INSTITUTE OF PHYSICS PUBLISHING JOURNAL OF NEURAL ENGINEERING

J. Neural Eng. **3** (2006) 21–34 doi:10.1088/1741-2560/3/1/003

Architectures for high-performance FPGA implementations of neural models

Randall K Weinstein[1] **and Robert H Lee**[1,2]

[1] Laboratory for Neuroengineering, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA
[2] Wallace H Coulter Department of Biomedical Engineering, Emory University, Atlanta, GA 30322, USA

E-mail: rlee2@emory.edu

Received 17 August 2005
Accepted for publication 2 November 2005
Published 19 December 2005
Online at stacks.iop.org/JNE/3/21

Abstract
As the complexity of neural models continues to increase (larger populations, varied ionic conductances, more detailed morphologies, etc) traditional software-based models have difficulty scaling to reach the performance levels desired. This paper describes the use of FPGAs, or field programmable gate arrays, to easily implement a wide variety of neural models with the performance of custom analogue circuits or computer clusters, the reconfigurability of software, and at a cost rivalling personal computers. FPGAs reach this level of performance by enabling the design of neural models as parallel processed data paths. These architectures provide for a wide range of single-compartment, multi-compartment and population models to be readily converted to FPGA implementations. Generalized architectures are described for the efficient modelling of a first-order, nonlinear differential equation in throughput maximizing or latency minimizing data-path configurations. The homogeneity of population and multicompartment models is exploited to form deep pipelines for improved performance. Limitations of FPGA architectures and future research areas are explored.

1. Introduction

Today's neural circuit modeller is often constrained by the limited performance of conventional personal computers. While computing power might double every 18 months according to Moore's law, neural modellers often change the computational requirements by orders of magnitude. Population models can always simulate more neurons or morphological models can always simulate more compartments. The electrophysiologist can always simulate additional conductances, while the systems biologist can add additional cellular processes to a model. In general, current neural circuit modellers make substantial tradeoffs by limiting complexity to reduce processing requirements. FPGAs, or field programmable gate arrays, offer a high performance, configurable, scalable platform for enabling current and next generation neural models.

FPGAs are specialized integrated circuits that are designed with 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 coprocessors (Pradeep *et al* 2005, Xiaoyang *et al* 2004), DSP processors (Kamalizad *et al* 2003, Walke *et al* 2000) or stream processors (Krishnamurthy *et al* 2002, Lee *et al* 1999).

FPGAs have been less commonly used in bio-related fields, with several exceptions. Protein (Oliver *et al* 2005) and DNA (Brown *et al* 2004) 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 (Budge *et al* 2004, Resat *et al* 2004). Most modelling applications on FPGAs have been limited to studying neural networks consisting of reduced neurons (Blake *et al* 1997, Botros and Abdul-Aziz 1994, Changjian and Hammerstrom 2003, Omondi and Rajapakse 2002). More complex models have been implemented in analogue VLSI (Bragg *et al* 2002, Jung *et al* 2001). Recently, in our laboratory, several implementations of conductance-based neural models have emerged including Hodgkin and Huxley's (1952) and Booth and Rinzel's (1995) models (Graas *et al* 2004), and a ten-compartment motoneuron model (Weinstein and Lee 2005). However, each of these designs was somewhat specific to the model being implemented. This paper 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.

2. Methods

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^{-1} analogue-to-digital converters (ADC), two 14 bit 160 MS s^{-1} digital-to-analogue converters (DAC) and 1 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 versus 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 modeller.

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 partial products (Yao and Swartzlander 1993).

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 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 1). Additional arithmetic steps are done biased towards the paths of lesser latency. If each of the addition operations has a latency of one cycle, then the skewing of the addition steps away from the multipliers enables a balanced tree of three to five 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 ($k, e \rightarrow y$).

2.3. Lookup 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 expression with a difficult-to-evaluate closed-form solution. 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 Boltzmann's equation as defined by:

$$m_\infty = \frac{1}{1 + \exp\left(\frac{V-\theta}{\sigma}\right)} \qquad (1)$$

where θ is the half activation voltage of the gate, σ is a measure of the slope of the sigmoid and V is the membrane potential. This equation is difficult to solve directly for two reasons.

