr (0 TO 200 BY 0.1) = 100;
PARAMETER GCa_NS (0 TO 50 BY 0.1) = 14;
```

```
PARAMETER GCa_ND (0 TO 10 BY 0.01) = .03;
PARAMETER GK_CaS (0 TO 100 BY 0.1) = 5;
PARAMETER GK_CaD (0 TO 100 BY 0.1) = 1.1;
PARAMETER GCa_L ( 0 TO 10 BY 0.01) = 0.33;
PARAMETER gleak ( 0 TO 10 BY 0.01) = 0.51;

// Static Parameters
PARAMETER C ( 0.5 TO 2 BY 0.01) = 1;
CONSTANT gc = 0.1; // coupling conductance (mS/cm^2)
CONSTANT p = 0.1;
CONSTANT Kd = 0.2; // uM
CONSTANT f = 0.01; // percent free to bound Ca
CONSTANT alpha = 0.009; // mol/C/um
CONSTANT kca = 2; // Ca removal rate

// Half Activation voltages in mV, Slopes in MV, Time Constants in millisecon
PARAMETER Vhm ( -50 TO -10 BY 0.5) = -35;
PARAMETER Sm ( -10 TO -3 BY 0.1 ) = -7.8;
PARAMETER Vhh ( -50 TO -10 BY 0.5 ) = -55;
PARAMETER Sh ( 3 TO 10 BY 0.1 ) = 7;
PARAMETER Vhn ( -50 TO -10 BY 0.5 ) = -28;
PARAMETER Sn ( -20 TO -3 BY 0.1 ) = -15;
PARAMETER VhmN ( -50 TO -10 BY 0.5 ) = -30;
PARAMETER SmN ( -10 TO -3 BY 0.1 ) = -5;
PARAMETER VhhN ( -50 TO -10 BY 0.5 ) = -45;
PARAMETER ShN ( 3 TO 10 BY 0.1 ) = 5;
PARAMETER VhmL ( -50 TO -10 BY 0.5 ) = -40;
PARAMETER SmL ( -20 TO -3 BY 0.1 ) = -7;
CONSTANT TaumN = 4;
```

```
CONSTANT TauhN = 40;
CONSTANT TaumL = 40;

// Reversal potentials in mV
CONSTANT ENa = 55;
CONSTANT EK = -80;
PARAMETER ECa ( 60 TO 200 BY 1) = 80;
PARAMETER Eleak ( -80 TO -60 BY 1) = -60;


// State Variable Declaration
STATE Vs ( -90 TO 60 BY 0.001 ) = -60;
STATE Vd ( -90 TO 60 BY 0.001 ) = -60;
STATE h ( 0 TO 0.999 BY 0.0001 ) = 0.9;
STATE n ( 0 TO 0.999 BY 0.0001 ) = 0;
STATE mnS ( 0 TO 0.999 BY 0.0001 ) = 0;
STATE hnS ( 0 TO 0.999 BY 0.0001 ) = 0.9;
STATE mnD ( 0 TO 0.999 BY 0.0001 ) = 0;
STATE hnD ( 0 TO 0.999 BY 0.0001 ) = 0.9;
STATE ml ( 0 TO 0.999 BY 0.0001 ) = 0;
STATE CaS ( 0 TO 0.999 BY 0.000001 ) = 0;
STATE CaD ( 0 TO 0.999 BY 0.00000001 ) = 0;


// I_stim is 1V during the specified time range (t_on -- t_off),
// 0V otherwise
Iapp   = {Istep WHEN t > t_on AND t < t_off,
```

```
        0 OTHERWISE}
    + Iconst
    + {Iramp * t WHEN t < t_off,
        0 OTHERWISE};

