Universitat Politècnica de Catalunya

Escola Tècnica Superior d'Enginyeria de Telecomunicació

Departament d'Enginyeria Electronica

Master Thesis

Ubichip Virtual Machine and visualization of Spiking Neural Network Parameters

Author:
Jan Budzisz

Supervisor:
Prof. Jordi Madrenas

January 26, 2010

Abstract

Since ages people have been wondering the inner workings of brain. There is more and more information about processes involving particular brain regions, yet synthesizing neural-networks in a large scale has always been a problem. Despite the fact that the popular desktop computer is becoming more powerful, simulating massively parallel computations aways proved to be a challenge. Due to its genericness, in very specialized work, such as neural networks, it is far better to design and use dedicated array of processors.

Such approach has been used in development of the Ubichip – a device designed specifically for such purposes. The design goal was to simplify the execution elements to suit the needs of efficient neuron model algorithms emulation.

As with every development cycle of a complex tool there are many tasks that may be carried out in parallel. This, of course, effects in allowing the development team to shorten the cycle and provide the solution faster. This however, requires a set of tools that will allow to prototype and verify work progress against some set of basic rules.

The Ubichip was already supported by a toolkit named SpiNDeK, that allowed to create networks and with the use of ModelSim simulate the code. This solution for proof-checking the algorithms however proved to be cumbersome and due to many levels of indirection – slow. To increase efficiency when working with code, the programmer should not have to wait endlessly watching the progress bars. To remedy this, improve efficiency and encourage more precise tuning and development of new neuron-model algorithms the Ubichip Virtual Machine was born.

At first it was just meant to be a simple visualization tool, but as it turned out there was a missing link in the chain of tools available for the Ubichip, which had to be filled.

Thus, in this work a virtual machine for the Ubichip has been developed, as well as a visualization tool that enables a convenient display of the evolution of spiking neurons in a network.

Acknowledgements

First of all I would like to sincerely thank my supervisor, prof. Jordi Madrenas for his support of the project, guidance and discussions about possible improvements and new features of the Virtual Machine, the support throughout entire semester and for providing useful tips during my stay at the UPC.

I am also grateful to Giovanny Sánchez Rivera, for testing the VM as well as providing few useful pointers as my development went on.

This great opportunity would not be possible without the support from dr Małgorzata Napieralska at Technical University of Łódź, which I am very thankful for answering many questions and guiding me through the process of dealing with all paperwork for the Erasmus program.

Finally, I owe a debt of gratitude to my beloved girlfriend and my parents for providing me with support and encouragement over my whole studies.

Glossary

AER
Address Event Representation

ALU
Arithmetic Logic Unit

CAM
Content Addressable Memory

CIL
Common Intermediate Language

CPU
Central Processing Unit

GC
Garbage Collector

GUI
Graphical User Interface

HTML
HyperText Markup Language

JIT
Just-in-Time compiler

LSB
Least Significant Bit

MSB
Most Significant Bit

MSIL
Microsoft Intermediate Language

OS
Operating System

PC
Program Counter

PE
: Processing Element

RPN
: Reverse Polish Notation

SIMD
: Single Instruction Multiple Data

SYA
: Shunting-yard algorithm

UI
: User Interface

VM
: Virtual Machine

XML
: eXtensible Meta Language

Contents

1 Introduction

Rapidly changing world around requires us to constantly adapt and solve increasingly demanding tasks. Methods of labor and design have changed many times during the last decade, only to create new challenges. In late 1980's computers began to play more important role, as Computer-aided Design software emerged and allowed us to carry out more and more complex tasks, by managing growing number aspects of design by itself, allowing designers to concentrate on function and key feature optimizations. Many simulation frameworks came along with CAD software. At first they were relatively very limited due to hardware constrains. Yet with the development of modern CPUs the power to simulate became very cheap. But, as stated in the beginning, with new knowledge new questions arose, which required other approach – an approach that is no longer one domain specific, is not strictly tailored towards certain problem solving, and is flexible and self-adaptable.

The nature itself seems to be the best answer. Many patterns found in the world surrounding us were developed over million years, taking it from simple yet brilliant ideas into almost pure perfection. Evolution observed among all living beings allows to optimize key aspects of life to preserve fitness and self-adapt to changes in the surrounding world. One such brilliant idea of Mother Nature is the brain which essentially has simple principles of operation, but put together its size and complexity and we have the most capable problem solving tool known to man. Its brilliance is the inherent ability to adapt, tolerate faults, precisely interpret incoming stimulus and to match or even guess patterns.

First attempts to create a working model of a neural network were done by Rosenblatt in 1957[11]. These was a primitive electro-mechanical device that was meant to recognize symbols, but as it turned out it was not working for more complex symbols and was also sensitive to position of it and the size on the viewing field. Next successiful concept was proposed by Bernard Widrow in 1960. It was a network built from many electro-chemical elements called Adaline (Adaptive linear element)[1].

No wonder that with growing availability of tools the drive to simulate and understand better neural networks was propelled. Such simulations, however,

seemed and still are a very demanding task. The previously mentioned computers, despite their increasing computation capabilities, still did not provide sufficient processing power to simulate large scale neural networks in real time. The key word here is "simulation" – which means that there are levels of indirection. This effectively means that instead of harnessing full available CPU power, we must sacrifice some of it to do some management and translation of hardware features. The more complex and more flexible the simulation software is, the more levels of indirection it may have leading to inferior performance.

If the simulation of neural network is such an ineffective task on generic CPUs why not create a tailored design suited to resolve the problem of performance?

The first effort to create a solution for building more sophisticated networks at UPC and several other European universities, was project POETIC. It was based around a notion of building chip containing directly mapped neurons. However building larger networks would require enormous amounts of chips and managing its interconnections would be a error prone and mundane task. The concepts, experience and the conclusions helped other projects.

1.1 Objectives

The initial objective of my project was to deliver a tool for visualize some aspects of Spiking Neural Networks simulated by the Ubichip. Yet with the ongoing development, the focus has shifted from this visualization tool to a high level of abstraction software implementation of Ubichip that is coupled with GUI (Graphical User Interface) capable of presenting graphically its results. This software implementation, referred further as Virtual Machine, became the main concept around which this work is based.

2 Spiking Neural Networks

For years scientists have studied human brains. First through extracting tissue from animals and analyzing samples under a microscope, then by isolating single fibers and applying small currents to it, finally trying to sample currents on a living animal and also applying small currents to stimulate different brain areas. The ongoing research on human brain allows us to deeper understand its inner workings and how to create better models.

Use of modern computers capable of simulation of a discrete set of neuron and synapse properties allowed to gain further understanding of their interaction. Biological discoveries, careful study and observation led to derivation of mathematical models – starting from simplified versions – created to allow fast and simple computations of larger neural networks, to more elaborate description – used in smaller networks, but geared towards deeper understanding of processes involved, with a whole range of models available in between those two.

Currently a handful of different neural network models exists, each focusing on different features of their functioning. The European Perplexus project concentrates on the emulation of large-scale Spiking Neural Networks, which fall into the third generation of neural network models, taking into account not only neuronal and synaptic state but also the concept of time. This approach, contrary to typical perceptron networks, means that the neurons do fire only when its membrane potential[1] reaches a specified threshold value. The signal emitted by the neuron causes the change of potentials of other connected neurons. This model was proposed in 1952 by Alan Lloyd Hodgkin and Andrew Huxley[15], and has been successively refined. The spiking neuron model used in the Perplexus project was proposed by Iglesias and Villa[6].

2.1 Classic neuron model

A neuron or nerve cell is an element from which the nervous system is built – this includes brain, spinal cord and connections to and between other tissues

[1] Membrane Potential – an intrinsic quality of the neuron related to its membrane electrical charge

like muscles. Neuron cell is responsible for transmitting as well as for processing informations. It is excitable by external electrical currents, but there are also some neurons that are self-excitable and spike regularly. The information between cells is transmitted through with involvements of electrochemical methods.

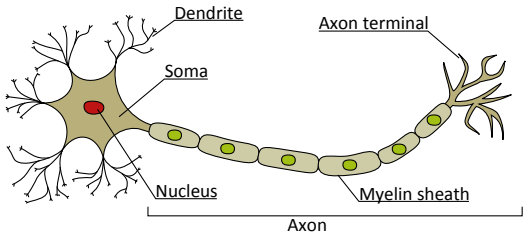


Figure 1: Overview of nerve cell

The neuron (as seen on Figure 1) may be divided into three functional parts. First one is the cell body, also called soma, which contains the nucleus that produces the most of RNA used to produce proteins. Its membrane integrates input stimuli from other neurons and fires (produces a spike), when the membrane potential rises over a threshold. Second part are the dendrites through which the neuron receives the impulses called spikes. The last part is the axon ending with axon terminals that transmit the impulse through synapses – connections with nearby cells. There are also cells that are wrapped around axon, and act as a insulator as well as greatly increase the speed of information transfer, this is called myelin sheath produced by Schwann cells.

A single neuron transmits impulses from dendrites to axons. With some careful observations and experiments a simple mathematical model can be proposed which acts as a base for our further study.

$$s_j = \Phi\left(\sum_i w_{ji} s_i\right) \tag{1}$$

This is a basic equation for evaluation of simple artificial neurons output or, more precisely, its post-synaptic spike (s_j), based on inputs – pre-synaptic spikes (s_i). The w_{ji} provides a synaptic weights table and Φ is an activation function.

Neural networks are made of vast numbers of interconnected neurons. There are of course no synaptic weights and activation function which are just abstracts of

more complex processes involved in the workings of a neuron and its interaction. The model above was used and extended by scientists until more sophisticated kinetic models were developed, such as Hodgkin-Huxley model, but it still is useful for basic understanding of neuron workings.

2.2 Bioinspired spiking neuron model

The neuron model used in the Perplexus project is more complicated that the one shown in the previous section. The basic model uses only current states of input and has no memory effect. This effectively disables it from achieving more accurate results and with availability of more advanced models should be avoided unless used for a specific purpose.

The model used in the Perplexus project was proposed by Javier Iglesias and Alessandro E.P. Villa[6] and uses more complex equations for both postsynaptic spike and synaptic weight. All neurons are simulated using leaky-integrate-and-fire model.

Each time step the membrane potential is calculated using following equation:

$$\begin{aligned} V_i(t+1) = V_{rest[q]} + B_i(t) \\ + (1 - S_i(t)) \times \left(\left(V_i(t) - V_{rest[q]}\right) k_{mem[q]}\right) \\ + \sum_j w_{ji}(t) \end{aligned} \quad (2)$$

where $V_{rest[q]}$ is the resting potential value, B_i is the background activity, S_i is the state of the unit, $k_{mem[q]} = e^{\frac{-1}{\tau_{mem[q]}}}$ is the leakage constant, w_{ji} are post-synaptic potentials (see equation 4).

The units state S_i, as used in the Perplexus project, is a function of membrane potential V_i and is defined as:

$$S_i(t) = \mathcal{H}\left(V_i(t) - \Theta_{[q]_i}\right), \quad \text{where} \quad \mathcal{H}(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \quad (3)$$

The post-synaptic potential w_{ji} is a function defined as:

$$w_{ji}(t+1) = S_j(t)A_{ji}(t)P_{[q_j,q_i]} \tag{4}$$

where S_j is state of pre-synaptic unit, $P_{[q_j,q_i]}$ defines type of synapse and A_{ji} is its activation level.

The activation level is defined by following expression:

$$A_{ji} = \begin{cases} 0, & (A_{ji} = 1) \wedge (L_{ji} < L_{min}) \\ 1, & (A_{ji} = 0) \wedge (L_{ji} < L_{max}) \vee (A_{ji} = 2) \wedge (L_{ji} < L_{min}) \\ 2, & (A_{ji} = 1) \wedge (L_{ji} < L_{max}) \vee (A_{ji} = 3) \wedge (L_{ji} < L_{min}) \\ 4, & (A_{ji} = 2) \wedge (L_{ji} < L_{max}) \\ A_{ji}, & (L_{ji} \geq L_{max}) \wedge (L_{ji} \leq L_{min}) \end{cases} \tag{5}$$

where L_{max} and L_{min} are user-defined boundaries and L_{ji} is real-valued variable used to implement plasticity rule. This variable is defined as

$$L_{ji}(t+1) = L_{ji}(t) \times k_{act[q_j,q_i]} + (S_i(t)M_j(t)) - (S_j(t)M_i(t)), \quad \text{where} \quad k_{act[q]} = e^{\frac{-1}{\tau_{act[q]}}} \tag{6}$$

where M_j and M_i are memories of latest spike interval expressed by following equations:

$$M_j(t+1) = S_j(t)M_{max[q_j]} + (1 - S_j(t))\, M_j(t)k_{syn[q_j]} \tag{7}$$

$$M_i(t+1) = S_i(t)M_{max[q_i]} + (1 - S_i(t))\, M_i(t)k_{syn[q_i]} \tag{8}$$

The P defines the so-called type of synapse that is a potential value expressed in mV that causes depolarization or hyperpolarization. Background activity is simulating noise that is generated by firing of nearby neurons as well as other phenomenon.

This complex algorithm has been already implemented (see Appendix E) and its results are shown in Section 7. Because of the programmability of both the hardware implementation of Ubichip and the VM the current algorithm can be further extended or replaced by another one completely.

3 Ubichip

An integrated circuit targeted to support the real-time emulation of large scale spiking neural networks was developed with the Perplexus project as a EU-funded project in collaboration with other European universities. The so-called Ubichip supports in its current implementation the emulation of 100 neurons and 300 synapses per neuron. The architecture allows emulation of a network consisting from at least 10,000 neurons. Each of the participating universities contribute different experience to the project. The Université Joseph Fourier Grenoble (UJF), France, provide mathematical model of the biologically-inspired spiking neurons. At Universitat Politècnica de Catalunya (UPC), Spain, the Advanced Hardware Architectures Group develops the Ubichip architecture – the custom-designed processor for parallel computations. Closely to UPCs work, Politechnika Łódzka (TUL), Poland, verifies and synthesizes the hardware Ubichip models from VHDL descriptions.

The main focus has been placed on building a scalable platform of wirelessly connected units, to execute and observe inner workings of neural network. Unlike computer-based projects, utilizing generic CPUs like x86 or ARM architecture, Perplexus focuses on the use of custom built hardware tailored for execution of vast amount of parallel operations.

3.1 Hardware

Better execution speed was achieved by deferring parallel computations to a large yet simple array of execution units which is governed by a Sequencer unit both dispatching instructions to Processing Elements (PE) and executing control instructions. Such separation of tasks allowed to simplify the required hardware for each PE while allowing certain degree of flexibility and superior performance to more generic solutions.

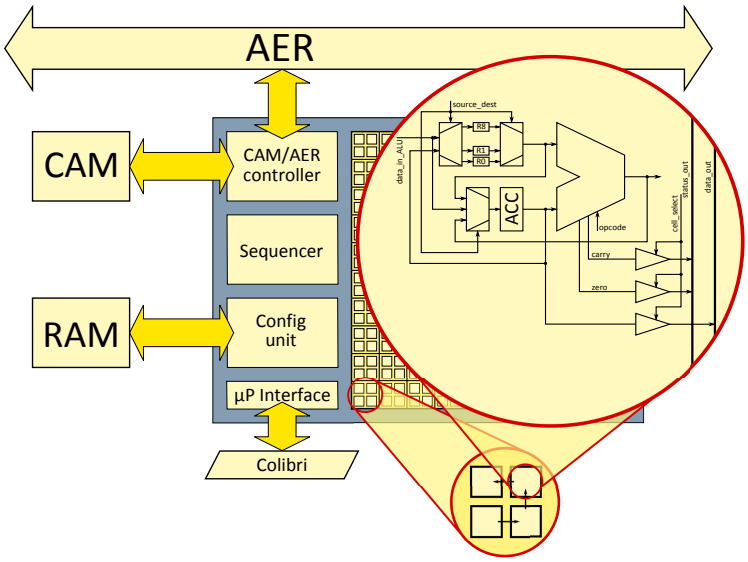


Figure 2: Detailed Ubichip architecture (multiprocessor mode)

3.1.1 PE array

The Ubichip processes data in digital domain with neuron-level parallelism. Contrary to generic CPUs which are able to process data in series only, or given array of CPUs or multi-core processor, computations could be carried out in parallel, but it would be done with high overhead cost for thread synchronization, the Ubichip is a simple yet efficient SIMD processor tailored for such operation. Furthermore, as it is fully programmable, it may be used with different algorithms.