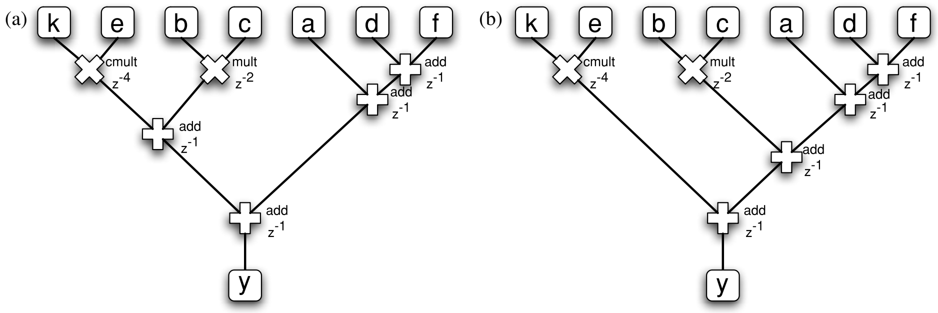

Figure 1. Example expression trees for the equation $a + bc + d + ke + f$ where k is a constant. Two operand multiply operations are shown with two units of delay while constant multiplies have four units of delay. All adders are given one unit delay. Additional pipeline registers can be added arbitrarily 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.

First, the exponential has no simple closed-form solution that is efficient in an FPGA. Second, the inverse function is as difficult as a division, which is generally solved iteratively, not directly like a multiplication. Division by σ can be simplified by reframing the expression as multiplication by a new parameter, $1/\sigma$.

Since equation (2) is difficult to evaluate directly in hardware, it is a good candidate for a lookup table. A ROM indexed by V 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 input to an address for the ROM. For the general case:

$$\text{adder}(x) = (x - \min(x)) \cdot \frac{2^n - 1}{\min(x) + \max(x)} \qquad (2)$$

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 × 36 bits, 1k × 18 bits, 2k × 9 bits, 4k × 4 bits, 8k × 2 bits or 16k × 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 × 18 bit configuration.

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 2. Daisy chained logic blocks (arbitrary length chain of six 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 2's 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 2(a). The area per operation was split into two components, registers, or flip-flops, and look-up tables (LUTs). Each value plotted (see figure 2(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.

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 exist four slices and two tristate buffers. Each slice contains two 1 bit registers, two 1 bit

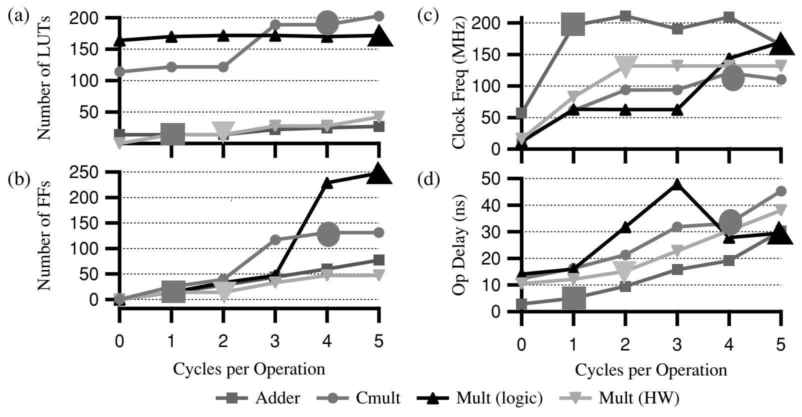

Figure 2. Area versus performance tradeoff. (a), (b) Performance and (c), (d) area utilization of basic arithmetic operations. The benchmarks were performed with 14 bit operands and 14 bit output. The inputs and outputs were double buffered and assigned to external pins on the FPGA. All performance results 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 versus performance tradeoff. These plots also demonstrate the performance and area advantage of using adders and hardware embedded multipliers versus constant multipliers and logic-based multipliers. When throughput is to be maximized above all else, then adders and logic-based multipliers are preferred.

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[a]	135 slices	0 slices

[a] The peak throughput of the multiplier is achieved when implementing in logic while the minimum latency is achieved using an embedded multiplier block.

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 CLB's.) 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. These data show 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 modeller when implementing area-constrained designs (Shi *et al* 2004).