// Steady state values
Tauh = 30/(exp((Vs+50)/15)+exp(-(Vs+50)/16));
Taun = 7/(exp((Vs+40)/40)+exp(-(Vs+40)/50));
minf = 1/(1+exp((Vs-Vhm)/Sm));
hinf = 1/(1+exp((Vs-Vhh)/Sh));
ninf = 1/(1+exp((Vs-Vhn)/Sn));
mnSinf = 1/(1+exp((Vs-VhmN)/SmN));
hnSinf = 1/(1+exp((Vs-VhhN)/ShN));
mnDinf = 1/(1+exp((Vd-VhmN)/SmN));
hnDinf = 1/(1+exp((Vd-VhhN)/ShN));
mlinf = 1/(1+exp((Vd-VhmL)/SmL));
INaS = GNa*minf*sqr(minf)*h*(Vs-ENa);
IKS = (GK_dr*sqr(sqr(n)) + GK_CaS*CaS/(CaS+Kd))*(Vs-EK);
ICaS = GCa_NS*mnS*mnS*hnS*(Vs-ECa);
IleakS = gleak*(Vs-Eleak);
IcouplingS = gc/p*(Vs-Vd);
IKD = GK_CaD*CaD/(CaD+Kd)*(Vd-EK);
ICaD = (GCa_ND*mnD*mnD*hnD+GCa_L*ml)*(Vd-ECa);
IleakD = gleak*(Vd-Eleak);
IcouplingD = gc/(1-p)*(Vd-Vs);

// Differential Equations
d(h) = (hinf-h)/Tauh;
d(n) = (ninf-n)/Taun;
```

```
    d(mnS) = (mnSinf-mnS)/TaumN;
    d(hnS) = (hnSinf-hnS)/TauhN;
    d(mnD) = (mnDinf-mnD)/TaumN;
    d(hnD) = (hnDinf-hnD)/TauhN;
    d(ml) = (mlinf-ml)/TaumL;
    d(CaS) = f*(-alpha*ICaS-kca*CaS);
    d(CaD) = f*(-alpha*ICaD-kca*CaD);
    d(Vs) = 1/C*(Iapp-INaS-IKS-ICaS-IleakS-IcouplingS);
    d(Vd) = 1/C*(-IKD-ICaD-IleakD-IcouplingD);


ENDSYSTEM brk;

MAIN
  FUN euler_integrate (dt, t, state, eq) =
    state + dt * eq(t);

  FUN integrate (dt, t, state, eq) =
        euler_integrate (dt, t, state, eq);

  STATE t (0 TO 1000 BY 0.001) = 0;
  PARAMETER dt (0.001 TO 0.1 BY 0.001) = 0.01;
  t=t+dt;

  SYSTEM neuron = new brk [dt, t, integrate];

