HedgeHog Fused Spiking Neural Network Emulator/Compute Engine

Designed for use with RISC-V® employed as supervisor/trainer or stand-alone

Overview

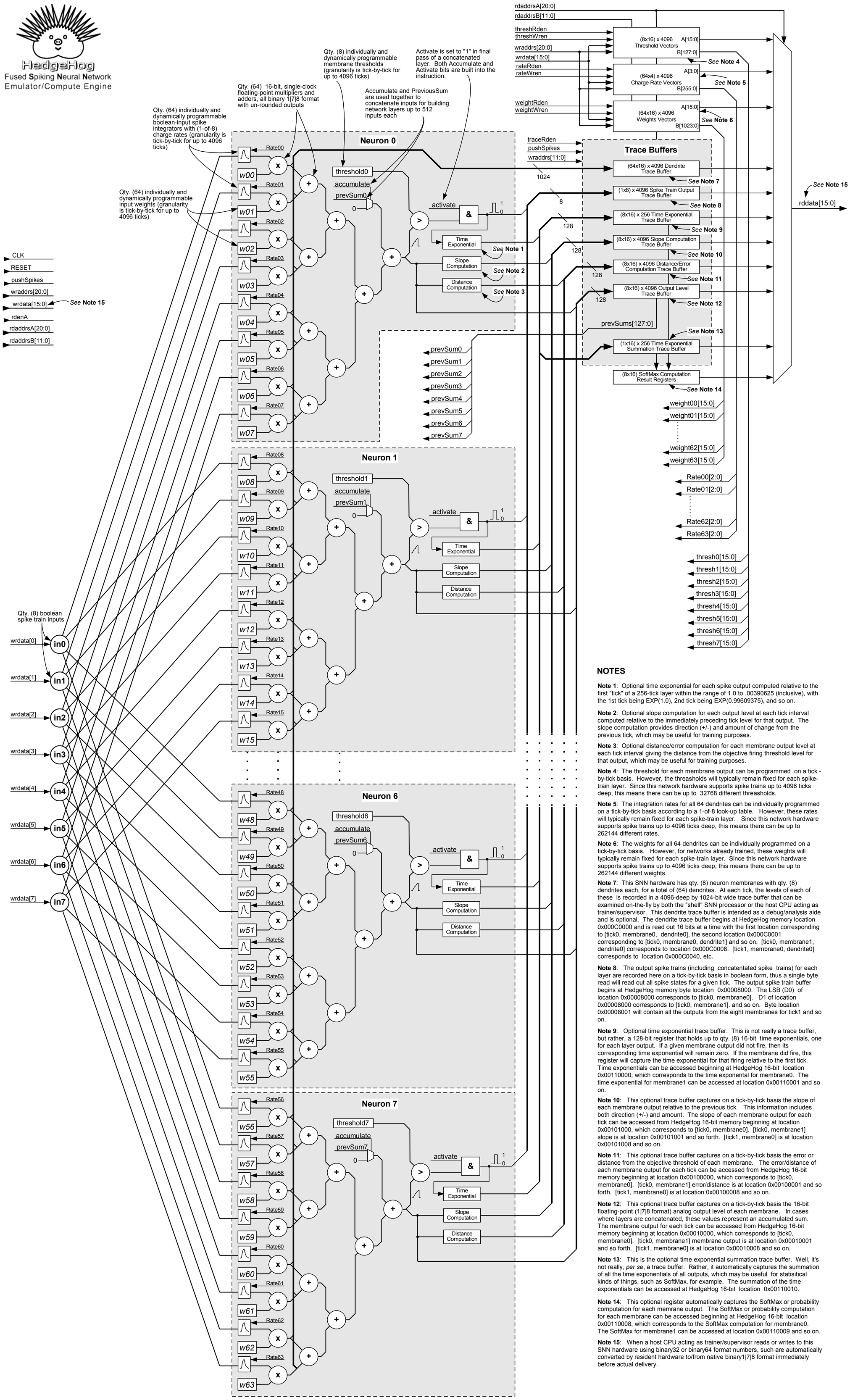
Created specifically for incorporation into a simplified mover “shell” version of the SYMPL 64-bit Universal Floating-Point Compute Engine, this memory-mapped Fused Spiking Neural Network (FSNN) is designed for ready implementation within the programmable fabric of Xilinx® Kintex® Ultra™ Plus brand FPGAs. This implementation, including the SYMPL mover “shell” that houses it and makes it go, consumes only 33% of the LUTs in Xilinx's smallest Kintex Ultra Plus device and will clock at over 100MHz in the -3 speed grade.