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 2, each operation can be represented by the following expressions, where x, y are intermediate values, states or parameters, k is a constant, and p is the stage in the execution pipeline.

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

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

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 the time step 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 1 ms (even if they were all placed on the chip simultaneously). Thus, for the sake of convenience, the data presented here are generally from emulation of the FPGA directly in Simulink.

3. The base architecture

3.1. *FitzHugh–Nagumo*

For the 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 (FitzHugh 1961, Nagumo *et al* 1962) is a reduced, dimensionless representation of the Hodgkin and Huxley model (Hodgkin and Huxley 1952). 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_m), 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:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = u - \frac{1}{3}u^3 - w + I \qquad (4)$$

$$\frac{\mathrm{d}w}{\mathrm{d}t} = \varepsilon(b_0 + b_1 u - w) \qquad (5)$$

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 equation (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.

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 equations (4) and (5), the left-hand side is the differential term and the right-hand side is denoted by 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 equations (4) and (5), respectively, as follows:

$$f(u, w) = u - \tfrac{1}{3}u^3 - w + I$$
$$g(u, w) = \varepsilon(b_0 + b_1 u - w)$$

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 paper, this separated data path provides a general case for rapid mapping into various architectures including population and multicompartment modelling.

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:

$$u[n+1] = u[n] + \Delta t\left(u[n] - \tfrac{1}{3}u[n]^3 - w[n] + I\right)$$
$$w[n+1] = w[n] + \Delta t \cdot \varepsilon(b_0 + b_1 u - w).$$

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 1.

$$f(u, w) = \left[((u - w) + I) - \left((u \cdot u) \cdot \left(\tfrac{1}{3}u\right)\right)\right]$$
$$g(u, w) = [\varepsilon \cdot ((b_0 - w) + (b_1 \cdot u))].$$

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 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 equation (3) to the redefined functions f and g and obtain:

$$f(u[p], w[p]) = \mathrm{Sub}(\mathrm{Add}(\mathrm{Sub}(u[p], w[p]), I[p]), \ldots, \mathrm{Mult}(\mathrm{Mult}(u[p], w[p]), \mathrm{CMult}(1/3, u[p])))$$

$$g(u[p], w[p]) = \mathrm{Mult}(\varepsilon[p], \mathrm{Add}(\mathrm{Sub}(b_0[p], w[p]), \ldots, \mathrm{Mult}(b_1[p], u[p]))).$$