    OUTPUT "neuron.V?", "t";
ENDMAIN;
```

REFERENCES

[1] "IEEE 754: Standard for binary floating point arithmetic," Tech. Rep. P754, IEEE, New York, 1985.

[2] Allen, F., Coteus, P., Crumley, P., Curioni, A., Denneau, M., Donath, W., Eleftheriou, M., Fitch, B., Fleischer, B., Georgiou, C., and others, "Blue Gene: a vision for protein science using a petaflop supercomputer," *IBM Systems Journal*, vol. 40, no. 2, pp. 310–327, 2001.

[3] Banerjee, P., Bagchi, D., Haldar, M., Nayak, A., Kim, V., and Uribe, R., "Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design," in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pp. 263–264, 2003.

[4] Banerjee, P., Chandy, J., Gupta, M., Hodges, U., J.G., H., Lain, A., Palermo, D., Ramaswamy, S., and Su, E., "An overview of the PARADIGM compiler for distributed-memory multicomputers," *IEEE Computer*, vol. 28, Oct 1995.

[5] Banerjee, P., Haldar, M., Nayak, A., Kim, V., Saxena, V., Parkes, S., Bagchi, D., Pal, S., Tripathi, N., Zaretsky, D., Anderson, R., and Uribe, J., "Overview of a compiler for synthesizing MATLAB programs onto FPGAs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, pp. 312–324, 2004.

[6] Banerjee, P., Shenoy, N., Choudhary, A., Hauck, S., Backmann, C., Haldar, M., Joisha, P., Jones, A., Kanhare, A., Nayak, A., Periyacheri, S., Walkden, M., and Zaretsky, D., "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, (Napa Valley, CA), pp. 39–48.

[7] Blake, J. J., Maguire, L. P., McGinnity, T. M., and McDaid, L. J., "Using xilinx FPGAs to implement neural networks and fuzzy systems," *Neural and Fuzzy Systems: Design, Hardware and Applications (Digest No: 1997/133), IEE Colloquium on*, p. 1, 1997.

[8] Boatin, W., "Characterization of neuron models," Master's thesis, Georgia Institute of Technology, 2005.

[9] Booth, V. and Rinzel, J., "A minimal, compartmental model for a dendritic origin of bistability of motoneuron firing patterns," *Journal of Computational Neuroscience.*, vol. 2, no. 4, pp. 299–312, 1995.

[10] Booth, V., Rinzel, J., and Kiehn, O., "Compartmental model of vertebrate motoneurons for ca2+-dependent spiking and plateau potentials under pharmacological treatment," *J Neurophysiol*, vol. 78, no. 6, pp. 3371–85, 1997.

[11] Botros, N. M. and Abdul-Aziz, M., "Hardware implementation of an artificial neural network using field programmable gate arrays (FPGA's)," *Industrial Electronics, IEEE Transactions on*, vol. 41, no. 6, p. 665, 1994.

[12] Bower, J. and Beeman, D., *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. New York: Springer-Verlag, second ed., 1998.

[13] Bragg, J., Brown, E., Hasler, P., and DeWeerth, S., "A silicon model of an adapting motoneuron," *IEEE International Symposium on Circuits and Systems*, vol. 4, pp. IV–261–IV–264, 2002.

[14] Brock, L., Coombs, J., and Eccles, J., "Action potentials of motoneurons with intracellular electrodes," *Proc. Univ. Otago Med. School*, vol. 29, pp. 14–15, 1951.

[15] Brown, B. O., Yin, M. L., and Cheng, Y., "DNA sequence matching processor using FPGA and JAVA interface," *Engineering in Medicine and Biology Society, 2004. EMBC 2004. Conference Proceedings. 26th Annual International Conference of the*, vol. 2, p. 3043, 2004.

[16] Budge, S. E., Mayampurath, A. M., and Solinsky, J. C., "Real-time registration and display of confocal microscope imagery for multiple-band analysis," *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, vol. 2, p. 1535, 2004.

[17] Burke, R. E., "Motor units in cat muscles: anatomical considerations in relation to motor unit types," *Advances in Neurology.*, vol. 36, pp. 31–45, 1982.

[18] Butera, Jr., R., Rinzel, J., and Smith, J. C., "Models of respiratory rhythm generation in the Pre-Bötzinger complex. I. Bursting pacemaker neurons," *J Neurophys*, vol. 82, no. 1, pp. 382–397, 1999.