On Figure 2 the Ubichip architecture is shown in detail. Each PE is made of four macrocells, 4 bits each, making PE a 16-bit unit. Each macrocell architecture is shown on Figure 3.

Each macrocell is capable of 4-bit computations. While it may seem to be a low value compared to nowadays 64-bit CPUs in desktop computers or 128-bit ones in gaming consoles or GPUs, its strength lies in the capability to group together macrocells, creating effectively PEs of desirable bit size. What is also important is that high precision is not required in bioinspired neural networks. Grouping macrocells together is done by configuring border routing, so that e.g. carry and zero bits are being propagated accordingly. It can be adapted to the precision requirements for the algorithm to be executed.

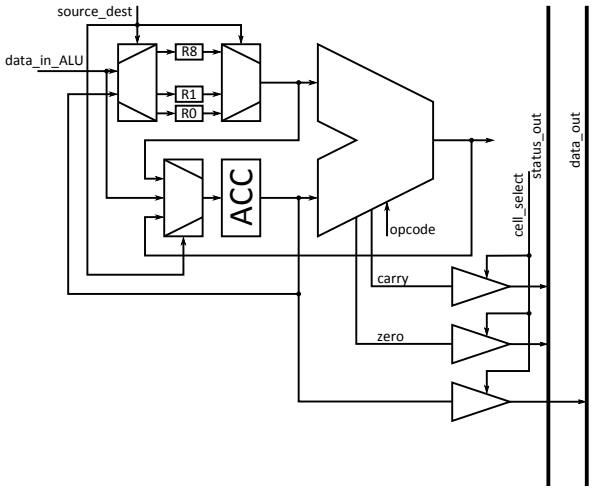


Figure 3: Macrocell logical overview

3.1.2 Sequencer

The governor for all PEs is the Sequencer unit which executes control and management instructions while dispatching all arithmetics to the array. This way the whole processor works on a SIMD scheme (Single-Instruction Multiple-Data). Such approach separates more complex control and memory access instructions and encapsulates them in one unit. There are, of course, drawbacks of this concept – e.g. there are no conditional instruction like the ones used in most assembler codes or higher level programming languages – instead a notion of `FREEZE/UNFREEZE` was introduced. It offers certain benefits over branching instructions, like e.g. a steady number of cycles per execution phase, which results in simplification of cross-chip synchronization.

3.2 Working cycle

The code execution is divided into two stages – execution phase, referred further as phase 1, and CAM controller mode phase, referred as phase 2. During the first phase, as the name implies, the algorithm code is executed, until `STOP` or `HALT` opcode is encountered. Because the second opcode effectively stalls all further processing we will consider only `STOP` to be the way to enter into phase 2.

Second stage is meant for exchanging spikes between neurons and synapses. This is done by a simple bus called AER (Address-Event Representation). During this phase CAM controller scans through all PEs (neurons) for occurring spikes, to be broadcasted by means of the AER bus. To identify incoming spikes clearly the information encoding each spike contains unique chip id, as all Ubichips have a unique id number assigned during setup, along with x and y positions. Then each spike on the AER bus is compared against CAM memory of each Ubichip and if pattern is matched, the spike is stored in the right place in the SRAM.

When all PEs have been scanned and spikes transfered, Ubichips enter phase 1 and the execution is resumed. This closes the cycle.

4 The Virtual Machine

4.1 The main idea

4.1.1 The Real World

The notion of virtualizing is widely used nowadays. From small and simple scripting languages like Ruby[5] using specialized virtual execution units, through more advanced enviroments like older versions of Java[13] (the newer ones use JIT (Just-In-Time) compilation), to full fledged virtualization software like VMWare ESX[14]. All of these are built for different purposes, but they allow easy creation of hardware-independent feature-rich execution enviroments. This simplifies both development and deployment of software. Tools like VMWare Workstation or Sun VirtualBox[12] allow to use server hardware more efficiently, thus effectively giving opportunities for reducing costs. It is achieved by paralellizing multiple tasks and OSes (Operating Systems) on single hardware.

These practices are very often used in modern web infrastructures – because a web server only responds to request sent by users, most of the time it is idle. Using one machine solely for this purpose would incur much waste of resources. This is where virtualization technologies start to become useful. They allow to setup multiple "guest" OSes, so they do not interfere with each other providing high security through separation, on a single hardware, thus utilizing it more efficiently. More advanced solutions allow managing larger infrastructures of hardware hosts, even with seamless switching virtual machines between hosts to distribute the load.

4.1.2 SpiNDeK tool

With development of Ubichip hardware, a tool to automate building of binary code and all other necessary files like contents of CAM memory were required. A toolkit called SpiNDeK was developed as a part of Master Thesis of Michael Hauptvogel[4]. It is created and tailored towards use with a specific simulation

algorithm developed along with it. With a relatively simple GUI (Graphical User Interface) it allows range of tasks to be carried out. Most important of them are:

- setting of various neuron parameters,
- generating a neural network of user-specified size and interconnections using models for connection-type and length distribution,
- generating data segment for RAM memory contents,
- joining generated data section with existing algorithm,
- assembling code and storing result in a VHDL array for simulation,
- invoking ModelSim to carry out simulation and parse the results into a HTML table.

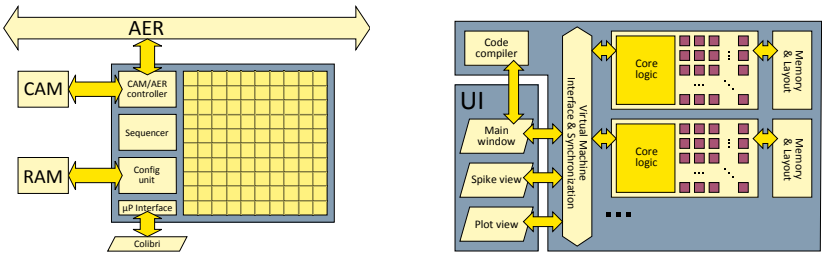
4.1.3 Research

The SpiNDeK tool has been used to create neural networks and the digital VHDL simulation is being carried by ModelSim. The simulation however is slow, because for every change of parameters the whole model must be recompiled. When this process is over, the simulation itself takes significant amount of time, as the ModelSim simulates not only the actual code but also the FPGA underneath. To add even more complexity the output is a simple text table with a lot of data. This of course is not very helpful and is hard to read by us, humans. There is too much information at any given moment. It would be much easier to be able to graphically express these values. This is where the idea of building a VM for the Ubichip came into being.

At first, as mentioned above, the sole effort was put into increasing of code execution speed, thus allowing faster viewing of the results and developing the visualization software. However as this concept expanded it became clear that it may deliver substantial benefits for code developers allowing them to instantaneously monitor all available parameters in an easy manner. Instead of searching through one big result table the VM is able to display the variables in one specific place, so that it may be observed separately.

4.2 Differences from the hardware implementation of Ubichip

To achieve performance required to execute Ubichip code the architecture could not be just copied and implemented exactly as in hardware version. Such approach would incur too much overhead as well as cause problems with maintainability. To overcome this, the chip design was divided into its logical elements and only these were implemented. To see the differences it is best to show them as a graphical representation.



(a) Ubichip (b) Virtual Machine

Figure 4: Comparison of Ubichip and Virtual Machine architecture

Most visible difference is that the architecture presented on 4 is open and is extended just by adding other elements, which are then joined by AER bus. On the other hand the VM is self-contained and may not be extendable in a way the Ubichip does. It may be possible to integrate with other system by either extending `VM` class to act as an extension to other AER buses or subclassing `Ubichip` class, when the computer should act as a master for synchronization and controlling external units. Both methods provide different benefits, but due to object-oriented approach they should be easily implementable with the use of existing code. More detailed information on the extension possibilites may be found in section 8.2.

There are also differences that are not visible in the provided picture. One of the most significant distinction is that the Ubichip processing macrocells are composed of four smaller parts called ubicells, which are the actual processing elements of the array. Because the VM is aimed at (relatively to previous simulation method) fast throughput, a different approach had to be taken. The single instance of virtual chip contains only the actual processing elements.

There is an ongoing architecture development to enable Ubichip to be reconfigurable, so the macrocells can be merged into processing elements of varying bit-sizes (though they will be multiples of four). As mentioned earlier this is not directly possible because of different architecture. In order to achieve similar functionality in the VM a feature had been proposed by the promoter to allow setting configurable register size, which effectively is the bit-size of the processing element. However due to the fact that internal processing is done on the 32-bit variables, this value is the limiting factor while configuring VM. For more details refer to Section 4.5 on page 16.

Another difference is that the responsibility for data exchange between chips belongs to the main class which also governs the work of every single virtual chip, whereas AER bus is responsible solely for data exchange. Such approach allows to easily simulate multiple chips and track the results of their interactions.

4.3 Assembly language

Because of the unorthodox instruction set, which does not directly support conditional branching, the most effective way to program the Ubichip is to use the low-level assembly. This compared to most high-level programming languages seems to be a hard and mundane task, but in the end the code fragments generated by hand usually execute faster or at least at the same speed as code generated by a compiler. And since the whole idea of Ubichip is geared towards just the simulation of neurons the actual code that needs to be written is only the algorithm, which usually is not very long. The complete instruction set available can be found in Appendix A.

The assembler file consists of three parts:

- definitions,
- data-segment and
- code.

The first section is used to define values of certain labels that can be later used multiple times when providing parameter for assembly command.

Next section being with a `.data` label and provides a way to define data needed for the algorithm. Separate data fields are stored in a key/value pair. The key can be used later in the code section.

In the last section the actual code is located, hence it begins with `.code` label. It may be divided into different code blocks by placing code-labels beginning with a dot. These labels can be then used with `GOTO` and `GOTOF` commands. Each individual opcode may only take one parameter, but both VMs and SpiNDeKs compiler support extended syntax (see Appendix B). For examples of assembly code see Appendix E.

4.4 Assembler

To allow the user to view the results more rapidly after each code change the VM is reading source code from a text file, which then is assembled into bytecode. The bytecode is then being embedded in the same way as it would be expected to be by the Ubichip:

1. Instruction Pointer
2. Data pointers
3. Code
4. Data.

The class responsible for the assembly process also reads additional segments of code – the defines from which the main application gets the values for size of the array and number of synapses. Its use is pretty straightforward and comes down to supplying the source file to the constructor of the class. The result can be then fetched from the `Bytecode` property of the object instance as array of bytes.

Besides the native assembly code defined in Appendix A, the code generator supports all command shorthands listed in Appendix B.

The assembly process is split into three phases. During the first phase the source file is tokenized. The definition section is read and a dictionary based on defined symbols is created. The data is also read and stored internally along with its labels. Finally, each line with command is split into command name and its list of parameters. The complex commands are then split into sub-commands.

After that, in the second phase, the memory layout is created – placeholders for pointers are created, and code is being translated into actual opcodes. Parallel to this process a list of label addresses is being built and every appearance of `GOTO` or `GOTOF` command is being tracked, for later step. Then the data is appended after the code which leads to beginning of third pass.

The last step is only used to fill the placeholders for the data pointers with appropriate addresses and writing code label addresses for `GOTO` and `GOTOF` instructions.

The `Code` class also contains a helper method for retrieving commands from the memory, which is being used by the code execution methods in the `Ubichip` class.

4.5 Memory layout file

Due to the ever-changing and evolving nature of every development cycle my attempt was to create a VM that is as flexible as it was possible to be done in time I was given for this project. With such capabilities and tweaking options the configuration may prove to be much complicated – each degree of flexibility adds complexity both to the application code and the setup, making the usage learning curve slower for the user.

In a simple project aimed at performing simple tasks many elements may be hard-coded. It simplifies the development, allows rapid prototyping and often also increases application performance. Such tools however, are very limited and every change usually requires changing the code and recompiling whole project. This, of course, is inconvenient, slow and prone to errors. So as long as project is tuned to handle one specific task such approach may be sufficient. As the project grows larger and is used by multiple users with different needs a different solution must be used.

4.5.1 Necessity is the mother of invention

First version of the VM was aimed at using rigid memory layout defined by previous developers of algorithm. Soon however it became clear that work on other algorithms needed different memory layout. This is when the idea of memory layout file was introduced. At first it was primarily aimed just at providing the structure of RAM contents, but soon adopted a few configuration options which allow changing of PE registers size, pointer size and its step as well as spike routing options to allow creator to specify its parametrized source and destination.

The layout configuration is stored as a XML file – which is both easily readable by humans and plenty of frameworks support it, allows hierarchical structures and navigating though it is usually simple, yet efficient. Because the VM is written in .NET Framework, which supports XML as its native configuration format, it made implementation of configuration code an easy task.

The basic structure of the layout file is as follows:

```
<MemoryLayout>
  <Options>
    <!-- Options go here -->
  </Options>
  <Layout>
    <Item>
      <Segment>
        <Variable />
      </Segment>
    </Item>
  </Layout>
</MemoryLayout>
```

4.5.2 `Layout` section explained

The `Layout` section defining the memory structure consists of many `Item`s, each being representation of one or few pointers located at the beginning of RAM contents. The sequence in which `Item` elements appear should match exactly the sequence in which they occur in memory.

The `Segment` element describes one memory segment. Data available at each memory pointer is distributed equally among all declared segments, i.e.: basic memory layout (which is included in Appendix F for reference) consists two different declared segments for synapses, and each segment is repeated the number of neurons times.

Let's assume we have 4 neurons, each having only one synapse. If each synapse requires two segments to store all the necessary data this effectively means that we have 8 segments in total. First half of this memory is then used as a sequence of repeated variables declared in first segment (4 SP1 for different neurons) and the other half as repeated variables of the second segment (four SP2s, one for each neuron).

Each segment is divided into `Variable`s, which define the name, size and offset of the variable in each segment.

Because this may seem to be confusing at first, and a picture may be worth thousand words its best to look at the graphical explanation of this concept visible in figure 5.

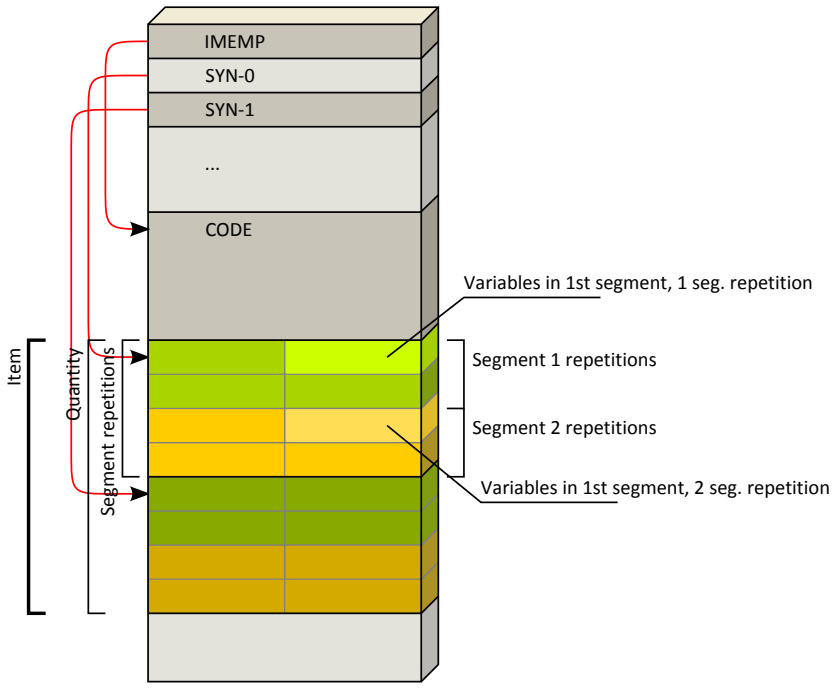


Figure 5: Memory layout explained

Each `Item` may have additional attributes:

`name`
: Specifies the name of the element.