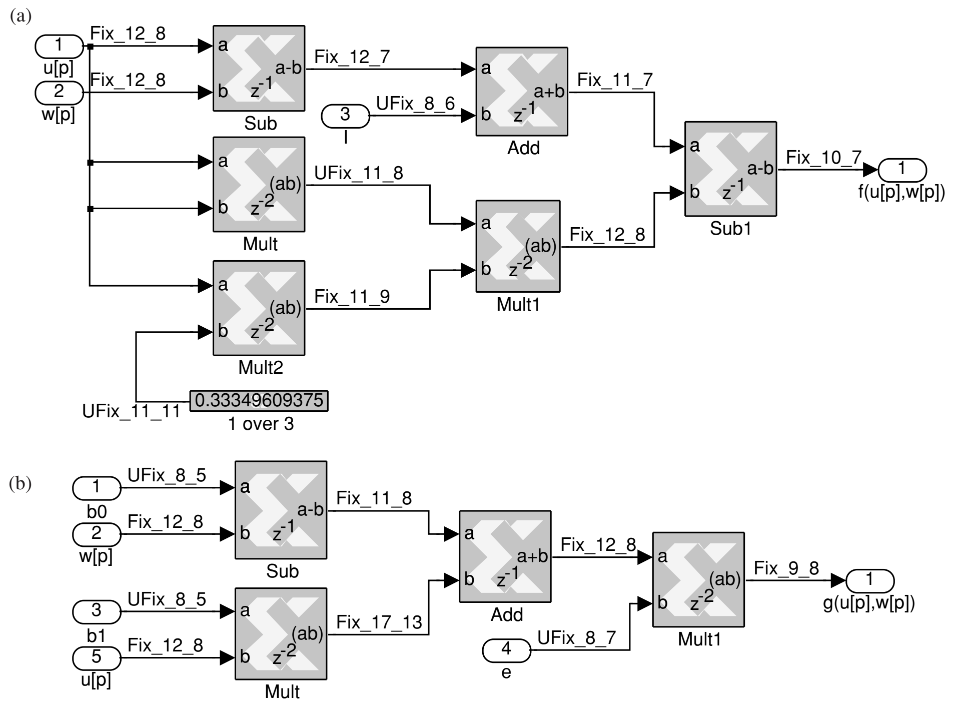

Figure 3. Simulink block diagram of intermediate calculations (a) $f(u, w)$ and (b) $g(u, w)$. Each operation is identified by the label on the block. All fixed-point data types are labelled on the wires interconnecting the blocks, where the first portion (Fix/UFix) defines the value to be signed or 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.

Evaluating these mappings yields equations (6) and (7). This representation is useful to describe the overall delay through the data path, in this example, six cycles. Since there is a skew between the shortest latency paths and the longest latency paths (from one cycle to six 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.

$$f(u[p], w[p]) = \begin{bmatrix} ((u[p-1] - w[p-1]) + I[p-1]) \cdots \\ ((u[p-6] \cdot u[p-6]) \cdot (1/3u[p-4])) \end{bmatrix} \tag{6}$$

$$g(u[p], w[p]) = \left[\varepsilon[p-3] \cdot \begin{pmatrix} (b_0[p-3] - w[p-3]) + \cdots \\ (b_1[p-6] \cdot u[p-6]) \end{pmatrix} \right]. \tag{7}$$

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 3 shows the System Generator data path corresponding to equations (6) and (7). For these different architectures, only the implementation of the state variables will change as is expounded upon in the following sections.

4. Single model, single unit case

4.1. Multi-cycle architecture

The general model simulator will execute a single version of a model according to a set protocol. The modeller will often run simulations to manually tune parameters, trying to replicate a particular behaviour. 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.

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f(u, w) \rightarrow u[n+1] = u[n] + \Delta t \cdot f(u[n], w[n]) \tag{8}$$

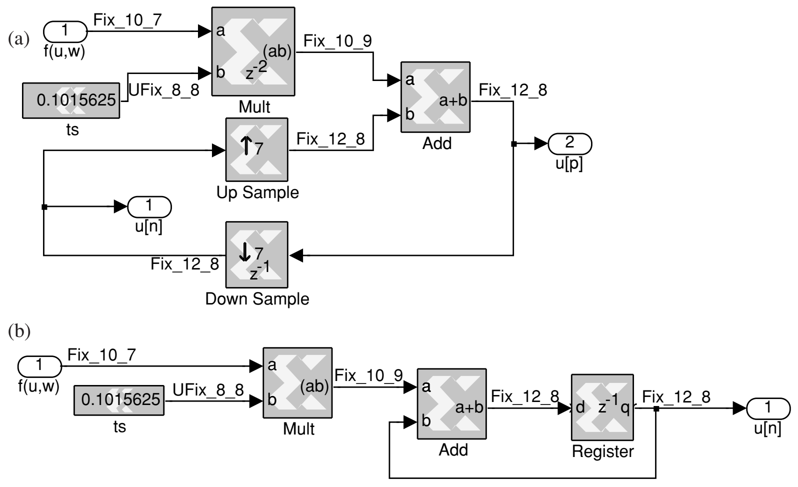

Figure 4. Simulink block diagram of state calculations for single unit models. (a) The multi-cycle approach to single unit forward-Euler integration. The intermediate calculation is scaled by the time constant and added to the previous state. The output of the state, $u[n]$, is clocked once for every 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.

$$\frac{\mathrm{d}w}{\mathrm{d}t} = g(u, w) \rightarrow w[n+1] = w[n] + \Delta t \cdot g(u[n], w[n]). \tag{9}$$

Equations (8) and (9) representing the state calculation can be combined with the data path formulation in equations (6) and (7) to generate the full expression for the model. The data path needs to be modified to include the time-step multiplication adding another two to four delays to the data path. The total delay for the data path is increased to seven clock cycles. The translation of the state equation into hardware is only a single register clocked at the slower clock rate. Equations (10) and (11) 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.