[19] Butera, Jr., R., Rinzel, J., and Smith, J. C., "Models of respiratory rhythm generation in the Pre-Bötzinger complex. II. Populations of coupled pacemaker neurons," *J Neurophys*, vol. 82, no. 1, pp. 398–415, 1999.

[20] Cardoso, J. and Neto, H., "Compilation for FPGA-based reconfigurable hardware," *Design & Test of Computers, IEEE*, vol. 20, pp. 65–75, 2003.

[21] Changjian, G. and Hammerstrom, D., "Platform performance comparison of PALM network on Pentium 4 and FPGA," *Neural Networks, 2003. Proceedings of the International Joint Conference on*, vol. 2, pp. 995–1000, 2003.

[22] Church, A., *The calculi of lambda-conversion*. Princeton University Press, 1941.

[23] Coric, S., Latinovic, I., and Pavasovic, A., "A neural network FPGA implementation," in *Neural Network Applications in Electrical Engineering, 2000. NEUREL 2000. Proceedings of the 5th Seminar on*, pp. 117–120, 2000.

[24] Coric, S., Latinovic, I., and Pavasovic, A., "Design, implementation and comparison of three general-purpose neurons," in *Proceedings of 23rd International Conference on Microelectronics (MIEL 2002), 12-15 May 2002*, vol. vol.2, (Nis, Yugoslavia), pp. 601–4, 2002.

[25] Davoodi, R., Brown, I., Lan, N., Mileusnic, M., and Loeb, G., "An integrated package of neuromusculoskeletal modeling tools in SIMULINK," in *23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Oct 25-28 2001*, vol. 2, (Istanbul, Turkey), pp. 1205–1208, 2001.

[26] de Bruijn, N., "Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem," *Indag. Math*, vol. 34, no. 5, pp. 381–392, 1972.

[27] de Dinechin, F. and Tisserand, A., "Multipartite table methods," *Computers, IEEE Transactions on*, vol. 54, pp. 319–330, Mar. 2005.

[28] Detrey, J. and Dinechin, F., "Table-based polynomials for fast hardware function evaluation," in *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*, pp. 328–333, Jul. 2005.

[29] Diepenhorst, M., van Veelen, M., Nijhuis, J. A. G., and Spaanenburg, L., "Automatic generation of VHDL code for neural applications," in *International Joint Conference on Neural Networks(IJCNN'99)*, vol. 4, p. 2302, 1999.

[30] Drewes, R., "Modeling the brain with NCS and brainlab," *Linux Journal*, vol. 2005, no. 134, 2005.

[31] Farquhar, E. and P., H., "A bio-physically inspired silicon neuron," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 52, pp. 477–488, 2005.

[32] FitzHugh, R., "Impulses and physiological states in theoretical models of nerve membrane," *Biophys J*, vol. 1, pp. 445–466, 1961.

[33] Foster, W., Ungar, L., and Schwaber, J., "Significance of conductances in hodgkin–huxley models," *J Neurophysiol*, vol. 70, pp. 2502–2518, 1993.

[34] Girau, B., "FPNA: interaction between FPGA and neural computation," *International Journal of Neural Systems.*, vol. 10, no. 3, pp. 243–59, 2000.

[35] Golowasch, J., Goldman, M. S., Abbott, L., and Marder, E., "Failure of averaging in the construction of a conductance-based neuron model," *J Neurophysiol*, vol. 87, pp. 1129–1131, Feb. 2002.

[36] Graas, E., *Exploration of Alternatives to General-Purpose Computers in Neural Simulation*. PhD thesis, Georgia Institute of Technology, 2003.

[37] Graas, E., Brown, E., and Lee, R., "An FPGA-based approach to high-speed simulation of conductance-based neuron models," *Neuroinformatics*, vol. 2, no. 4, pp. 417–435, 2004.

[38] Haldar, M., Nagrajshenoy, A., Choudhary, A., and Banerjee, P., "FPGA hardware synthesis from MATLAB," in *14th International Conference on VLSI Design (VLSI DESIGN 2001), Jan 3-7 2001*, Proceedings of the IEEE International Conference on VLSI Design, (Bangalore, India), pp. 299–304, 2001.