`size`
: Specifies size of one segment in bits. Must be a multiple of 8.

`quantity`
: Number of whole item repetitions, each repetition is using one pointer. Default: 1.

`segments`
: Sum of number of repetitions of every segment.

The last two of these attributes are expressions evaluated during setup. Available variables are: synapses, neurons.

`Variable`s attributes:

`name`
: Specifies the name of the variable. This field to be accessible for spike routing or by plot expressions should begin with a character and be followed by alphanumerics. This value has to be unique among other `Variable` elements.

`offset`
: Specifies the offset in bits from the LSB. Default: 0.

`size`
: Specifies size of the variable in bits. May not be greater than 32 bits.

4.5.3 `Options` section explained

In the `Options` section one can define using elements:

`RegisterSize`
: Width of PE's register. Range: 1–32.

`PointerSize`
: Bit width of pointers located at the beginning of memory. Must be a multiple of 8, but not larger than 32 bits.

`PointerStep`
: Bit step occurring with unitary change of the pointer. Must be a multiple of 8.

`SpikeSource`
: See below.

`SpikeDest`
: See below.

The `SpikeSource` and `SpikeDest` share the same node layout and attributes. The `source` attribute specifies the behaviour of each spike accessors. You may use `memory`, `accumulator` or `none` as its value depending on your specific needs. If `memory` option is chosen, you have to define `Item`, `Index`, `Segment` and `Variable` elements. The value of the first and the last is treated respectively

as `Item` and `Variable` name attributes defined in the `Layout` section, while the other two define expressions that are evaluated during each access. Available variables: synapses, neurons, synapse, neuron. Using basic memory layout the spike should be transfered to a specific S_j (refer to section 2.1), so the `SpikeDest` may look similar to example below.

```
<SpikeDest>
  <Item>synapses</Item>
  <Index>synapse</Index>
  <Segment>neuron</Segment>
  <Variable>Sj</Variable>
</SpikeDest>
```

For a spike source one may want to use the PEs Accumulators LSB, or LSB of R0, first register. This can be easily achieved by using following code.

```
<SpikeSource source="accumulator" />
```

For the more adventurous, who want to test different aspects of their algorithms `none` set as source may prove useful e.g. for generating spikes. Basic setup of `none` outputs a 0 every time when read, and discards every write when set as a destination. The first of this behaviours can be altered by entering an expression which is evaluated for every neuron/synapse read and if the evaluation value is non-zero it is treated as a spike. The expression should be entered into a `Expression` node, like in the code below.

```
<SpikeSource source="none">
  <Expression>1*(neuron-2)*(synapse-1)</Expression>
</SpikeSource>
```

This feature, however, is experimental and should be currently treated as such.

For working examples see Appendix G.

4.6 Boosting execution speed

The virtualization techniques proved to be very promising for both efficiency and flexibility. The two versions of execution engines were developed as a proof-of-concept, to showcase the benefits within this technology. First and currently used engine is a bytecode parser which can execute code on a reference computer with an equivalent of about 1MHz. During the development a second engine was created to achieve even better efficiency. The bytecode parser was replaced by a translation of Ubichips native code into CIL (Common Intermediate Language)[8, 7] code, which upon execution is JIT compiled to the native code of executing machine. This version has outperformed the basic one four times yielding 4MHz virtualized processing power with use of a simple chip array (4 chips, 8x8 neurons each).

5 User interface

The UI (User Interface) plays an enormous role in most of current programs. This is because the graphical information is easily understood by us, in comparison to pure text output which is harder to interpret and draw conclusion from. Each UI is designed towards achieving of some more specific objective, but it is meant to provide means of control and provide feedback to the user.

5.1 Main window

Main window (Fig. 6) is the main working space of the application. This window integrates a lot of features and provides a way to control the VM. On the top side of the window a menu bar provides user with options to create a new VM, by opening assembly source file and providing path for CAM files.

5.1.1 Main tab

Main tab (Fig. 6) is divided into two sections – first showing current memory contents of first chip, and the second one representing neuron layout graphically, their excitement state (green when firing, red when not) and their connections with other neurons. Unfortunatley, it cannot be distinguished which of the synapses are excitatory and which are inhibitory. Clicking on the neuron adds it to or removes from the list of tracked neurons. When at least one neuron is on the list a window described in Section 5.5 is shown. When user removes last neuron from this list the additional window is closed.

5.1.2 Debug tab

Debug tab (Fig. 7) presents values of various registers showing state of Sequencer and individual Processing Elements. The number of visible columns can be changed via Options tab described in section 5.1.4. The register values are fetched after instruction execution, so results of every command is in the same line, contrary to the output of SpiNDeK, which aligns differently opcode and register data – in SpiNDeK the register data after opcode execution is shown in the line following current, or to put it differently, register data is aligned with a new opcode that is about to be executed.

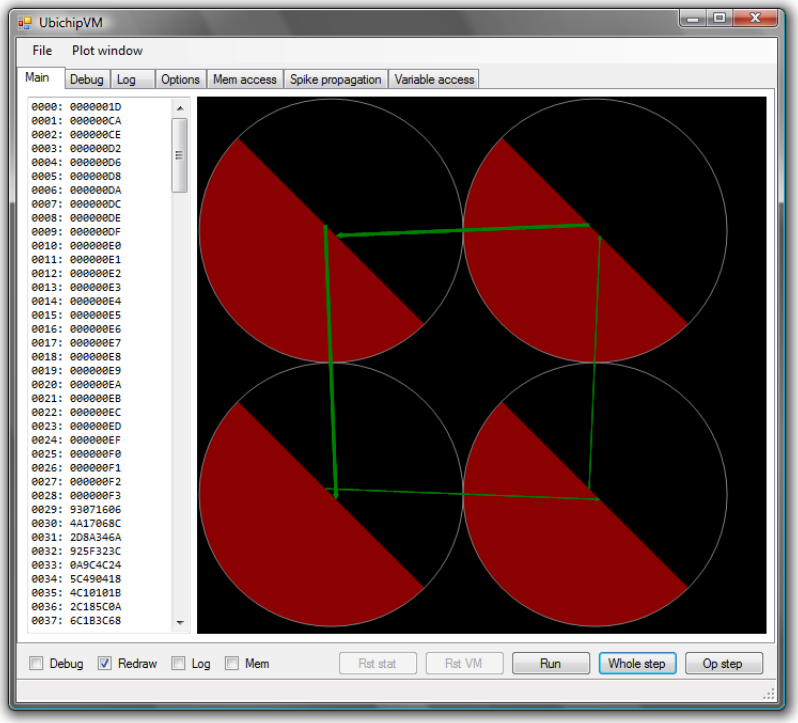


Figure 6: Main window tab

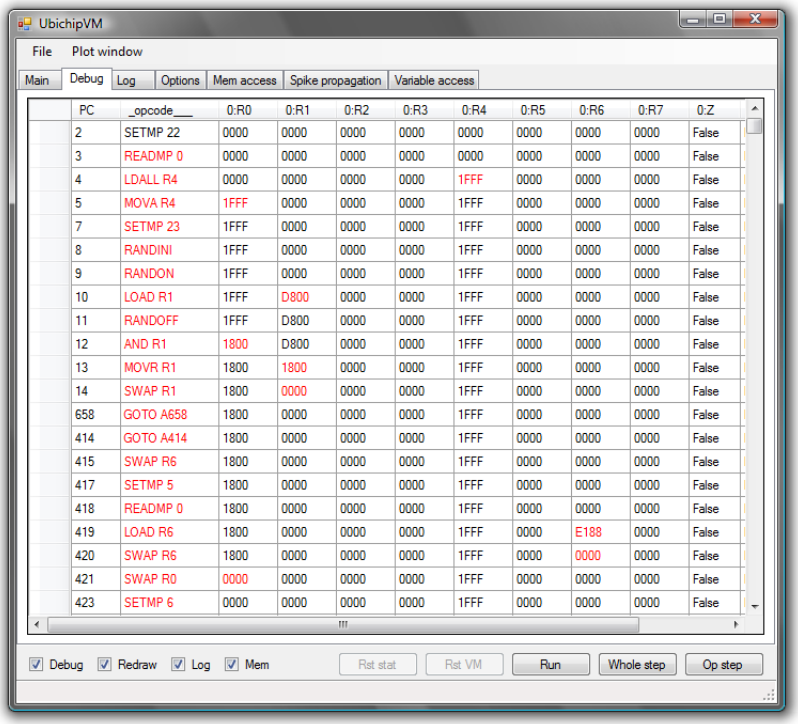


Figure 7: Debug tab

5.1.3 Log tab

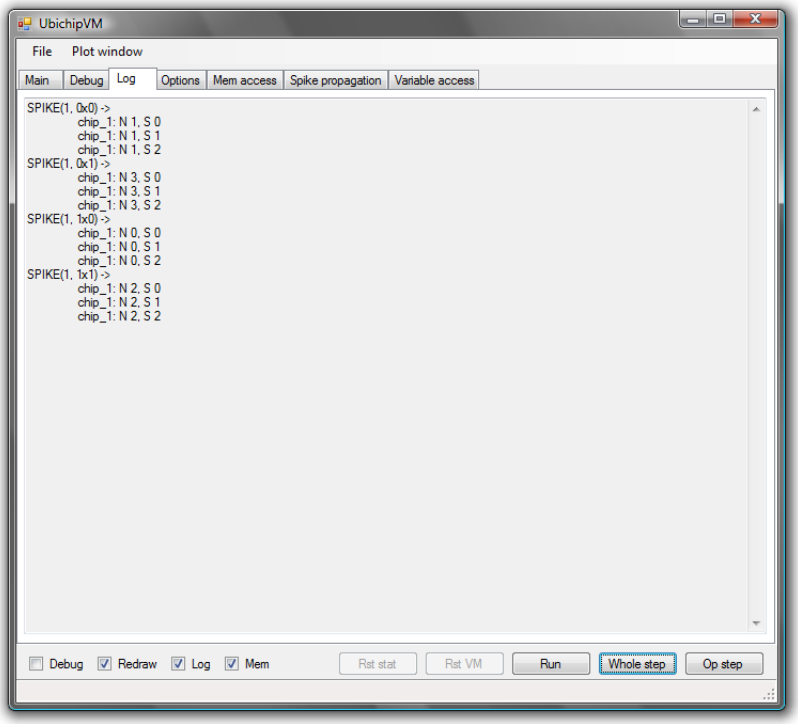


Figure 8: Log tab displaying routing of spikes

The log tab (Fig. 8) displays spike signals that are sent along the virtual AER bus and which synapses receive these signals.

5.1.4 Options

Options tab (Fig. 9) enables to choose which column user wants to display – allowing to focus on particular register.

5.1.5 Spike propagation

Spike propagation tab (Fig. 10) allows to juxtapose all presynaptic and postsynaptic spikes. The values are added at ending of each phase.

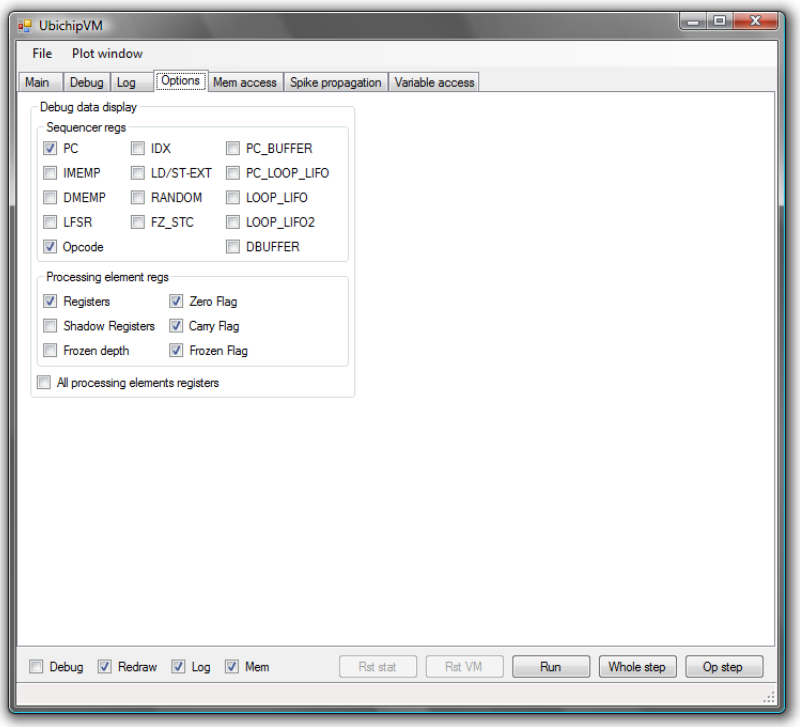


Figure 9: Options tab

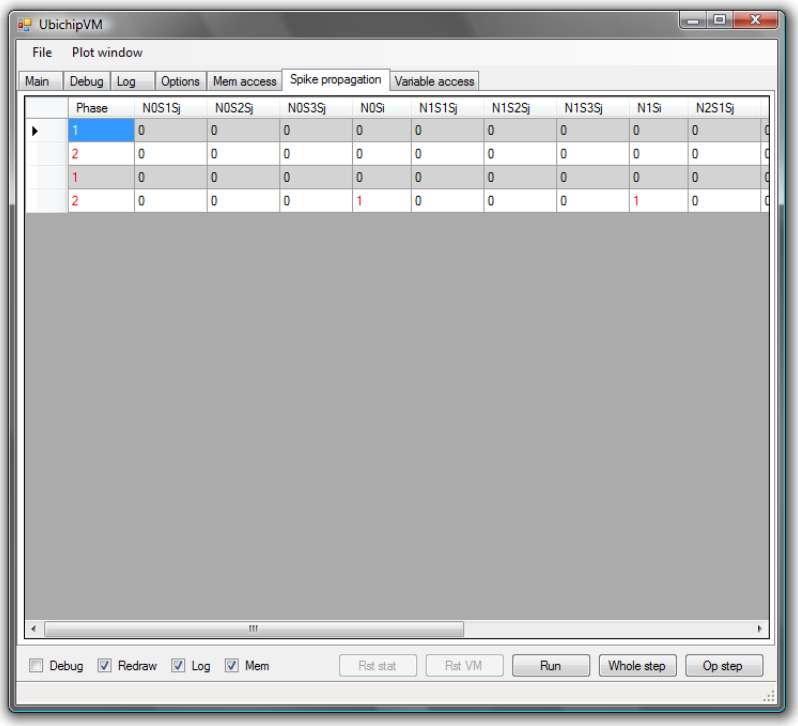


Figure 10: Spike propagation tab

5.1.6 Variable access

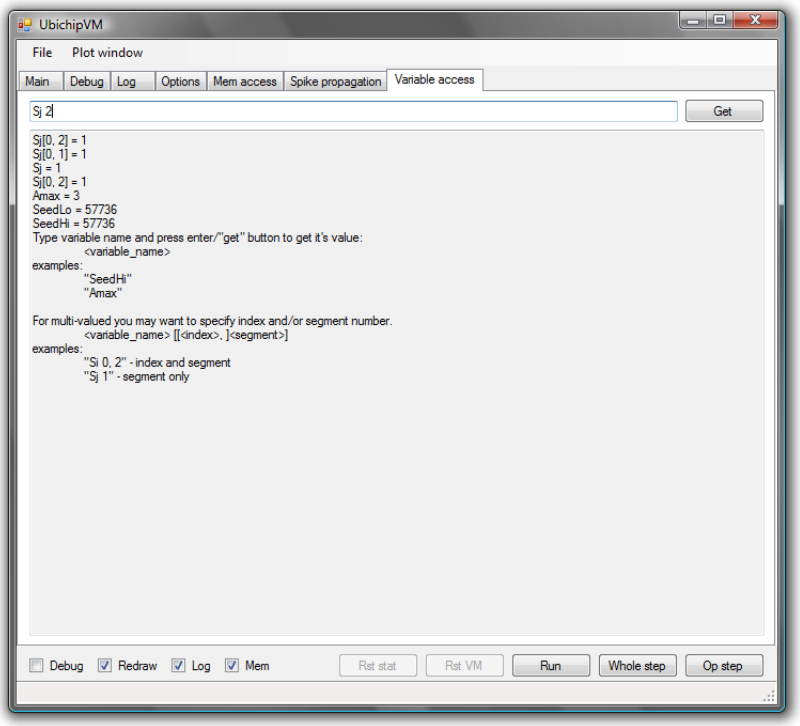

UbichipVM
File
Plot window
Main
Debug
Log
Options
Mem access
Spike propagation
Variable access
Sj 2
Get
Sj[0, 2] = 1
Sj[0, 1] = 1
Sj = 1
Sj[0, 2] = 1
Amax = 3
SeedLo = 57736
SeedHi = 57736
Type variable name and press enter/"get" button to get it's value:
<variable_name>
examples:
"SeedHi"
"Amax"
For multi-valued you may want to specify index and/or segment number.
<variable_name> [[<index>,]]
examples:
"Si 0, 2" - index and segment
"Sj 1" - segment only
Debug
Redraw
Log
Mem
Rst stat
Rst VM
Run
Whole step
Op step

Figure 11: Variable access tab

Variable access tab (Fig. 11) enables user to view values of variables defined in the Memory Layout file. User may specify additional numbers for reading variable from specific segment and index.