Dubbed the “HedgeHog” because it does this one thing so very well, it implements in hardware qty. (8) neuron membranes with qty. (8) programmable 128-tick input spike integrators each, 4096-tick real-time trace buffers for all dendrite output levels, all membrane output levels, slope computations, distance/error computations, time exponential computations, etc., such computations being done automatically in hardware using its qty. (72) multiply, (72) division, (88) addition, (72) exponential and (8) compare hardware (or equivalent) operators. Thus, clocking at 100MHz in a Xilinx Kintex Ultra Plus, it can perform roughly qty. (312) floating-point operations per clock cycle or roughly 31 billion floating-point operations per second. The Accumulate and Activate bits in the mover shell instruction set enable concatenation of these (8) neuron membranes to build layers with up to qty. (512) inputs and outputs each. The real-time monitor and data exchange feature of the mover shell enables a RISC-V or other CPU to read and write from/to the HedgeHog resources as if ordinary memory.

The Verilog test bench available at the SYMPL HedgeHog repository at GitHub automatically converts human-readable decimal character sequences for weights and thresholds, enabling you to create your test data using Google Sheets online spreadsheet. When your test run is complete, the test bench then automatically converts the binary formatted floating-point trace buffer results from each dendrite and membrane level to decimal character sequences before writing them to their respective output files, enabling you to view your results in numeric or graphic form using Google Sheets online spreadsheet. For examples of resulting graphs on data produced by the Verilog test bench simulations, refer to the reverse side of this document.

If you are exploring or experimenting with spiking neural networks for FPGA embeddable AI applications using RISC-V as host CPU, then the SYMPL HedgeHog is for you.

HedgeHog
Fused Spiking Neural Network
Emulator/Compute Engine



NOTES

Note 1: Optional time exponential for each spike output computed relative to the first "tick" of a 256-tick layer within the range of 1.0 to .00390625 (inclusive), with the 1st tick being EXP(1.0), 2nd tick being EXP(0.99609375), and so on.

Note 2: Optional slope computation for each output level at each tick interval computed relative to the immediately preceding tick level for that output. The slope computation provides direction (+/-) and amount of change from the previous tick, which may be useful for training purposes.

Note 3: Optional distance/error computation for each membrane output level at each tick interval giving the distance from the objective firing threshold level for that output, which may be useful for training purposes.

Note 4: The threshold for each membrane output can be programmed on a tick-by-tick basis. However, the thresholds will typically remain fixed for each spike-train layer. Since this network hardware supports spike trains up to 4096 ticks deep, this means there can be up to 32768 different threasholds.

Note 5: The integration rates for all 64 dendrites can be individually programmed on a tick-by-tick basis according to a 1-of-8 look-up table. However, these rates will typically remain fixed for each spike-train layer. Since this network hardware supports spike trains up to 4096 ticks deep, this means there can be up to 262144 different rates.

Note 6: The weights for all 64 dendrites can be individually programmed on a tick-by-tick basis. However, for networks already trained, these weights will typically remain fixed for each spike-train layer. Since this network hardware supports spike trains up to 4096 ticks deep, this means there can be up to 262144 different weights.

Note 7: This SNN hardware has qty. (8) neuron membranes with qty. (8) dendrites each, for a total of (64) dendrites. At each tick, the levels of each of these is recorded in a 4096-deep by 1024-bit wide trace buffer that can be examined on-the-fly by both the "shell" SNN processor or the host CPU acting as trainer/supervisor. This dendrite trace buffer is intended as a debug/analysis aide and is optional. The dendrite trace buffer begins at HedgeHog memory location 0x000C0000 and is read out 16 bits at a time with the first location corresponding to [tick0, membrane0, dendrite0], the second location 0x000C0001 corresponding to [tick0, membrane0, dendrite1] and so on. [tick0, membrane1, dendrite0] corresponds to location 0x000C0008. [tick1, membrane0, dendrite0] corresponds to location 0x000C0040, etc.

Note 8: The output spike trains (including concatentated spike trains) for each layer are recorded here on a tick-by-tick basis in boolean form, thus a single byte read will read out all spike states for a given tick. The output spike train buffer begins at HedgeHog memory byte location 0x00008000. The LSB (D0) of location 0x00008000 corresponds to [tick0, membrane0]. D1 of location 0x00008000 corresponds to [tick0, membrane1]. and so on. Byte location 0x00008001 will contain all the outputs from the eight membranes for tick1 and so on.