$$u[n+1, p] = u[n, p] + \cdots$$
$$\Delta t \cdot \begin{bmatrix} ((u[n, p-2] - w[n, p-2]) + I[n, p-2]) - \cdots \\ ((u[n, p-7] \cdot u[n, p-7]) \cdot (1/3u[n, p-5])) \end{bmatrix} \tag{10}$$

$$w[n+1, p] = w[n, p] + \cdots$$
$$\Delta t \cdot \left[\varepsilon[n, p-3] \cdot \begin{pmatrix} (b_0[n, p-3] - w[n, p-3]) + \cdots \\ (b_1[n, p-6] \cdot u[n, p-6]) \end{pmatrix} \right]. \tag{11}$$

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 the 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:

$$u[n+1] = u[n] + \Delta t \begin{bmatrix} ((u[n] - w[n]) + I[n]) - \cdots \\ ((u[n] \cdot u[n]) \cdot (1/3u[n])) \end{bmatrix}$$

$$w[n+1] = w[n] + \Delta t \cdot \left[\varepsilon[n] \cdot \begin{pmatrix} (b_0[n] - w[n]) + \cdots \\ (b_1[n] \cdot u[n]) \end{pmatrix} \right].$$

The forward-Euler method is relatively simple to implement within System Generator, taking only four blocks as shown in figure 4(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.

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 $\varepsilon \cdot \Delta t$. This is only possible in the trivial case of Euler integration with fixed step sizes.

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

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 of 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 increases, 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 modeller, 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 in figure 5 was used for testing and synthesizing the single-cycle and multi-cycle models. 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 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.

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 ten cycles until the first output appears, ten simultaneous models can be executed without a loss of performance. The data path would produce the ten 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 you are interested in simulating, for example, all the neurons in a particular nuclei or circuit or (2) when the number of simultaneous simulations is flexible and more is better, such as in automated parameter searching or population modelling. The first scenario applies more constraints to the model and produces a very 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.

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

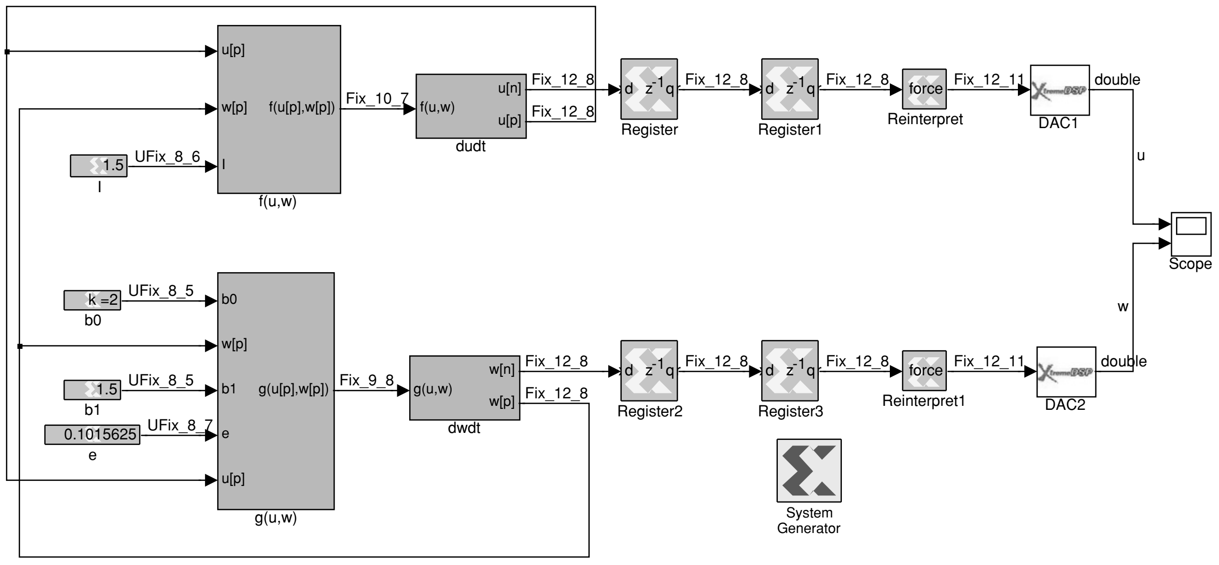

Figure 5. Top-level view of the FitzHugh–Nagumo model. Parameters are defined in Xilinx constant blocks (on the left) moving into the intermediate calculation. The states are evaluated next, with feedback paths to the inputs of the intermediate calculations. The outputs, u and w are double-buffered for performance and scaled for analogue output via the on-board data converters (on the right).