5.2 Open VM dialog

The dialog shown in figure (Fig. 12) is used for creating a new virtual Ubichip infrastructure. By providing the assembly file and path to folder containing `*.mif` files. The number of neurons on every chip is determined from the `define` section of code, while the number of Ubichips equals the number of `*.mif` files.

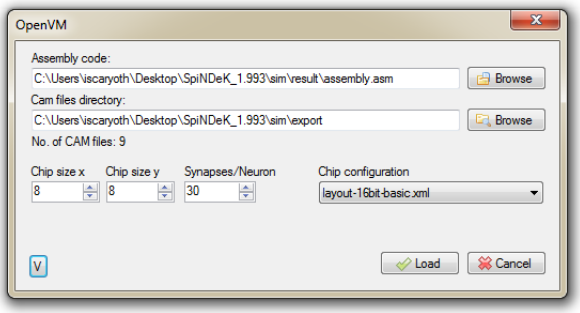


Figure 12: Open VM dialog

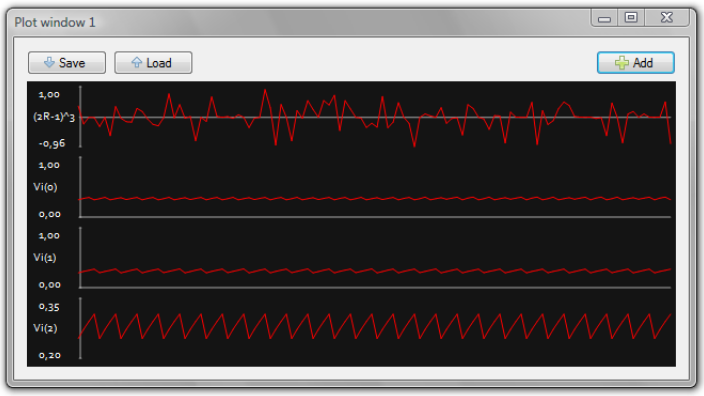


Figure 13: Plot window

5.3 Plot window

Plot window (Fig. 13) displays time plots drawn using expressions defined by the user. Because creating complex expressions is a non trivial task, the user may save specific configuration of plot expressions to a file, for later use with similar project. New plots may be added by clicking on the "Add" button or copying existing ones. The second option is achieved by right-clicking on one plot and choosing "Copy" from the menu (Fig. 14). Existing plots can be altered by either double-clicking or choosing "Edit" from context menu. Other options of the menu allow simple management of plots.

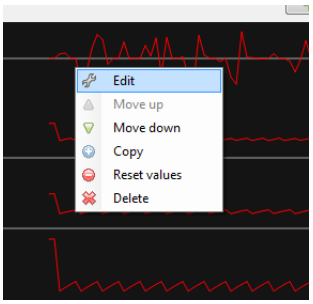


Figure 14: Plot context menu

5.4 Editing plot

Plot editing window (Fig. 15) allows definition of displayed plot name (for informative purposes only), its expression, as well minimum and maximum value (but it may be changed upon to accomodate showing of new sample). The "Digital" checkbox enables better visibility of signals that are inherently digital or are otherwise limited to a discrete set of values, and present the value in a stepped way. In the read-only textbox below all accessible parameters are displayed along with possible parameter ranges. Ranges are sets closed on both sides. The valid operations are listed in Appendix D and few examples are provided in Section 7.

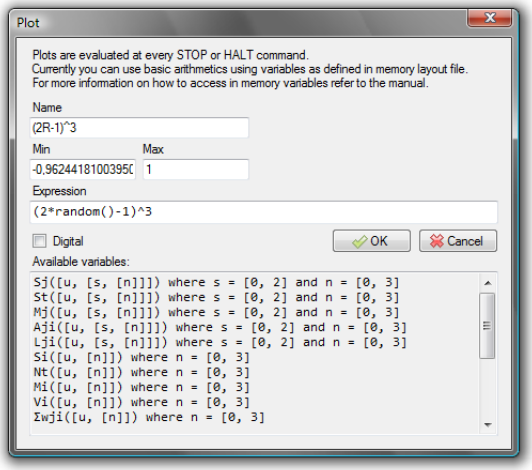


Figure 15: Plot edit window

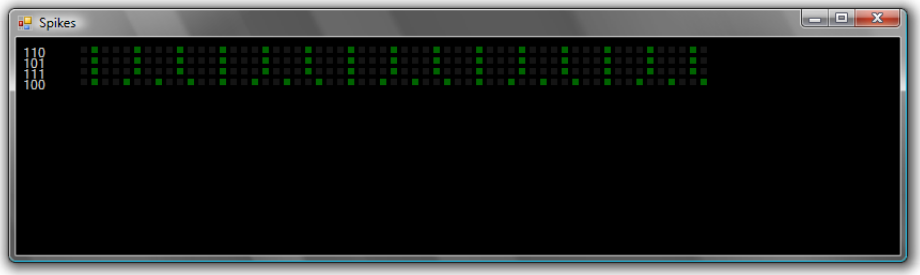


Figure 16: Spike window

5.5 Spike window

Spike window (Fig. 16) is a simple and easily readable form of displaying neurons that fired.

6 Implementation details

The main project objective was to deliver a tool that would allow a developer to monitor various artificial network parameters. Initially it was meant to fetch the data from the hardware instance of Ubichip. The VM was developed primarily as an aid in the development of the actual tool, but soon it became a larger project of its own. At first, the VM was meant to increase the speed of development process of the actual tool, but soon enough, after realizing the potential it had, it became the main objective.

Nowadays programmers can choose from magnitude of high level languages designed for fast application development and easy deployment. Also the number of additional libraries and tools provided by both community and language/framework developers is astonishing. All this is meant to increase developer productivity by providing means to allow immediate application creation with ready made components and help or even eliminate the possibility of writing insecure or faulty code. When choosing specific language many different aspects should be considered. Beside the language itself one should also take into account:

- compiler, speed of its code,
- the framework,
- available libraries,
- IDE,
- debugging capabilities and
- portability.

6.1 .NET Framework

Due to the numerous reasons I will try to explain later, I have chosen C# as my projects development language. It is object-oriented language, which is actively developed by Microsoft and was designed as a language for their .NET Framework. This framework consists of CLI (Common Language Infrastructure) which is a stack-based code execution environment and variety of compilers. The source code is compiled into a intermediate language called CIL (Common Intermediate Language), which upon first execution is JIT compiled into native code. .NET Framework makes extensive use of object references and GC (Garbage Collector).

The language is placed as a rival to Sun Java, as it shares many similarities. There are, of course, differences both in the syntax and in functionalities. The main differences between their respective execution environments are that the CLI was designed from ground up to host multiple languages, while Java Virtual Machine was initially meant for Java only and was extended recently to support other languages.

.NET is very rapidly expanding. It is being used for server-side dynamically create web pages, as standalone desktop applications, services or components, with the use of Compact Framework it may be used on smart phones or other personal devices, and recently as client-side rich web application Silverlight[9].

6.2 Decision-making process

There are many reasons that contribute to the overall decision why have I chosen this particular language. To name the most important arguments (in no particular order):

- availability of great tools – most importantly Visual Studio which is famous not only for Intellisense technology that suggest use of possible methods, properties, classes, etc., but also provides extensive debugging capabilities that are very helpful and provide deep insight on the program inner workings; VS supports also some other neat features like code refactoring;

- extensive collection of libraries, both built in the BCL and provided by users;
- GC – the CLI handles freeing of resources,
- easy and powerful syntax (delegates, anonymous functions, lambda expressions),
- JIT – good execution speed, which can be further improved by building dynamic methods suited for specific purposes,
- easy deployment – should easily run in any environment supporting .NET Framework,
- some degree of portability – with improvements being done on Mono[10] or DotGNU[2] projects.

What played a crucial role in this decision-making process was my professional work experience which allowed me to start writing this project faster because of my knowledge about both the framework and the language itself. Due to this, the learning curve of new tools and techniques was very fast and allowed me to focus on the project and its functionality.

6.3 UbichipVM Namespace

The code layout of UI is simple and makes use of VM interfacing classes. The general overview of classes and their respective connections are shown in Figure 17.

`MainDisplay` is main UI (User Interface) class. It acts as the primary user interface window. Manages VM and plot/spike windows.

`OpenVM` acts as advanced "open virtual machine" dialog. Allows user to setup code file, CAM files directory, desired memory layout and network organization.

`SpikeMemory` is a helper class for storing spikes. `MainDisplay` provides the data, while `SpikeDisplay` provides graphical output.

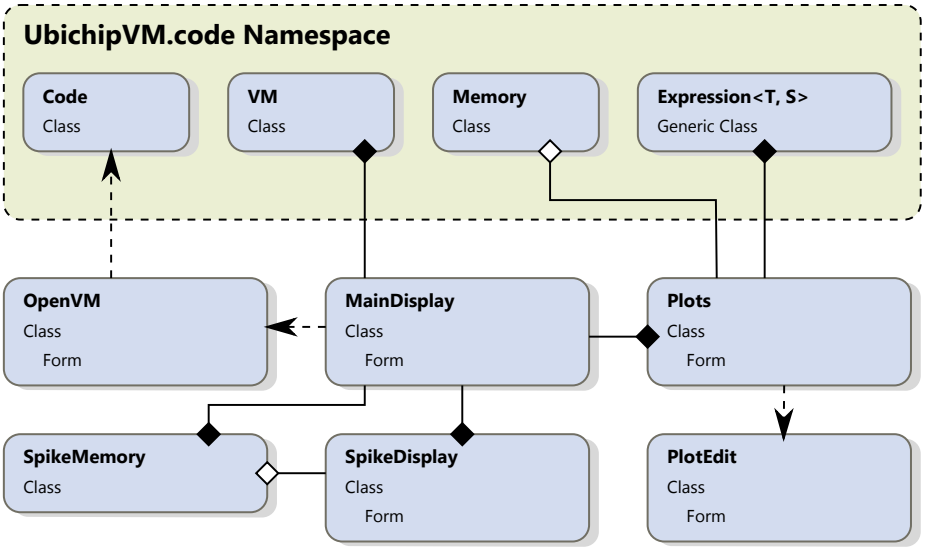


Figure 17: General class overview

`SpikeDisplay` graphically represents spikes.

`Plots` provides a window for drawing plots from used defined expressions.

`PlotEdit` handles the editing of plotting expressions.

6.4 UbichipVM.code Namespace

While UI Namespace is straightforward, the `code` namespace is fairly complex. There are a lot of associations, aggregations and compositions most of which are shown on the class diagram shown in Figure 23.

In the following sections I will try to provide an overview of this namespace most significant classes.

6.4.1 `Ubichip`

The most important class is the Ubichip class – it acts as a single instance of real-world Ubichip: handles execution of code, memory controller (for phase 1) and signaling external "devices" for synchronization. It provides also some statistics about code execution – number of instructions executed and virtual clock cycles per phase.

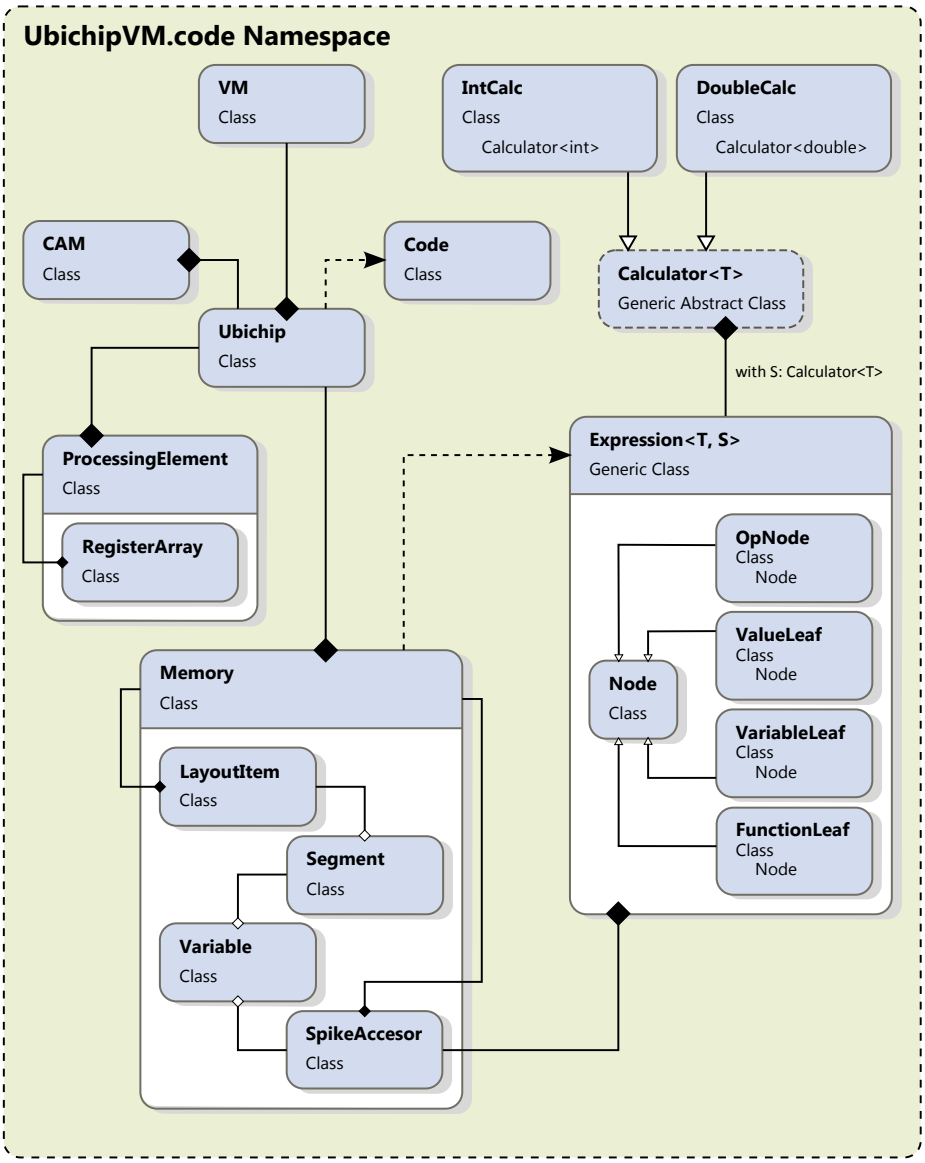


Figure 18: Virtual Machine `code` namespace class overview; for more detailed diagram see Appendix C.