[39] Hines, M. L. and Carnevale, N. T., "The NEURON simulation environment," *Neural Comp.*, vol. 9, no. 6, pp. 1179–1209, 1997.

[40] Hodgkin, A. and Huxley, A., "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, pp. 500–544, 1952.

[41] Hounsgaard, J., Hultborn, H., Jerspersen, B., and Kiehn, O., "Intrinsic membrane properties causing a bistable behaviour of alpha-motoneurons.," *Experimental Brain Research*, vol. 55, no. 2, pp. 391–4, 1984.

[42] Ito, M., Takagi, N., and Yajima, S., "Efficient initial approximation and fast converging methods for division and square root," *arith*, vol. 00, p. 2, 1995.

[43] Johnston, D. and Wu, S., *Foundations of Cellular Neurophysiology*. Cambridge, Massachusetts: MIT Press, 1995.

[44] Jones, A., Bagchi, D., Pal, S., Tang, X., Choudhary, A., and Banerjee, P., "PACT HDL: a C compiler targeting ASICs and FPGAs with power and performance optimizations," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Proceedings of the 2002 international conference on compilers, architecture, and synthesis for embedded systems*, (Grenoble, France), pp. 188–197, ACM Press, 2002.

[45] Jones, S. M. and Lee, R. H., "Fast Amplification of Dynamic Synaptic Inputs in Spinal Motoneurons In Vivo," *J Neurophysiol*, p. 00537.2006, 2006.

[46] Kamalizad, A. H., Pan, C., and Bagherzadeh, N., "Fast parallel fft on a reconfigurable computation platform," *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*, p. 254, 2003.

[47] Krishnamurthy, R., Yalamanchill, S., Schwan, K., and West, R., "Architecture and hardware for scheduling gigabit packet streams," *High Performance Interconnects, 2002. Proceedings. 10th Symposium on*, p. 52, 2002.

[48] Kuo, C. C. and Bean, B. P., "Na^+ channels must deactivate to recover from inactivation," *Neuron*, vol. 12, no. 4, pp. 819–29, 1994.

[49] Landin, P., "A correspondence between ALGOL 60 and church's lambda-notation," *Communications of the ACM*, vol. 8, pp. 89–101, 65.

[50] Lee, D. C., Harper, S. J., Athanas, P. M., and Midkiff, S. F., "A stream-based reconfigurable router prototype," *Communications, 1999. ICC '99. 1999 IEEE International Conference on*, vol. 1, p. 581, 1999.

[51] Lee, R. H. and Heckman, C. J., "Essential role of a fast persistent inward current in action potential initiation and control of rhythmic firing," *Journal of Neurophysiology.*, vol. 85, no. 1, pp. 472–5, 2001.

[52] Lee, R. H., Kuo, J. J., Jiang, M. C., and Heckman, C. J., "Influence of active dendritic currents on input-output processing in spinal motoneurons in vivo," *J Neurophysiol*, vol. 89, no. 1, pp. 27–39, 2003.

[53] Lee, R. H. and Heckman, C. J., "Adjustable amplification of synaptic input in the dendrites of spinal motoneurons in vivo," *J. Neurosci.*, vol. 20, no. 17, pp. 6734–6740, 2000.

[54] Liddell, E. and Sherrington, C., "Reflexes in response to stretch (myotatic reflexes)," *Proc R Soc Lond B Biol Sci*, vol. 96, pp. 212–242, 1924.

[55] Mak, T., Rachmuth, G., Lam, K. P., and Poon, C.-S., "Field programmable gate array implementation of neuronal ion channel dynamics," *Neural Engineering, 2005. Conference Proceedings. 2nd International IEEE EMBS Conference on*, pp. 144–8, 2005.

[56] Moraru, I., Schaff, J., Slepchenko, B., and Loew, L., "The Virtual Cell: An Integrated Modeling Environment for Experimental and Computational Cell Biology," *Annals of the New York Academy of Sciences*, vol. 971, no. 1, p. 595, 2002.

[57] Muchnick, S., *Advanced compiler design and implementation*. San Francisco: Morgan Kaufmann, 1997.

[58] Nagumo, J., Arimoto, S., and Yoshizawa, S., "An active pulse transmission line simulating nerve axon," *Proc. IRE*, vol. 50, pp. 2061–2070, 1962.

[59] Oliver, T., Schmidt, B., Maskell, D. L., and Vinod, A. P., "A reconfigurable architecture for scanning biosequence databases," *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, p. 4799, 2005.