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 equation (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 than 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 than the critical path. Adding extra delays such that all paths are balanced is 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 1(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.

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 the previous work (Graas *et al* 2004), 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

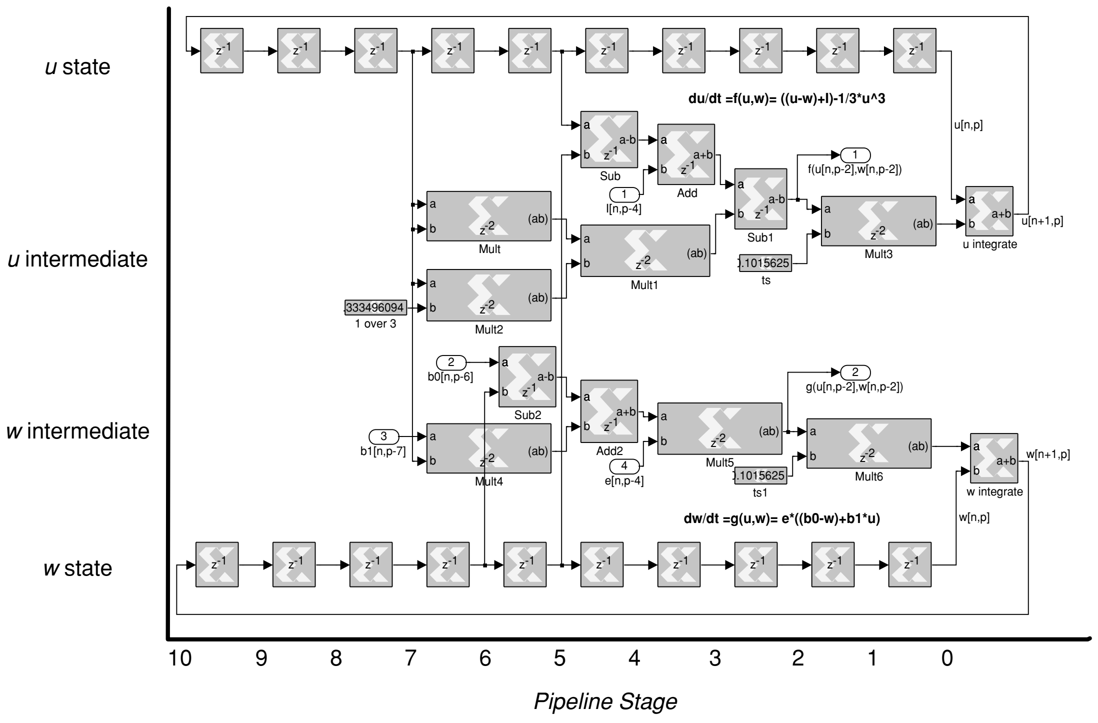

Figure 6. Simulink block diagram of a multiple unit model. Ten iterations of the FitzHugh–Nagumo model are run simultaneously through the constructed ten stage pipeline. All blocks are depicted to be in scale with respect to cycle latency. Since the data path requires only seven cycles per iteration, the state register chain must be at least seven stages within this architecture. It was chosen to be ten for this example.

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 six cycles, then the input of that state should be tapped six 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 6.

5.3. Shared versus 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 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 $I[\mathrm{p}-d]$, 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 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 7.

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.