`Ubichip (VM vm, Memory mem, CAM cam, int id, int width, int height, int synapses)` the constructor requires a valid reference to `VM` and `Memory` classes, because of that it is meant to be instantiated by the `VM` class. Nevertheless it may be created by hand for special purposes. `id` is the identification number that will be used when a spike is generated for broadcasting. The last three arguments define basic organization information.

`Ubichip (VM vm, Memory mem, CAM cam, int id, int width, int height, int synapses, int version)` extended version of constructor meant for choosing behaviour of certain opcodes. Possible values: 0, 1

`ProcessingElement Select (int x, int y)` method for retreiving an instance `ProcessingElement` class encapsulating state of single PE.

`ProcessingElement Select (int n)` same as above, but treats PEs layout as an array instead of a matrix.

`void Start ()` starts continuous execution. The execution takes place in a different thread.

`void Reset ()` disables execution and resets Sequencer registers.

`void Resume ()` resumes previously stopped execution.

`void Int_Ack ()` disables execution.

`void Phase2 ()` scans the PEs array and notifies associated `VM` of them.

`void Spike (int id, int x, int y)` handles storing incoming spike.

`void Abort ()` aborts currently executing thread. This is meant for use when application request quitting.

6.4.2 VM

`VM` is a simple class that manages underlying virtual Ubichip structure, synchronizes them and acts as the AER bus for spike exchange.

`VM (int nchips, int nx, int ny, int synapses, Memory mem, CAM[] cams)` constructor that besides the organization information requires instance of `Memory` class and same number of `CAM` objects for routing as specified by `nchips` parameter

`void Spike (int id, int x, int y)` used to broadcast spike signal across virtual AER bus.

`void Stop (Ubichip sender)` used by connected Ubichips to signal that they have reached a `STOP` opcode. Once all chips reach this stop, the `VM` initiates phase 2.

`void FullCycle ()` continues execution for phase 1 and then performs phase 2.

`void Step ()` in phase 1 executes one instruction, for phase 2 performs all necessary operations in single methods call.

`void Abort ()` sends Abort signal to all PEs.

6.4.3 `Memory`

The `Memory` class was developed to allow modifications to memory layout. Its structure is described in Section 4.5 on page 16. It uses an XML file as an input. This particular format was chosen because of following reasons:

- human-readable,
- hierarchical,
- availability of parsing components,
- easy to use,
- flexible.

`Memory (XmlDocument layout, byte[] contents, int nneurons, int nsynapses)` constructor that takes an XML document, SRAM contents and basic structure organization required while parsing layout file for dynamic structrues evaluation.

`List<string> VariablesList ()` returns list of available variables

`Memory Clone ()` returns cloned Memory object. Used for instancing multiple virtual Ubichips.

What is more, `Memory` class implements `this[string name]` property which returns `Variable` object, which can be used for storing or retrieving variables in memory of particular Ubichip instance.

6.4.4 `Memory.Variable`

This class provides a simple functionality for accessing memory variables. It supports both storing and retrieving values from any given segment and whole block repetition (e.g. multiple occurrences of synapse parameters block defined by default memory layout). To use it you must retrieve this object from an instance of `Memory` class and use one of following functions:

`uint Get (int index, int segment)` as the name suggest it retreives value of variable at specified index and segment,

`void Set (int index, int segment, uint value)` method for storing `value` into variable located at index and segment.

6.5 `Code`

A class responsible for compiling code and used by `Ubichip` for decoding instruction bytecodes.

`Code (string code)` constructor that takes source code as a string. After that the memory contents that are produced can be retreived through `Bytecode` property, along with define values accessible with `Defines` property.

`Code.Instruction Decode (byte [] code, uint addr)` decodes instruction at a specified address in memory. The `Code.Instruction` is a simple self-explanatory structure that contains opcode value, its parameteres and size of instruction that is used for advancing PC. This is a static method.

6.6 Challenges

During the development cycle a few essential decisions had to be made. During this project they were motivated by a desire to extend flexibility, prolonging the life of VM. Flexibility however, introduces many implementation challenges.

The primary example would be the implementation of easily accessible memory layout file. It was driven by the need to modify memory structure. Its shape was proposed entirely by me as a result of searching for best way to encapsulate current working model, as well as research of possible memory construct accessible from the Ubichip code. Also its implementation proved to be a challenge, but the outcome surpassed the expectations providing an easy way of accessing specific memory regions of a running VM and thus enabling easy construction of plots to visualize how the algorithm works.

Other important code fragments include the compiler, whose implementation was very helpful for understanding of the opcodes and basic memory structure. Although it was not a trivial task it enabled me to easily and almost instantly code the code execution section.

Designing the class layout and interaction required understanding of Ubichip interface and the way it communicates with other devices. I was able to break down the Ubidule into logical section that later were implemented as VM's classes.

7 Simulation results

Because the initial idea was to create a visualization-only tool, the plots play a vital role by visualizing the changes occurring in the neural network. Due to this I have performed a simulation to present the capabilities of this module. For this section all simulations were carried out on a network made of 9 Ubichips, each simulating an 8x8 array of neurons, each having 30 synapses.

One of basic plot is showing membrane potential (V_i) for two neurons. One in the center of the network, and another one just next to it. From the plot one can read that that in 2/3 of the time it shows there is a slight increase in maximum oscillation value for first neuron potential.

$$\text{Vi(center)} = Vi(4, 0, 32) \tag{9}$$

$$\text{Vi(center} + 1) = Vi(4, 0, 33) \tag{10}$$

To better understand what happens at this moment we should add more plots to our window. At the top of Figure 20 plot of potential value is shown with 17 more plots showing A_{ji} – activation level value for 17 synapses connected to this particular neuron. It is clearly visible that activation level of few synapses drops rapidly at the beginning reaching level 0. Shortly after the half of time shown on the plot, activation level of few more synapses rise. This increase has an effect on potential value which begins oscillating reaching higher levels. Such behaviour is expected and is showing adaptability of this network.

$$\text{Vi(center)} = Vi(4, 0, 32) \tag{11}$$

$$\text{A0} = Aji(4, 0, 32) \tag{12}$$

$$\text{A1} = Aji(4, 1, 32) \tag{13}$$

$$\ldots \tag{14}$$

In the last figure (21) a more advanced use of plot window is employed. The first plot shows sum of all incoming spike signals, next 6 plots present activation level,

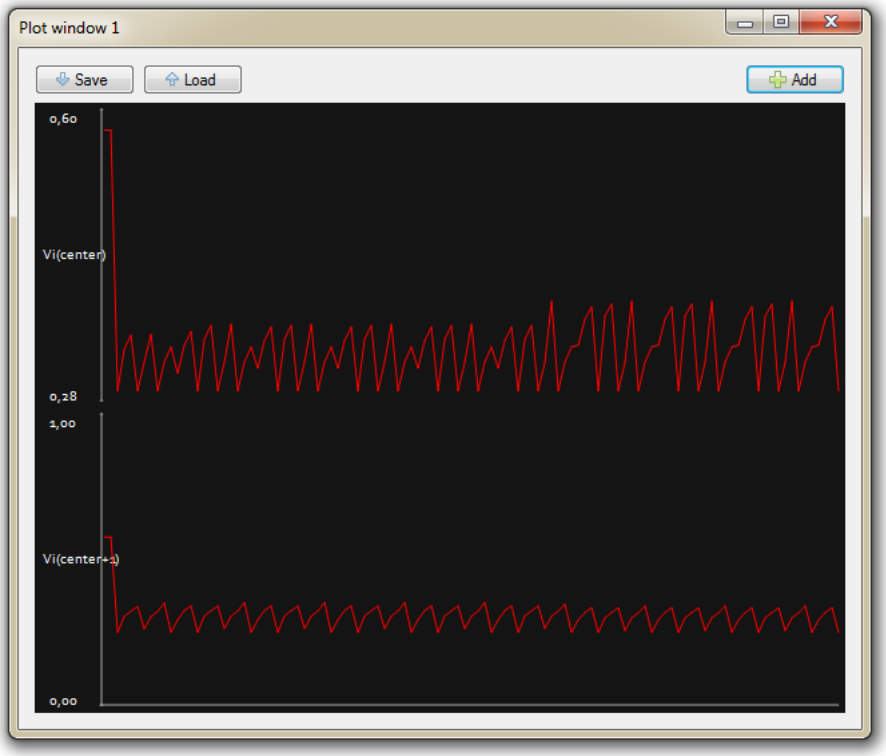


Figure 19: Visualization of potential value oscillations in a relatively large network (576 neurons, 30 synapses each)

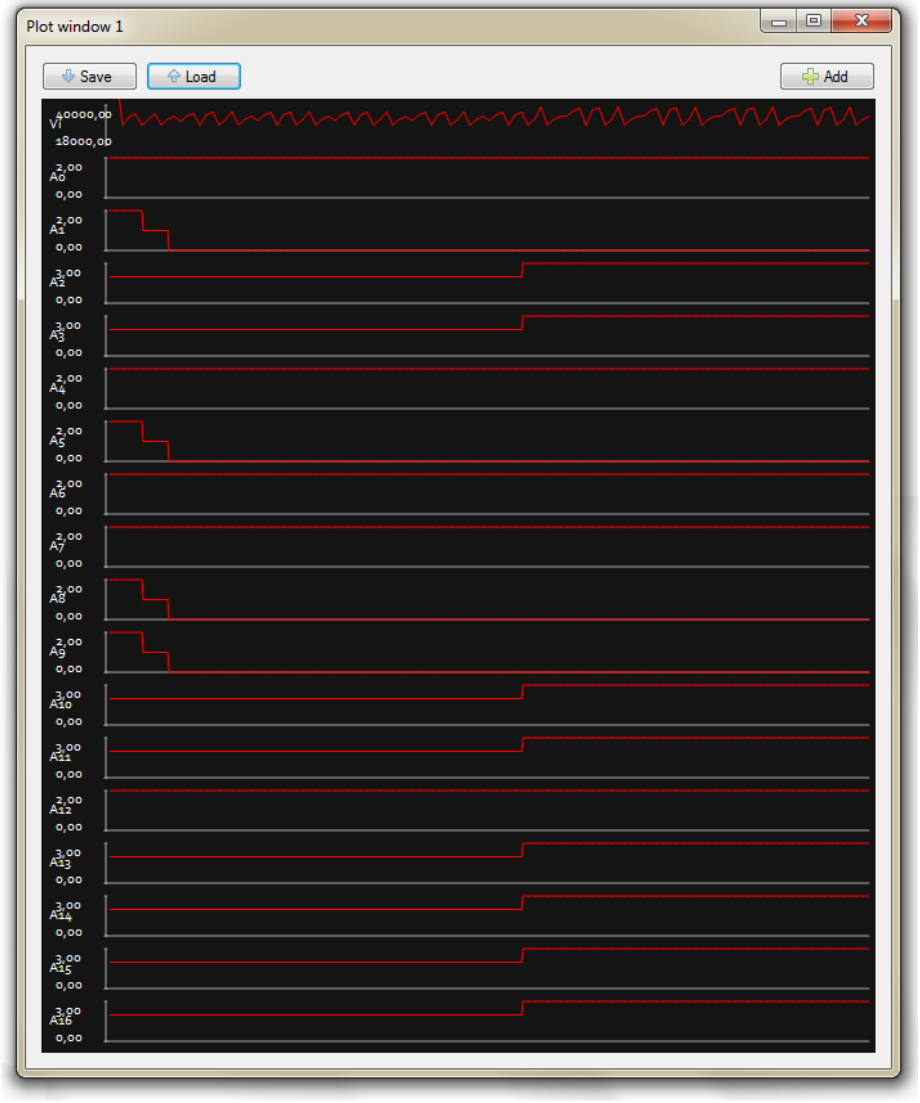


Figure 20: Potential oscillation and activation levels of few synapses

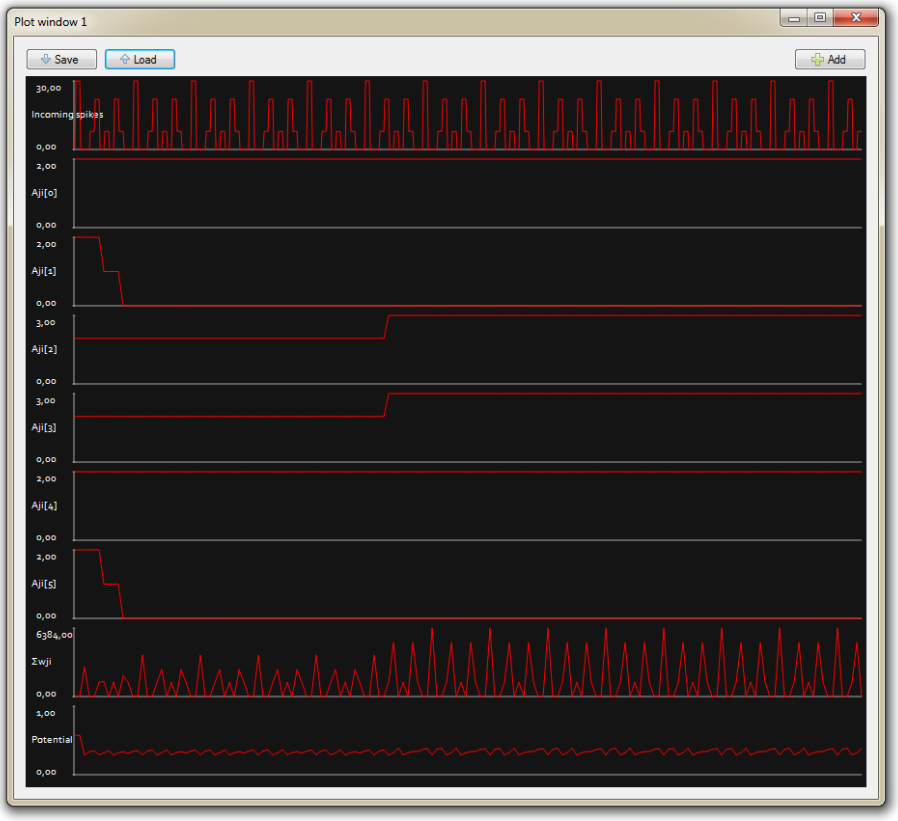


Figure 21: Plots showing number of incoming spikes, separate synaptic weights for 6 synapses, $\sum w_{ji}$ and membrane potential.

last but one shows sum of post-synaptic potentials, and membrane potential is drawn as the last plot.

$$\text{Incoming spikes} = \sum_{k=0}^{29} Sj(4,k,32), \quad \text{written explicitly as} \tag{15}$$

$$= Sj(4,0,32) + Sj(4,1,32) + \ldots + Sj(4,29,32) \tag{16}$$

$$\text{Aji[0]} = Aji(4,0,32) \tag{17}$$

$$\text{Aji[0]} = Aji(4,1,32) \tag{18}$$

$$\ldots \tag{19}$$

$$\text{Aji[5]} = Aji(4,5,32) \tag{20}$$

$$\Sigma\text{wji} = wji(4,0,32) \tag{21}$$

$$\text{Potential} = Vi(4,0,32)/10\text{\^{}}5 \tag{22}$$

8 Conclusions and future work

8.1 Conclusions

The initial concept of a visualization-only tool has been reevaluated and led to changing the main objective of the project to creating a fast and flexible Virtual Machine, on top of which the visualization was built using VMs exposed functionality. At first the aim of VM was to provide rapid prototyping environment for the visualization tool, but soon turned into a project of its own. Understanding the potential benefits of this technology helped gain focus on writing easily extendable VM that may work with user-provided memory layout, making it easily extensible and algorithm implementation-agnostic. The plotting facilities allow user to write arithmetic expressions for evaluation of algorithm code and assessment of neural network inner-workings.