[60] Omondi, A. R. and Rajapakse, J. C., "Neural networks in FPGAs," *Neural Information Processing, 2002. ICONIP '02. Proceedings of the 9th International Conference on*, vol. 2, p. 954, 2002.

[61] Pearson, M., Gilhespy, I., Gurney, K., Melhuish, C., Mitchinson, B., Nibouche, M., and Pipe, A., "A real-time, FPGA based, biologically plausible neural network processor," in *Artificial Neural Networks: Biological Inspirations - ICANN 2005: 15th International Conference*, vol. 3697, (Warsaw, Poland), 2005.

[62] Peck, C., Caglar, S., Kozloski, J., Rao, R., and Cecchi, G., "Large-scale neocortical simulation on BlueGene/L," in *Society for Neuroscience*, (688.5), 2006.

[63] Pradeep, R., Vinay, S., Burman, S., and Kamakoti, V., "Fpga based agile algorithm-on-demand coprocessor," *Design, Automation and Test in Europe, 2005. Proceedings*, p. 82, 2005.

[64] Prinz, A. A., Billimoria, C. P., and Marder, E., "Alternatives to hand-tuning conductance-based models: Construction and analysis of databases of model neurons," *J Neurophysiol*, vol. 90, pp. 3998–4015, Aug. 2003.

[65] Raikov, I., Preyer, A., and Butera, R. J., "MRCI: A flexible real-time dynamic clamp system for electrophysiology experiments," *Journal of Neuroscience Methods.*, vol. 132, Jan 2004.

[66] Resat, M. S., Solinsky, J. C., Wiley, H. S., Perrine, K. A., Seim, T. A., and Budge, S. E., "3-d multispectral monitoring of living cell signaling using confocal imaging and fpga processing," *Biomedical Imaging: Macro to Nano, 2004. IEEE International Symposium on*, p. 680, 2004.

[67] Ros, E., Ortigosa, E. M., Agfs, R., Carrillo, R., Prieto, A., and Arnold, M., "Spiking neurons computing platform," in *Computational Intelligence and Bioinspired Systems: 8th International Workshop on Artificial Neural Networks*, vol. 3512, (Barcelona, Spain), p. 471, June 2005.

[68] Saldaña, R. and Yu, W., "Cellular automata explorations on a beowulf cluster computer," *Proceedings of Cellular Automata Symposium*, 2001.

[69] Savageau, M. and Voit, E., "Recasting nonlinear differential equations as s-systems: a cononical nonlinear form," *Math. biosci.*, vol. 87, no. 1, pp. 83–115, 1987.

[70] Schulte, M. and Stine, J., "Approximating elementary functions with symmetric bipartite tables," *IEEE Transactions on Computers*, vol. 48, pp. 842–847, Aug. 1999.

[71] Schwindt, P. C. and Crill, W. E., "Properties of a persistent inward current in normal and TEA-injected motoneurons," *Journal of Neurophysiology.*, vol. 43, no. 6, pp. 1700–24, 1980.

[72] Segev, I., Fleshman, J. W., J., and Burke, R. E., "Computer simulation of group ia epsps using morphologically realistic models of cat alpha-motoneurons," *Journal of Neurophysiology.*, vol. 64, no. 2, pp. 648–60, 1990.

[73] Shenoy, U., Choudhary, A., and Banerjee, P., "Symphany: A system for automatic synthesis of adaptive systems," Tech. Rep. CPDC-TR-9903-002, Center for Parallel and Distributed Computing, Northwestern University, Mar 1999.

[74] Sherrington, C., "On the anatomical constitution of nerves of skeletal muscles; with remarks on recurrent fibres in the ventral spinal nerve-root," *J Physiol*, vol. 17, pp. 211–258, 1894.

[75] Sherrington, C., *Integrative Action of the Nervous System*. New Haven, CT: Yale Univ. Press, 1906.

[76] Shi, C. and Brodersen, R. W., "An automated floating-point to fixed-point conversion methodology," in *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on*, vol. 2, pp. 529–532, 2003.

[77] Shi, C., Hwang, J., McMillan, S., Root, A., and Singh, V., "A system level resource estimation tool for FPGAs," in *International Conference, Field Programmable Logics and Its Applications*, 2004.