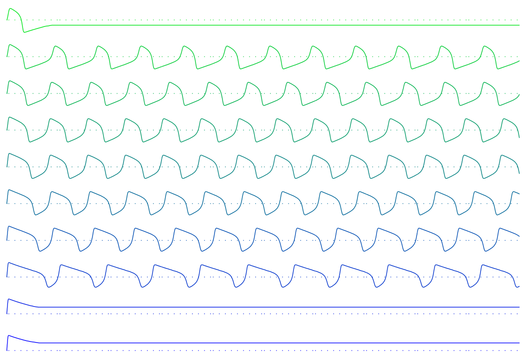

Figure 7. Output traces from a cycle-accurate, fixed-point simulation within Simulink of ten concurrently executing Fitzhugh–Nagumo models. The traces were generated from the model shown in figure 6. All parameters were kept constant with the exception of I, which was varied from 0.99 to 2.97 with a 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.

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 the previous work within this laboratory (Weinstein and Lee 2005).

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 modelling 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-centre 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 the voltage states from the adjacent units. 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 unit's 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.

7. Discussion

This paper 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.

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,

Table 3. Calculations to determine range values per operation type.

$R^* = R_1 \bigcirc R_2$	High range value (H^*)	Low range value (L*)
Addition '+'	$H_1 + H_2$	$L_1 + L_2$
Subtraction '−'	$\max(H_1 - L_2, L_1 - H_2)$	$\min(H_1 - L_2, L_1 - H_2)$
Multiplication '∗'	$\max(L_1{\cdot}L_2, L_1{\cdot}H_2, H_1{\cdot}L_2, H_1{\cdot}H_2)$	$\min(L_1{\cdot}L_2, L_1{\cdot}H_2, H_1{\cdot}L_2, H_1{\cdot}H_2)$
Division '/'[a]	$\max(L_1/L_2, L_1/H_2, H_1/L_2, H_1/H_2)$	$\min(L_1/L_2, L_1/H_2, H_1/L_2, H_1/H_2)$

[a] The division assumes that the range of the inputs does not cross zero.

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. On the other end of the scale, 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 (Shi *et al* 2004). 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 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. We define S to denote a signed number (versus 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 ith 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 are determined by:

$$\begin{aligned} S_i &= \begin{cases} 1, & L_i < 0 \\ 1, & H_i < 0 \\ 0, & \text{otherwise} \end{cases} \\ &\text{Given : max int} = \max\lfloor |L_i| \rfloor, \lfloor |H_i| \rfloor \\ Z_i &= \begin{cases} \lfloor \log_2(\text{maxint}) + 1 \rfloor & \text{maxint} > 0 \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \tag{12}$$

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 and 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 $4f$ 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 combining 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 behaviour.

The remaining class of errors, quantization errors, is substantially more difficult to reason through intuitively and requires 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.

7.2. External interfacing

By utilizing the techniques described in this text, a modeller 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 simulation 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 analogue data converters, inputs/outputs can be transferred to the FPGA at a 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 Xtreme-DSP 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.

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 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 techniques described in this paper. However, future work is needed to find the techniques to enable hundreds to thousands of simultaneous systems to run.

7.4. Autogeneration

While the architecture presented in this paper permits the construction of complex neural models, the difficulty of manually creating these models grows dramatically with the complexity of the model. The System Generator environment is not a typical modelling language and lacks general programmatic constructs such as functions, procedures and iterations. Modularity is therefore difficult to achieve requiring the model as a whole to be fully verified, independent of its parts. Additionally, the masking of parameters within each schematic block makes verification by inspection difficult. Any alterations to a model can typically require long testing/verification cycles. Consequently, the next logical step in the evolution of the FPGA-as-neural-simulation platform is the development of a 'compiler' to automatically generate the necessary blocks and possibly include protocol and other input–output harnesses.

Acknowledgments

This work was supported by the Human Brain Project via NINDS, NIMH and NIBIB (NS046851). The authors would also like to thank C T Church for his valuable comments and corrections throughout the drafting of this paper.