The main objective was achieved with creation of the expandable code-base. The program can be used both by developers working on new algorithm types or adjusting existing ones and scientists, who are more interested in the results of simulation that can be seen on plots, rather internals of the algorithm. Addressing issues and providing functionality for such a wide array of people that has different needs and motivations is a demanding task. With careful planning, however, the difficulties can be overcome.

For the algorithm developers the benefits of using VM are mainly the easier readable debug output which aligns commands with their actual results, rather than displaying results along with following opcode, ability to select which registers should be displayed in debug output, improving readability while focusing on certain aspects.

The speed of execution is substantially improved in comparison with ModelSim simulation. This has been achieved by implementing higher level of abstraction, e.g. arithmetical operations are executed directly using CPU instead of emulating the logic gates that form Ubichips ALU. The performance is not up to par with pure-hardware implementation, yet still it provides enough throughput for

simulation purposes. However, as explained in following section, it may be improved by directly interfacing hardware and using current tool to control the process.

To sum up, I believe that this project provides a useful tool for scientist and developers alike. The ability to easily view algorithm variables, all hardware simulated registers and adapt memory layout along with PEs precision to the changing needs makes it a fairly generic and flexible tool which can be easily used for current and future advancements of neuronal simulation algorithms.

8.2 Possible future extensions

The difference between entities responsible for cross-chip data exchange create a challenge when one would like to integrate VM with external system – either another virtual machine or some external hardware like a real Ubichip or some sensory inputs.

Currently the main class `VM` is a self-contained AER bus and manager for all virtual `Ubichip`s. Both possible solutions to allow such flexibility involve modifying the `VM` class, but both require different code changes.

First approach would be to allow the `VM` to connect seamlessly with another AER bus either a real one connected to Ubichips, or another virtual one. These modifications would possibly force the VM to act as a slave to existing AER bus. It may not be the case but some of the control possibilities may be lost, on the other side allowing to connect easily with a larger system and enabling e.g. to track changes occurring in single chip, while dispatching major part of calculations into actual Ubichip infrastructure.

Second solution requires some code refactoring – extracting the interface from `Ubichip` class and making `VM` use this new interface. This would allow one to implement its own class that could connect to external chip and controlling it remoteley, then fetching the results and sending appropriate data through methods available on the `VM` class. This approach could be easily used for supplying various classes for specific needs – sensory input, external chips or output devices. However, each "device" would have to be added separately, so this solution is feasible only for some smaller, but precise experiments.

These are two basic possibilities of extending current VM to operate with external processing units. However due to the complexity such solution would introduce it was not considered as a milestone for implementation in this projects.

References

[1] *Adaline*. URL: http://www.cs.utsa.edu/~bylander/cs4793/learnsc32.pdf.

[2] *DotGNU project*. URL: http://www.gnu.org/software/dotgnu/.

[3] Marc Hortas Garcia. "Disseny i desenvolupament de software i algorisme per a xarxes neuronals spiking bioinspirades". MSc thesis. Universitat Politècnica de Catalunya, 2009.

[4] Michael Hauptvogel. "Design of a bio-inspired spiking network environment". MSc thesis. Universitat Politècnica de Catalunya, 2008.

[5] James Edward Gray II. *The Ruby VM: Episode I*. URL: http://blog.grayproductions.net/articles/the_ruby_vm_episode_i.

[6] Alessandro E.P. Villa Javier Iglesias. "Dynamics of pruning in simulated large-scale spiking neural networks". In: *BioSystems* 79 (2005), pp. 11–20.

[7] Microsoft. *CIL OpCodes*. URL: http://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes_members.aspx.

[8] Microsoft. *ILGenerator Class*. URL: http://msdn.microsoft.com/en-us/library/system.reflection.emit.ilgenerator.aspx.

[9] Microsoft. *Microsoft Silverlight*. 2009. URL: http://silverlight.net/.

[10] *Mono project*. URL: http://www.mono-project.com/Main_Page.

[11] Frank Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". In: *Psychological Review* 65.6 (1958), pp. 386–408.

[12] Sun. *Sun VirtualBox*. URL: http://www.virtualbox.org/.

[13] Frank Yellin Tim Lindholm. *The Java Virtual Machine Specification*. Prentice Hall PTR, 1999.

[14] VMWare. *VMWare ESX Server*. URL: http://www.vmware.com/products/esx/.

[15] Christopher M. Bishop Wolfgang Maass. *Pulsed Neural Networks*. MIT Press, 2001.

A Supported opcodes

Instructions used by Ubichip fall into two categories:

- data processing, executed by each PE and
- flow control, executed only by the Sequencer.

The encoding of instruction depends on its type. Most instruction that deal with registers and extended instructions are single-byte, but the SETMP, GOTO, GOTOF, LOOP and READMPR need two bytes for encoding additional parameters. Detailed encoding scheme may be seen in figure 22.

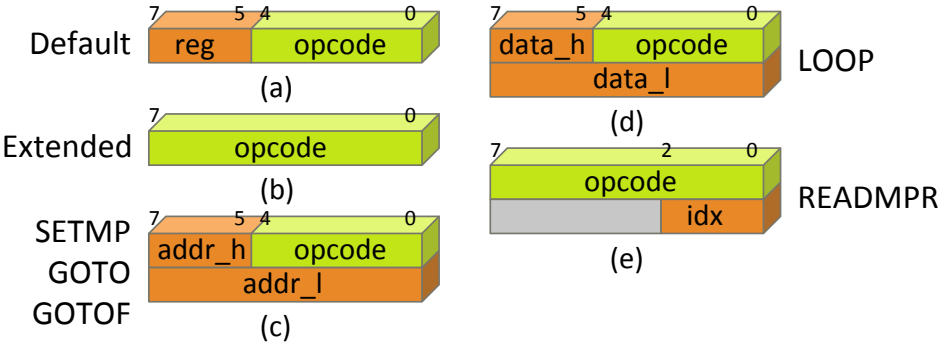


Figure 22: Instruction formats

The table below (1) shows a list of supported opcodes along with binary representation, additional notes and affected flags.

Table 1: Supported opcodes

Mnemonic	Opcode	Group	Function	Flags
EXTI	00000	–	Extended instructions	–
NOP	00000000	NOP	No operation	–

Mnemonic	**Opcode**	**Group**	**Function**	**Flags**
HALT	00100000	HALT	execution halted until Start or Step method invoked	–
RET	01000000	RET	PC ⇐ PC_BUFFER	–
SETZ	01100000	FLAG	Sets the zero flags Z ⇐ 1	Z
SETC	10000000	FLAG	Sets the carry flags C ⇐ 1	C
CLRZ	10100000	FLAG	Clears the zero flags Z ⇐ 0	Z
CLRC	11000000	FLAG	Clears the carry flag C ⇐ 0	C
TRANSFER	11100000	TRANSFER	D_BUFFER ⇐ Rx(PEa) Ry(PEb) ⇐ D_BUFFER	–
STC reg	00001	STOREC	SRAM ⇐ reg if C=1	–
STNC reg	00010	STOREC	SRAM ⇐ reg if C=0	–
STZ reg	00011	STOREC	SRAM ⇐ reg if Z=1	–
RST reg	00100	ALUOP	reg ⇐ 0; Z ⇐ 0	Z
STNZ reg	00101	STOREC	SRAM ⇐ reg if Z = 0	Z
SETMP data	00110	SETMP	DMEMP ⇐ data if data=0, DMEMP ⇐ LOOP_index	–
READMP n	00111	READMP	if n = 0 DMEMP ⇐ SRAM[DMEMP] if n >0 DMEMP ⇐ SRAM[DMEMP+ $2^{n-1} * \text{LOOP\_index}$)]	–
ADD reg	01000	ALUOP	ACC ⇐ ACC + reg	Z, C
SUB reg	01001	ALUOP	ACC ⇐ ACC − reg	Z, C
EXTI2	01010	–	Extended instructions 2	–
SHL	00001010	ALUOP	ACC ⇐ ACC << 1, carry ⇐ ACC[msb]	Z, C
ENDL	00101010	ENDL	if LOOP_LIFO[0] = LOOP_LIFO2[0] {	–

Mnemonic	Opcode	Group	Function	Flags

			pop PC_LIFO, LOOP_LIFO, LOOP_LIFO2
			} else {
			restore(PC);
			LOOP_LIFO[0] ⇐ LOOP_LIFO[0]+1
			}	

RANDINI	01001010	RANDINI	LFSR[63:32] ⇐ SRAM[DMEMP] LFSR[31:0] ⇐ SRAM[DMEMP+1]	–
RANDON	01101010	RANDOM	LFSR becomes source for LOAD and LDALL instructions	–
RANDOFF	10001010	RANDOM	LOAD and LDALL normal function is restored	–
STOP	10101010	STOP	stops code execution, invokes a VM method to start spike transfer, entering phase 2	–
TRANSFERX	11001010	TRANSFERX	D_BUFFER ⇐ Rx(PEa) Ry(MASK(PE)) ⇐ D_BUFFER	–
READMPR idx	11101010	READMPR	DMEMP ⇐ SRAM[DMEMP] INDEX_REG ⇐ PE[idx] LOAD_STOREC_EXTENDED ⇐ 1	–
EXTI3	01011	–	Extended instructions 3	–
SHR	00001011	ALUOP	ACC ⇐ ACC >> 1 carry ⇐ ACC[lsb]	Z, C
RANDON1	00101011	RANDOM	LFSR becomes source for LOAD and LDALL instructions LFSR_STEP ⇐ 1	–
FZ_STC_ON	01001011	FZ_STC	FZ_STC ⇐ 1 disables storing contents from frozen PEs to SRAM	–

Mnemonic	**Opcode**	**Group**	**Function**	**Flags**
FZ_STC_OFF	01101011	FZ_STC	FZ_STC ⇐ 0 restores default operation	–
RST_SEQ	10001011	RST_SEQ	resets sequencer	–
D2IMP	10101011	D2IMP	IMEMP ⇐ DMEMP PC ⇐ 0	–
MOVA reg	01100	ALUREG	ACC ⇐ reg	Z
AND reg	01101	ALUOP	ACC ⇐ ACC **AND** reg	Z
OR reg	01110	ALUOP	ACC ⇐ ACC **OR** reg	Z
INV reg	01111	ALUOP	ACC ⇐ **NOT** reg	Z
LOOP data	10000	LOOP	Push PC_LIFO, LOOP_LIFO, LOOP_LIFO2 LOOP_LIFO[0] ⇐ 1 LOOP_LIFO2[0] ⇐ data	Z
SET reg	10001	ALUOP	reg ⇐ reg.MaxVal	Z
SWAP reg	10010	ALUREG	reg ⇔ shadow_reg	–
LDALL reg	10011	LOAD_ALL	PE[*].reg ⇐ data_in_ALU (broadcast)	–
LOAD reg	10100	LOAD	PE[*].reg ⇐ data_in_ALU (iterative)	–
MOVTS reg	10101	ALUREG	shadow_reg ⇐ reg	–
MOVFS reg	10110	ALUREG	reg ⇐ shadow_reg	–
FREEZEC	10111	ALUREG	Increase frozen level if C=1	F
FREEZENC	11000	ALUREG	Increase frozen level if C=0	F
FREEZEZ	11001	ALUREG	Increase frozen level if Z=1	F
FREEZENZ	11010	ALUREG	Increase frozen level if Z=0	F
UNFREEZE	11011	ALUREG	Decrease frozen level	F
MOVR reg	11100	ALUREG	reg ⇐ ACC	–
GOTOF addr	11101	GOTOF	if all_frozen = 1, PC ⇐ addr; PC_BUFFER ⇐ PC	–
XOR reg	11110	ALUOP	ACC ⇐ ACC **XOR** reg	Z

Mnemonic	Opcode	Group	Function	Flags
GOTO addr	11111	GOTO	PC ⇐ addr; PC_BUFFER ⇐ PC	–

The SET instruction is dependent from the register length set in layout file. For given width of register all of its bits are set. E.g. for 16-bit organization this would result if setting the value of register to 0xFFFF, thus it is marked as registers maximal value.

The frozen level is a value that holds how many successive 'freezing' instructions with their condition met were executed, this allows multiple levels of nesting. On current implementation maximum level is set to 256.

ACC, the accumulator is just an other name for R0.

EXTI, EXTI2 and EXTI3 are just classes of commands that use full byte for opcode encoding. These have been included to show how the opcode-space has been partitioned and are followed by commands from its range.

B Compiler supported constructs

In order to simplify coding or its automatic generation a simple translation (shown in 2) table was intruduced to split more complex instruction sytnaxes into few actual instructions.

Table 2: Compiler code translation table

Instruction sytnax	**Generated code**
TRANSFER	TRANSFER
TRANSFER label	SETMP label READMP TRANSFER
TRANSFER label,c	SETMP label READMP SETC TRANSFER
TRANSFERX	TRANSFERX
TRANSFERX label	SETMP label READMP TRANSFERX
TRANSFERX label,c	SETMP label READMP SETC TRANSFERX
STC rx	STC rx
STC rx, label	SETMP label READMP STC rx

Instruction sytnax	**Generated code**
STC rx, label, ry	SETMP label READMPR ry STC rx
STC rx, label, ry, z	SETMP label READMPR ry SETZ STC rx
STNC rx	STNC rx
STNC rx, label	SETMP label READMP STNC rx
STNC rx, label, ry	SETMP label READMPR ry STNC rx
STNC rx, label, ry, z	SETMP label READMPR ry SETZ STNC rx
STZ rx	STZ rx
STZ rx, label	SETMP label READMP STZ rx
STZ rx, label, ry	SETMP label READMPR ry STZ rx
STZ rx, label, ry, z	SETMP label READMPR ry SETZ STZ rx

Instruction sytnax	**Generated code**
STNZ rx	STNZ rx
STNZ rx, label	SETMP label READMP STNZ rx
STNZ rx, label, ry	SETMP label READMPR ry STNZ rx
STNZ rx, label, ry, z	SETMP label READMPR ry SETZ STNZ rx
LOAD rx	LOAD rx
LOAD rx, label	SETMP label READMP LOAD rx
LOAD rx, label, ry	SETMP label READMPR ry LOAD rx
LOAD rx, label, ry, z	SETMP label READMPR ry SETZ LOAD rx
LDALL rx	LDALL rx
LDALL rx, label	SETMP label READMP LDALL rx
RANDINI	RANDINI
RANDINI label	SETMP label READMP