[78] Shi, C., "Statistic method for floating-point to fixed-point conversion," Master's thesis, University of California, Berkeley, 2002.

[79] Shi, C., *Floating-point to Fixed-point Conversion*. PhD thesis, University of California, Berkeley, 2004.

[80] Snider, R. K., Lukes, A. J., Zhu, Y., and Miller, J. P., "A digital methodology integrating experimental and theoretical neuroscience," in *Neural Engineering, 2003. Conference Proceedings. First International IEEE EMBS Conference on*, pp. 376–379, 2003.

[81] Sorensen, M., *Functional Consequences of Model Complexity in Hybrid Neural-Microelectronic Systems*. PhD thesis, Georgia Institute of Technology, 2005.

[82] Stine, J. and Schulte, M., "The symmetric table addition method for accurate function approximation," *Journal of VLSI Signal Processing*, vol. 21, 1999.

[83] Thibault, S. and Pellerin, D., "Optimizing impulse C code for performance," Tech. Rep. IATAPP-102, Impulse Accelerated Technologies, 2004.

[84] van Daalen, M., Jeavons, P., and Shawe-Taylor, J., "A stochastic neural architecture that exploits dynamically reconfigurable FPGAs," in *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pp. 202–211, 1993.

[85] Vetter, P., Roth, A., and Hausser, M., "Propagation of action potentials in dendrites depends on dendritic morphology," *J Neurophysiol*, vol. 85, no. 2, pp. 926–937, 2001.

[86] Waldemark, J., Millberg, M., Lindblad, T., Waldemark, K., and Becanovic, V., "Implementation of a pulse coupled neural network in FPGA," *International Journal of Neural Systems.*, vol. 10, no. 3, pp. 171–7, 2000.

[87] Weinstein, R. K. and Lee, R. H., "Computationally intensive motoneuron modeling in FPGAs," in *Society for Neuroscience*, (San Diego, CA), 2004.

[88] Weinstein, R. K. and Lee, R. H., "Design of high performance physiologically-complex motoneuron models in FPGAs," in *Neural Engineering, 2005. Conference Proceedings. 2nd International IEEE EMBS Conference on*, (Washington DC), pp. 526–528, 2005.

[89] Weinstein, R. K. and Lee, R. H., "Architectures for high-performance FPGA implementations of neural models," *Journal of Neural Engineering*, vol. 3, pp. 21–34, March 2006.

[90] Weinstein, R., Reid, M., and Lee, R., "Methodology and design flow for assisted neural-model implementations in FPGAs," *IEEE Transactions on Neural Systems & Rehabilitation*, in press expected March 2007.

[91] Wilson, E., Harris, F., and Goodman, P., "Implementation of a biologically realistic parallel neocortical-neural network simulator," in *Proceedings of the Tenth SIAM Conf. on Parallel Processing for Scientific Computing*, 2001.

[92] Xiaoyang, Z., Chao, C., and Qianling, Z., "A reconfigurable public-key cryptography coprocessor," *Advanced System Integrated Circuits 2004. Proceedings of 2004 IEEE Asia-Pacific Conference on*, p. 172, 2004.

[93] Xilinx, *Virtex-4 User Guide*, Mar. 2006.

[94] Yao, H. H. and Swartzlander, E. E., J., "Serial-parallel multipliers," *Signals, Systems and Computers, 1993. 1993 Conference Record of The Twenty-Seventh Asilomar Conference on*, p. 359, 1993.

[95] Zaretsky, D., Mittal, M., Tang, X., and Banergee, P., "Overview of the FREEDOM compiler for mapping DSP software to FPGAs," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 37–46, 2004.

VITA

Randall K. Weinstein received his B.S. in Computer Engineering for the Georgia Institute of Technology in 2001. Throughout college, he worked as a design engineer co-op for Atmel Corporation designing structured ASICs. Post-graduation, he was employed as a digital design engineer for the Wireless Communications and Networking Division of Agere Systems. In January 2003, he began his graduate work and is pursuing a Ph.D. degree in the Interdisciplinary Bioengineering program at the Georgia Institute of Technology.