References

Blake J J, Maguire L P, McGinnity T M and McDaid L J 1997 Using Xilinx FPGAs to implement neural networks and fuzzy systems *IEE Colloquium on Neural and Fuzzy Systems: Design, Hardware and Applications (*Digest No: *1997/133)* p 1

Booth V and Rinzel J 1995 A minimal, compartmental model for a dendritic origin of bistability of motoneuron firing patterns *J. Comput. Neurosci.* **2** 299–312

Botros N M and Abdul-Aziz M 1994 Hardware implementation of an artificial neural network using field programmable gate arrays (FPGA's) *IEEE Trans. Ind. Electron.* **41** 665

Bragg J A, Brown E A, Hasler P and DeWeerth S P 2002 A silicon model of an adapting motoneuron *IEEE Int. Symp. Circuits Syst.* **4** IV-261–IV-4

Brown B O, Yin M L and Cheng Y 2004 DNA sequence matching processor using FPGA and JAVA interface *Conf. Proc.: 26th Annual International Conf. of the Engineering in Medicine and Biology Society, EMBC 2004* vol 2 p 3043

Budge S E, Mayampurath A M and Solinsky J C 2004 Real-time registration and display of confocal microscope imagery for multiple-band analysis *Conference Record of the 38th Asilomar Conf. on Signals, Systems and Computers 2004* vol 2 p 1535

Changjian G and Hammerstrom D 2003 Platform performance comparison of PALM network on Pentium 4 and FPGA *Neural Networks 2003: Proc. Int. Joint Conf.* **2** 995

FitzHugh R 1961 Impulses and physiological states in theoretical models of nerve membrane *Biophys. J.* **1** 445–66

Graas E L, Brown E A and Lee R H 2004 An FPGA-based approach to high-speed simulation of conductance-based neuron models *Neuroinformatics* **2** 417–35

Hodgkin A L and Huxley A F 1952 A quantitative description of membrane current and its application to conduction and excitation in nerve *J. Physiol.* **117** 500–44

Jung R, Brauer E J and Abbas J J 2001 Real-time interaction between a neuromorphic electronic circuit and the spinal cord *IEEE Trans. Neural Syst. Rehabil. Eng.* **9** 319–26

Kamalizad A H, Pan C and Bagherzadeh N 2003 Fast parallel FFT on a reconfigurable computation platform *Proc. 15th Symp. on Computer Architecture and High Performance Computing* 254

Krishnamurthy R, Yalamanchill S, Schwan K and West R 2002 Architecture and hardware for scheduling gigabit packet streams *Proc. 10th Symp. on High Performance Interconnects* p 52

Lee D C, Harper S J, Athanas P M and Midkiff S F 1999 A stream-based reconfigurable router prototype *IEEE International Conf. on Communications: ICC '99* vol 1 p 581

Nagumo J, Arimoto S and Yoshizawa S 1962 An active pulse transmission line simulating nerve axon *Proc. IRE* **50** 2061–70

Oliver T, Schmidt B, Maskell D L and Vinod A P 2005 A reconfigurable architecture for scanning biosequence databases *IEEE Int. Symp. on Circuits and Systems: ISCAS 2005* p 4799

Omondi A R and Rajapakse J C 2002 Neural networks in FPGAs *Proc. 9th Int. Conf. on Neural Information Processing: ICONIP '02* vol 2 p 954

Pradeep R, Vinay S, Burman S and Kamakoti V 2005 FPGA based agile algorithm-on-demand coprocessor *Proc. Design, Automation and Test in Europe* p 82

Resat M S, Solinsky J C, Wiley H S, Perrine K A, Seim T A and Budge S E 2004 3-D multispectral monitoring of living cell signaling using confocal imaging and FPGA processing *IEEE Int. Symp. on Biomedical Imaging: Macro to Nano* p 680

Shi C, Hwang J, McMillan S, Root A and Singh V 2004 A system level resource estimation tool for FPGAs *Int. Conf. Field Programmable Logics and its Applications*

Walke R L, Dudley J and Sadler D 2000 An FPGA based digital radar receiver for soft radar *Conf. Record of the 34th Asilomar Conf. on Signals, Systems and Computers* vol 1 p 73

Weinstein R K and Lee R H 2005 Design of high performance physiologically-complex motoneuron models in FPGAs *Conf. Proc. 2nd Int. IEEE EMBS Conf. on Neural Engineering* pp 526–8

Xiaoyang Z, Chao C and Qianling Z 2004 A reconfigurable public-key cryptography coprocessor *Proc. 2004 IEEE Asia-Pacific Conf. on Advanced System Integrated Circuits* p 172

Yao H H and Swartzlander E E Jr 1993 Serial-parallel multipliers *Conf. Record of the 27th Asilomar Conf. on Signals, Systems and Computers* p 359