Instruction sytnax	Generated code
	RANDINI

C Detailed `VM` namespace class diagram

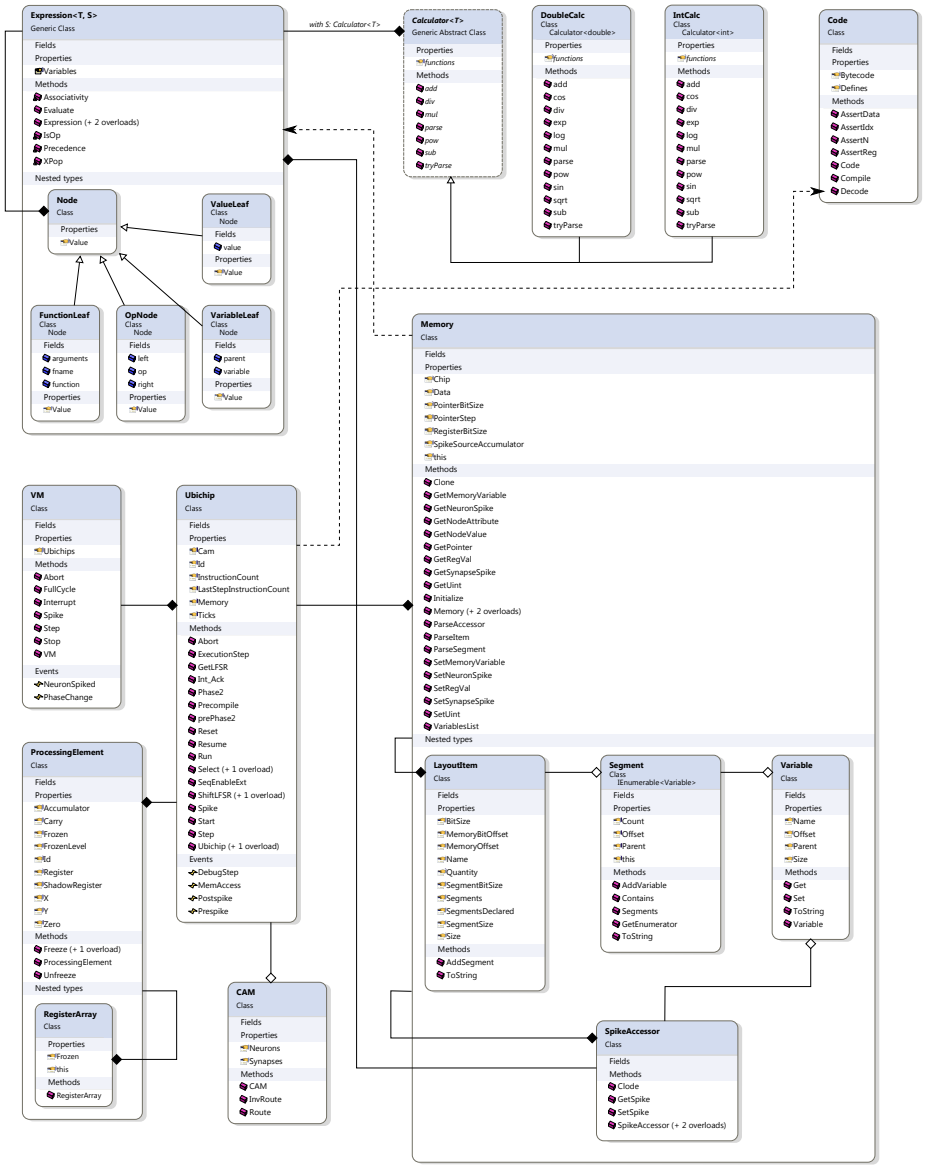


Figure 23: Virtual Machince `code` namespace class diagram

D The Expression Parser

Along with the development of the VM a several other projects smaller were made, which ultimately were merged into the main code. One such example was the Expression Parser, which at the beginning was meant only to implement basic arithmetics to allow for some flexibility in the layout file, so that it may be generic for any number of synapses or neurons per chip.

This simple parser however emerged into a more sophisticated one, as more functionalities were added. At first it was enabled to allow function calling. Of course these had to be specifically written to do so, but even though it can handle "function classes" which means one method may be responsible for handling different functions, a feature used while dealing with evaluating variables from the memory.

Another modification was a generalization of type on which the class operates, so it may natively use double or int types for it's computations. Due to the current parser/tokenizer however, it's not directly possible to use string types as values for calculations, even if one would implement `Calculator` class to handle the basic operations.

The Expression Parsers code is able to handle following:

- addition
- subtraction
- multiplication
- division
- raising to power
- prioritize by bracketing
- unary minus
- using variables
- calling functions.

These are parsed accordingly to the infix notation with the consideration of operation associativity and priority. Current implementation uses Shunting-yard algorithm to convert infix notation to Reverse Polish Notation which is used to create a parse tree for easy evaluation of the expression.

As an example we may consider following expression:

$$5^{x \times 2} \times (\sin(x - 1) + 2)$$

which can be translated into the RPN as:

$$5 \quad x \quad 2 \quad \times \quad \wedge \quad x \quad 1 \quad - \quad sin \quad 2 \quad + \quad \times$$

and then graphically represented by the tree it is parsed to:

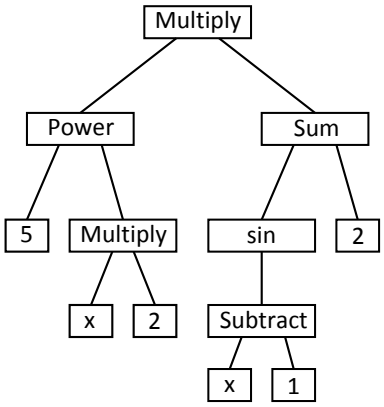


Figure 24: Example of a parse tree

The nodes in the tree are divided into following types:

- `ValueNode` – constant value leaf,
- `VariableNode` – variable valued leaf,
- `FunctionNode` – variable node, containing its arguments as separate subtrees,
- `OpNode` – operation-type nodes.

All of these nodes are subclasses of basic `Node` providing only basic value-output functionality; the first two are leaves only, the last is always a node, as it requires two operands, while the `FunctionNode` may contain other sub-nodes, but it is not necessary – e.g. a random function is parameterless.

E Assembly Example

The following pages present leaky integrate and fire algorithm initially developed by Michael Hauptvogel[4] and extended by Marc Hortas[3].

```
;4 Neurons, 2 Synapses
define size_x 4
define size_y 4
define synapses 2
.DATA
SYN-0="0ccc0ccc,0cce0ccc,7FFE7FFE,7FFE7FFE"
SYN-1="0cce0ccc,0ccc0ccc,7FFE7FFE,7FFE7FFE"
NEU-1="0ccc0ccc,0ccc0cce"
NEU-2="E188E188,E188E188"
NEU-3="00000000,00000000"
NEU-4="1FFF1FFF,1FFF1FFF"
AMAX="00000003"
DACT1="0000FFFA"
DACT2="0000012C"
DBACK="0000FEB9"
DMEM1="0000EF7D"
DMEM2="0000EF7D"
DSYN1="0000F9AE"
DSYN2="0000F9AE"
LMAX="00003FFF"
MASK1="0000E000"
MASK2="0000C000"
MMAX="00000666"
POT1="00000054"
POT2="0000FFB0"
PROB="00001FFF"
SEED="A553A75A,A554A75A"
THETA1="0000F060"
THETA2="0000F060"
UNO="00000001"
VREST1="0000E188"
VREST2="0000E188"

.CODE
;----------------INIT SOME VARIABLES-------------------------------------
LDALL R4,PROB
MOVA R4
SETMP SEED
RANDINI
RANDON
LOAD R1
RANDOFF
AND R1
MOVR R1
SWAP R1
;------------------------------------------------------------------------------------------
GOTO MAIN
;------------------------------------------------------------------------------------------
; *************************** PROCEDURES BEGIN ************************
; ------------------------ 00 Exponential Decay -------------------------
.DECAY
RST R1
MOVA R2
MOVR R4
SHL
FREEZENC
 RST R0
 SUB R2
 MOVR R2
UNFREEZE
LOOP 16
 MOVA R2
```

```
 SHL
 MOVR R2
 FREEZENC
  MOVA R1
  ADD R3
  MOVR R1
 UNFREEZE
 MOVA R3
 SHR
 MOVR R3
ENDL
MOVA R1
SHR
MOVR R1
MOVA R4
SHL
FREEZENC
 RST R0
 SUB R1
 MOVR R1
UNFREEZE
MOVA R1
MOVR R2
RST R1
RET
; ------------------------ 01 Membrane Value ------------------------------
.01MembraneValue
SWAP R5  ;SWAP TYP+SI
LDALL R3,DMEM1 ;R3=DECAY DONATOR
LDALL R4,VREST1 ;NEURON=TYPE1 (STANDARD LOAD)
MOVA R5
SHR
SHR
FREEZENC ;IF NEURON=TYPE2 (CONDITIONAL LOAD)
 LDALL R3,DMEM2 ;R3=DECAY DONATOR
 LDALL R4,VREST2
UNFREEZE
RST R2 ;R2=0
MOVA R5
SHR ;--> NEURON SPIKE
FREEZEC  ;IF (SI == 0) {
 SWAP R6  ;APPEND=((VI - VREST) * KMEM);
 MOVA R6  ;ACC=VI
 SUB R4  ;-VREST
 MOVR R2  ;R2=OPERAND
 GOTO DECAY ;PROCESS DECAY: ;R2=OPERAND, R3=DECAY DONATOR --> R2=RESULT DECAY GEHT NICHT, REST JA
UNFREEZE
   ;VI = VREST+SUMWEIGHTS+APPEND;
LDALL R4,VREST1  ;LOAD R4 START (AGAIN)
MOVA R5
SHR
SHR
FREEZENC ;IF NEURON=TYPE2 (CONDITIONAL LOAD)
 LDALL R4,VREST2
UNFREEZE  ;LOAD R4 END
MOVA R4   ;ACC=VREST (OLD VALUE OF R4!!)
SWAP R0
MOVR R1
SWAP R0
ADD R1   ;+SUMWEIGHTS
ADD R2   ;+APPEND

MOVR R6   ;ACC -> VI
SWAP R6   ;SWAP BACK VI
SWAP R5   ;SWAP BACK TYPE+SI

RST R0   ;SUMWEIGHTS=0 --> NEEDED FOR LATER!
SWAP R0   ;SWAP BACK SUMWEIGHTS
RET
; ------------------------ 02 Synaptic Weight ------------------------------
.02SynapticWeight
RST R1  ;WJI = 0 --> ALSO USED FOR ELSE
MOVA R6
```

```
SHR
FREEZENC ;IF(SJ[S]==1)
 LDALL R4,POT1 ;TYPE=0 ;ELSE
 MOVA R6
 SHR
 SHR
 FREEZENC
  LDALL R4,POT2
 UNFREEZE
 MOVFS R3;WJI[S]=(DOUBLE)A[AJI[S]]*P[S];
 MOVA R3 ;AJI SHR>> 1
 SHR
 MOVR R3
 FREEZENC
  MOVA R1 ;AJI=X1
  ADD R4 ;+ 1XPOT
  MOVR R1
 UNFREEZE
 MOVA R3
 SHR
 MOVR R3
 FREEZENC
  MOVA R1 ;AJI=1X
  ADD R4 ;+2XPOT
  ADD R4
  MOVR R1
 UNFREEZE
 MOVFS R3
 MOVA R3
 SHR
 FREEZENC
  SHR
  FREEZENC
   MOVA R1 ;AJI=11
   ADD R4 ;+1XPOT
   MOVR R1
  UNFREEZE
 UNFREEZE
UNFREEZE
SWAP R0  ;SUM --> ACC
ADD R1  ;ACC=SUM+WJI
SWAP R0  ;SR0=SUM
RET
; ------------------------ 03 Real Valued Variable -------------------------
.03RealValuedVariable
LDALL R3,DACT1
MOVA R6
SHR
SHR
 FREEZENC
  LDALL R3,DACT2
 UNFREEZE
;TYPE DATA LOAD - END
MOVFS R2; LJI->R2
GOTO DECAY ;PROCESS DECAY: ;R2=OPERAND, R3=DECAY DONATOR --> R2=RESULT DECAY
SWAP R5
MOVA R5
SWAP R5
SHR ;SI
FREEZENC
 MOVA R2
 ADD R5 ;+MJ
 MOVR R2
UNFREEZE
MOVA R6
SHR ;SJ
FREEZENC
 MOVA R2
 SWAP R4
 SUB R4 ;-MI
 SWAP R4
 MOVR R2
UNFREEZE
```

```
MOVTS R2;RES -> LJI
RET
; ------------------------ 04 Activation Variable --------------------------
.04ActivationVariable
LDALL R1,UNO
SWAP R3
MOVA R3
FREEZEZ  ; FREEZE WHEN AJI=0
 ;IF CONNECTION IS ACTIVE
 LDALL R0,LMAX  ; (LMAX-LJI) < 0 ?
 SWAP R2
 SUB R2
 SHL
 FREEZENC  ; -->YES
  MOVA R3
  ADD R1  ; AJI+1
  MOVR R3  ; AJI -> SR3
  LDALL R0,AMAX ;
  SUB R3  ; ACC=AMAX-AJI
  SHL  ; NEGATIVE?
  FREEZENC ; AJI-R1=0
   LDALL R3,AMAX
  UNFREEZE
  LDALL R0,LMAX ; LJI=LMAX/2
  SHR
  MOVR R2
 UNFREEZE
    ;ELSE IF (LJI[J] < LMIN) {
 MOVA R2   ; LJI-->ACC, LMIN=0
 SHL   ; (LJI-LMIN(=0)) < 0 ?
 FREEZENC  ; (LJI-LMIN) < 0 ? YES
  MOVA R3
  SUB R1  ; AJI-1, SR3 ACT.
  MOVR R3  ; AJI -> R3
  LDALL R0,LMAX ; LJI=LMAX/2
  SHR
  MOVR R2
 UNFREEZE
UNFREEZE
MOVA R3
FREEZENZ ;IF CONNECTION IS INACTIVE
 RST R2
UNFREEZE
SWAP R3
SWAP R2
RET
; ------------------------ 05 Memory of last presynaptic Spike -------------
.05MemoryOfLastPresynapticSpike
LDALL R3,DSYN1 ;TYPE=1
MOVA R6
SHR
SHR
FREEZENC
 LDALL R3,DSYN2 ;TYPE=2
UNFREEZE
MOVA R5
MOVR R2
GOTO DECAY ;R2=OPERAND, R3=DECAY DONATOR --> R2=RESULT DECAY
MOVA R6
SHR
FREEZENC
 LDALL R0,MMAX
 MOVR R2
UNFREEZE
MOVA R2
MOVR R5; RES IN R5
RET
; ------------------------ 06 Memory of last postsynaptic Spike -------------
.06MemoryOfLastPostsynapticSpike
LDALL R3,DSYN1 ;TYPE=1
SWAP R5
MOVA R5
SHR
```

```
SHR ;-->TYPE
FREEZENC
 LDALL R3,DSYN2 ;TYPE=2
UNFREEZE
SWAP R4    ;R2=MI
MOVA R4
SWAP R4
MOVR R2
GOTO DECAY ;R2=OPERAND, R3=DECAY DONATOR --> R2=RESULT DECAY
MOVA R5
SWAP R5
SHR ;-->SI
FREEZENC
 LDALL R0,MMAX
 MOVR R2 ;OVERWRITE DECAY RESULT
UNFREEZE
MOVA R2
SWAP R4
MOVR R4; RES IN SR4
SWAP R4
RET
; ------------------------ 07 Spike Update ---------------------------------
.07SpikeUpdate
; TYPE DATA LOAD BEGIN
LDALL R3,THETA1 ;TYPE=0 ;ELSE
SWAP R5
MOVA R5 ;TYPE+SI
SHR
SHR
FREEZENC
 LDALL R3,THETA2
UNFREEZE
 ; TYPE DATA LOAD END
MOVA R5 ;SET OUTPUTSPIKE=0
SHR
SHL
MOVR R5 ;INFO OF SI GETS LOST!
MOVA R3 ;ACC=THETA
SWAP R6
SUB R6 ;-VI
SWAP R6
SHL
FREEZENC
 MOVA R7   ; ACC <== refractary period
 SHL
 FREEZEC ; Freezeif C==1 (refractary period)
  MOVA R5
  LDALL R3,UNO
  ADD R3
  MOVR R5
  SET R7  ; activate refractary period
 UNFREEZE
UNFREEZE
SWAP R5  ;RESULT STORED IN SR5
RET
;------------------------- 08BackgroundActivity-------------------------------
.08BackgroundActivity
SWAP R7
MOVA R7
SWAP R7
MOVR R2   ; R2 <== Decay term
LDALL R3,DBACK
GOTO DECAY
SWAP R1 ; R1 <== Activation probability
LDALL R4,PROB
MOVA R4  ; ACC <== Probability
SUB R2
RANDON
CLRC
SUB R1 ; Probability - Activation probability
FREEZENC ; Probability > Activation probability
 LOAD R1 ; R1 <== new activation probability
 RANDOFF
```

```
 MOVA R4
 AND R1
 MOVR R1
 MOVA R4 ; init decay term (Probability=0)
 MOVR R2
 MOVA R7   ; ACC <== refractary period
 SHL
 FREEZEC ; C==1 ( refractary period)
  SWAP R5   ;  spike
  MOVA R5
  SHR
  SHL
  LDALL R3,UNO
  ADD R3
  MOVR R5
  SWAP R5
  SET R7  ; activate refractary period
 UNFREEZE
UNFREEZE
SWAP R1 ; SR1 <== Activation probability
MOVA R2 ; SR7 <== Decay term
SWAP R7
MOVR R7
SWAP R7
RET
;---------------------------------------------------------------------
.09REFRACTORYP
MOVA R7
SHL   ; -1ms
MOVR R7
RET
; ------------------------ 00 Neuron Load ---------------------------------
.00NeuronLoad
SWAP R6
LOAD R6,NEU-2 ;SR6<--VI
SWAP R6
SWAP R0
LOAD R0,NEU-3 ;SR0<--SUMWEIGHTS
SWAP R0
;*** ONLY SI+TYPE
LOAD R2,NEU-1 ;MI+Type+SI
MOVA R2
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SWAP R5
MOVR R5  ;TYPE+SI-->SR5
SWAP R5
```

```
;*** ONLY MI
MOVA R2
SHR
SHR
SWAP R4
MOVR R4  ;MI-->SR4
SWAP R4
LDALL R3,MASK1  ;load mask1/mask2
SWAP R5
MOVA R5
SWAP R5
SHR
SHR
FREEZENC
 LDALL R3,MASK2
UNFREEZE
LOAD R1,NEU-4  ; R1 <== refractary period::exponential
INV R3  ; ACC <== inv(mask)
AND R1  ; ACC <== exponential
MOVR R7
SWAP R7
MOVA R1  ; ACC<== refractary period::exponential
AND R3  ;ACC <== refractary period
MOVR R7
RET
; ------------------------  Neuron Save --------------------------------
.99NeuronSave
SWAP R4
SWAP R5 ;Type+SI
SWAP R1
SWAP R6
RST R3 ;R3 will HOLD MI-SI+TYPE
MOVA R4 ;14 least sign. bits
SHL
SHL
ADD R5
MOVR R3
;individual data store
RST R0
SHR
STNC R3,NEU-1 ;Mi+SI+Type
RST R0
SHR
STNC R6,NEU-2 ;VI
SWAP R0
CLRC
STNC R0,NEU-3 ;SUMWEIGHTS
SWAP R0
LDALL R3,MASK1  ; R3 <== mask1/mask2
MOVA R5
SWAP R5
SHR
SHR
FREEZENC
 LDALL R3,MASK2
UNFREEZE
MOVA R7  ; ACC <== refractary period
AND R3
SWAP R7  ;R7 <== exponential
OR R7  ;ACC <== refractary period or exponential
CLRC
STNC R0,NEU-4 ; MEM <== refractary period::exponential
RET
; ------------------------  Synapse Load --------------------------------
.00SynapseLoad
;***** 1. MJ+SI+TYPE *****
SETMP 0  ;LOAD LOOP INDEX!
READMP 1; READMPX
LOAD R2; <--MJ+SJ+TYPE
;*** ONLY SJ+TYPE
MOVA R2
SHL
SHL
```

```
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
MOVR R6  ;TYPE+SJ-->R6
;*** ONLY MJ
MOVA R2
SHR
SHR
MOVR R5  ;MJ-->R5
;***** 2. LJI+AJI *****
LOAD R2; <--LJI+AJI
;*** ONLY AJI
SWAP R3
MOVA R2
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
MOVR R3  ;AJI-->SR3
SWAP R3
;*** ONLY LJI
MOVA R2
SWAP R2
SHR
SHR
MOVR R2  ;LJI-->SR2
```

```
SWAP R2
RET
; ------------------------  Synapse Save --------------------------------
.99SynapseSave
SETMP 0  ;LOAD LOOP INDEX!
READMP 1;READMPX
;***** 1. MJ+SI+TYPE *****
MOVA R5; <--MJ
SHL
SHL
ADD R6; +TYPE+SJ
MOVR R3 ;composed DATA
RST R0
SHR
STNC R3 ;SAVE DATA
;***** 2. LJI+AJI *****
SWAP R2
MOVA R2; <--LJI
SWAP R2
SHL
SHL
SWAP R3
ADD R3; +AJI
SWAP R3
MOVR R3 ;composed DATA
RST R0
SHR
STNC R3 ;SAVE DATA
RET
;-------------------------------------Enable spikes propagation-----------------------
.SpikesEnable
SWAP R5 ; ACC <== Spikes
MOVA R5
SWAP R5
SETMP SYN-0 ; Point to Sj
READMP
SETC  ; C=1
RET
; **************************** PROCEDURES END ******************************

; **************************** MAIN PROGRAMME BEGIN ************************
.MAIN
GOTO 00NEURONLOAD
GOTO 01MEMBRANEVALUE
LOOP synapses
 GOTO 00SYNAPSELOAD
 GOTO 02SYNAPTICWEIGHT
 GOTO 03REALVALUEDVARIABLE
 GOTO 04ACTIVATIONVARIABLE
 GOTO 05MEMORYOFLASTPRESYNAPTICSPIKE
 GOTO 99SYNAPSESAVE
ENDL
GOTO 06MEMORYOFLASTPOSTSYNAPTICSPIKE
GOTO 07SPIKEUPDATE
GOTO 08BACKGROUNDACTIVITY
GOTO 09REFRACTORYP
GOTO 99NEURONSAVE
GOTO SPIKESENABLE
STOP  ; AER/CAM UPDATE OF SPIKES
GOTO MAIN
; **************************** MAIN PROGRAMME END ************************
```

F Basic algorithm memory layout

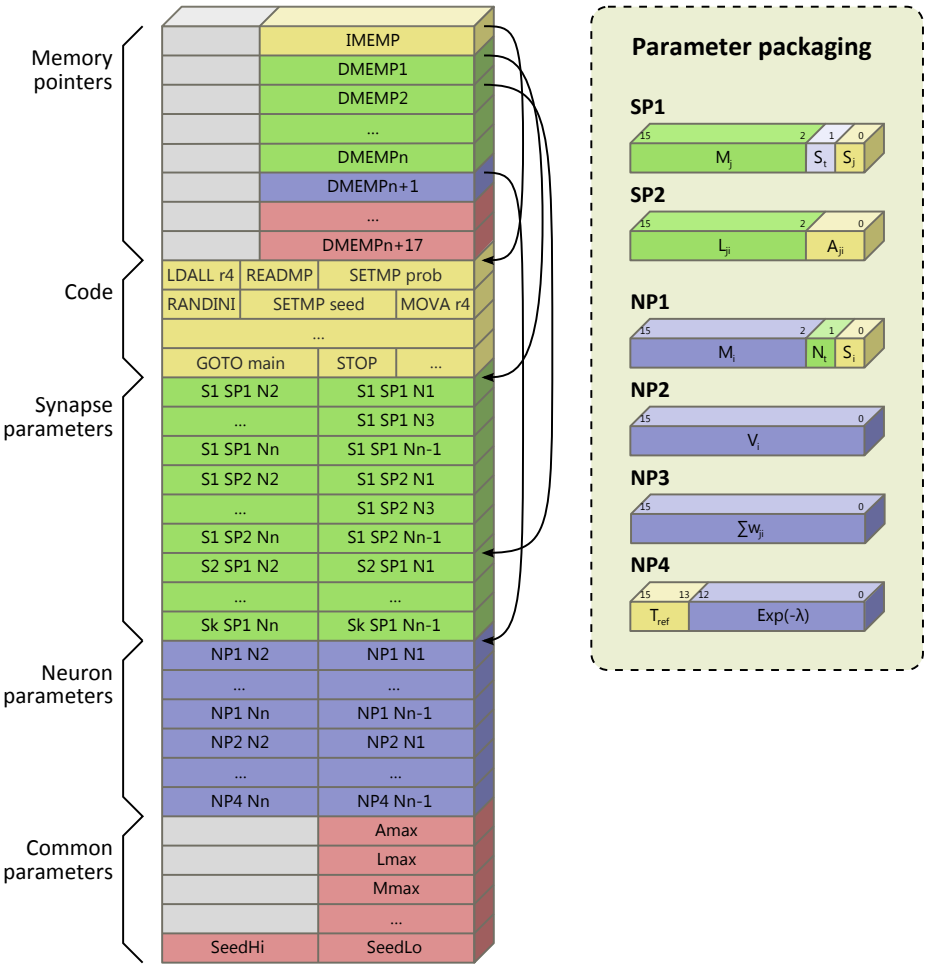


Figure 25: Memory layout for default algorithm

G Layout File Examples

The first proposed memory layout for leaky integrate and fire algorithm should be used with the following configuration file.

```
<MemoryLayout>
 <Options>
  <RegisterSize>16</RegisterSize>
  <Padding>16</Padding>
  <PointerSize>32</PointerSize>
  <PointerStep>32</PointerStep>
  <!-- spike transfer options -->
  <!--
   -index is used when field contains multiple items (ex.
       synapses), to access an item you specify an index
   -segment number
   -index and segments variables are evaluated (simple
       expression parser supporting basic operations and
       brackets)
   -available variables:
    -synapses
    -neurons
    -neuron
    -synapse
  -->
  <!-- spike source -->
  <!-- neuron_parameter_1/0/neuron/Si -->
  <!-- <SpikeSource source="memory">
   <Item>neuron_parameter_1</Item>
   <Segment>neuron</Segment>
   <Variable>Si</Variable>
  </SpikeSource> -->
  <SpikeSource source="accumulator" />
  <SpikeDest> <!-- synapses/synapse/neuron/Sj -->
   <Item>synapses</Item>
   <Index>synapse</Index>
   <Segment>neuron</Segment>
   <Variable>Sj</Variable>
  </SpikeDest>
```

```
</Options>
<Layout>
 <!-- instruction memory pointer -->
 <Item id="0" name="imemp" size="0" />
 <!-- data memory pointers -->

 <!-- size is defined in bits, how long one SP1/2 is -->
 <Item id="1" name="synapses" size="16" quantity="synapses"
     segments="2*neurons">
  <Segment>
   <Variable name="Sj" offset="0" size="1" />
   <Variable name="St" offset="1" size="1" />
   <Variable name="Mj" offset="2" size="14" />
  </Segment>
  <Segment>
   <Variable name="Aji" offset="0" size="2" />
   <Variable name="Lji" offset="2" size="14" />
  </Segment>
 </Item>

 <Item id="synapses+1" name="neuron_parameter_1" size="16"
     segments="neurons">
  <Segment>
   <Variable name="Si" offset="0" size="1" />
   <Variable name="Nt" offset="1" size="1" />
   <Variable name="Mi" offset="2" size="14" />
  </Segment>
 </Item>

 <Item id="synapses+2" name="neuron_parameter_2" size="16"
     segments="neurons">
  <Segment>
   <Variable name="Vi" offset="0" size="16" />
  </Segment>
 </Item>

 <Item id="synapses+3" name="neuron_parameter_3" size="16"
     segments="neurons">
  <Segment>
   <Variable name="Σwji" offset="0" size="16" />
```

```
 </Segment>
</Item>

<Item id="synapses+4" name="neuron_parameter_4" size="16"
    segments="neurons">
 <Segment>
  <Variable name=" Exp" offset="0" size="13" />
  <Variable name="Tref" offset="13" size="3" />
 </Segment>
</Item>

<Item name="amax" size="32">
 <Variable name="Amax" size="3" />
</Item>

<Item name="Lmax" size="32">
 <Variable name="Lmax" size="14" />
</Item>

<Item name="Mmax" size="32" />
<Item name="Theta1" size="32" />
<Item name="Theta2" size="32" />
<Item name="Pot1" size="32" />
<Item name="Pot2" size="32" />
<Item name="Vrest1" size="32" />
<Item name="Vrest2" size="32" />
<Item name="Dact1" size="32" />
<Item name="Dact2" size="32" />
<Item name="Dsyn1" size="32" />
<Item name="Dsyn2" size="32" />
<Item name="Dmem1" size="32" />
<Item name="Dmem2" size="32" />
<Item name="Uno" size="32" />
<Item name="Dback" size="32" />
<Item name="Prob" size="32" />
<Item name="Mask1" size="32" />
<Item name="Mask2" size="32" />
<Item name="Seed" size="64">
 <Variable name="SeedLo" size="32" offset="0" />
 <Variable name="SeedHi" size="32" offset="32" />
```

```
  </Item>
 </Layout>
</MemoryLayout>
```

Another example is a algorithm developed by Giovanny Sánchez Rivera which requires a different memory model. This model is described by the following memory layout file.

```
<MemoryLayout>
 <Options>
  <RegisterSize>16</RegisterSize>
  <Padding>16</Padding>
  <PointerSize>32</PointerSize>
  <PointerStep>32</PointerStep>

  <SpikeSource source="accumulator" />
  <SpikeDest source="none" />
 </Options>
 <Layout>
  <!-- instruction memory pointer -->
  <Item id="0" name="imemp" size="0" />

  <!-- data memory pointers -->
  <Item name="NP1" size="16" segments="neurons">
   <Segment>
    <Variable name="v" offset="0" size="16" />
   </Segment>
  </Item>

  <Item name="NP2" size="16" segments="neurons">
   <Segment>
    <Variable name="u" offset="0" size="16" />
   </Segment>
  </Item>

  <Item name="NP3" size="16" segments="neurons">
   <Segment>
    <Variable name="I" offset="0" size="16" />
   </Segment>
  </Item>
```

```
<Item name="NP4" size="16" segments="neurons">
 <Segment>
  <Variable name="s" offset="0" size="16" />
 </Segment>
</Item>

<Item name="NP5" size="16" segments="neurons">
 <Segment>
  <Variable name="sd" offset="0" size="16" />
 </Segment>
</Item>

<Item name="NP6" size="16" segments="neurons">
 <Segment>
  <Variable name="STDP" offset="0" size="16" />
 </Segment>
</Item>

<Item name="NP7" size="16" segments="neurons">
 <Segment>
  <Variable name="NeuronType" offset="1" size="15" />
  <Variable name="Si" offset="0" size="1" />
 </Segment>
</Item>

<Item name="Seed" size="64">
 <Variable name="SeedLo" size="32" offset="0" />
 <Variable name="SeedHi" size="32" offset="32" />
</Item>

<Item name="Prob" size="32">
 <Variable name="Prob" size="16" offset="0" />
</Item>

<Item name="CTEAE" size="32">
 <Variable name="CTEAE" size="16" />
</Item>

<Item name="CTEAI" size="32">
```

```
   <Variable name="CTEAI" size="16" />
  </Item>

  <Item name="CSTDP" size="32" />
  <Item name="CTEB" size="32" />
  <Item name="CTECE" size="32" />
  <Item name="CTECI" size="32" />
  <Item name="CTEDE" size="32" />
  <Item name="CTEDI" size="32" />
  <Item name="CTESE" size="32" />
  <Item name="CTESI" size="32" />
  <Item name="CTEIN" size="32" />
  <Item name="CTEZE" size="32" />
  <Item name="CTE30" size="32" />
  <Item name="CUNO" size="32" />
 </Layout>
</MemoryLayout>
```