Note 9: Optional time exponential trace buffer. This is not really a trace buffer, but rather, a 128-bit register that holds up to qty. (8) 16-bit time exponentials, one for each layer output. If a given membrane output did not fire, then its corresponding time exponential will remain zero. If the membrane did fire, this register will capture the time exponential for that firing relative to the first tick. Time exponentials can be accessed beginning at HedgeHog 16-bit location 0x00110000, which corresponds to the time exponental for membrane0. The time exponential for membrane1 can be accessed at location 0x00110001 and so on.

Note 10: This optional trace buffer captures on a tick-by-tick basis the slope of each membrane output relative to the previous tick. This information includes both direction (+/-) and amount. The slope of each membrane output for each tick can be accessed from HedgeHog 16-bit memory beginning at location 0x00101000, which corresponds to [tick0, membrane0]. [tick0, membrane1] slope is at location 0x00101001 and so forth. [tick1, membrane0] is at location 0x00101008 and so on.

Note 11: This optional trace buffer captures on a tick-by-tick basis the error or distance from the objective threshold of each membrane. The error/distance of each membrane output for each tick can be accessed from HedgeHog 16-bit memory beginning at location 0x00100000, which corresponds to [tick0, membrane0]. [tick0, membrane1] error/distance is at location 0x00100001 and so forth. [tick1, membrane0] is at location 0x00100008 and so on.

Note 12: This optional trace buffer captures on a tick-by-tick basis the 16-bit floating-point (1|7|8 format) analog output level of each membrane. In cases where layers are concatenated, these values represent an accumulated sum. The membrane output for each tick can be accessed from HedgeHog 16-bit memory beginning at location 0x00010000, which corresponds to [tick0, membrane0]. [tick0, membrane1] membrane output is at location 0x00010001 and so forth. [tick1, membrane0] is at location 0x00010008 and so on.

Note 13: This is the optional time exponential summation trace buffer. Well, it's not really, *per se*, a trace buffer. Rather, it automatically captures the summation of all the time exponentials of all outputs, which may be useful for statisitical kinds of things, such as SoftMax, for example. The summation of the time exponentials can be accessed at HedgeHog 16-bit location 0x00110010.

Note 14: This optional register automatically captures the SoftMax or probability computation for each memrane output. The SoftMax or probability computation for each membrane can be accessed beginning at HedgeHog 16-bit location 0x00110008, which corresponds to the SoftMax computation for membrane0. The SoftMax for membrane1 can be accessed at location 0x00110009 and so on.

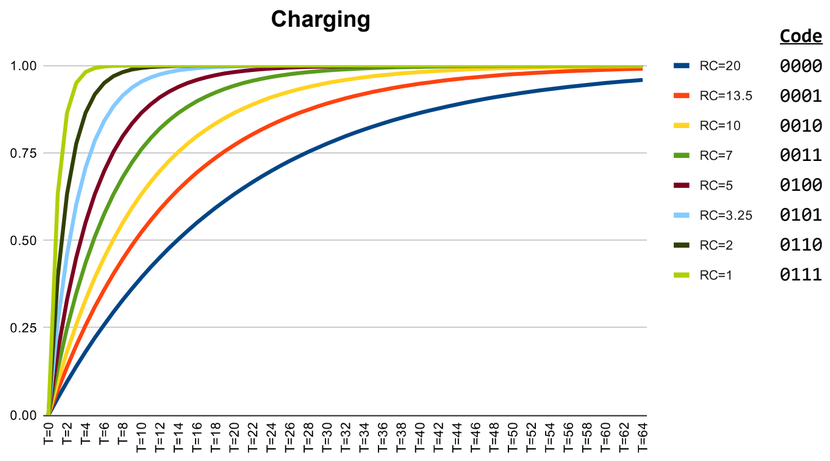
Note 15: When a host CPU acting as trainer/supervisor reads or writes to this SNN hardware using binary32 or binary64 format numbers, such are automatically converted by resident hardware to/from native binary1|7|8 format immediately before actual delivery.

Dynamically Selectable Input Spike Integration Charge Rates

Each of the 64 dendrites comprises an input spike integrator that can be dynamically configured for one of eight charge rates. Once an input spike triggers charging, the dendrite will begin charging according to the selected rate specified for that dendrite and subsequent incoming spikes on that dendrite will be ignored during this charge period until the membrane the dendrite is connected to fires or the dendrite completely decays due to the membrane not firing because its threshold was never reached. There are 8 possible charge rates that can be assigned to a given dendrite.

For charging, the dendrite output level (V_{tick}) at any tick relative to an input spike is automatically computed according to the following equation and graph (note that the legend at the bottom of the graph is the "tick" number):

$$V_{tick}=V_{weight}(1-\exp(-tick/RC))$$

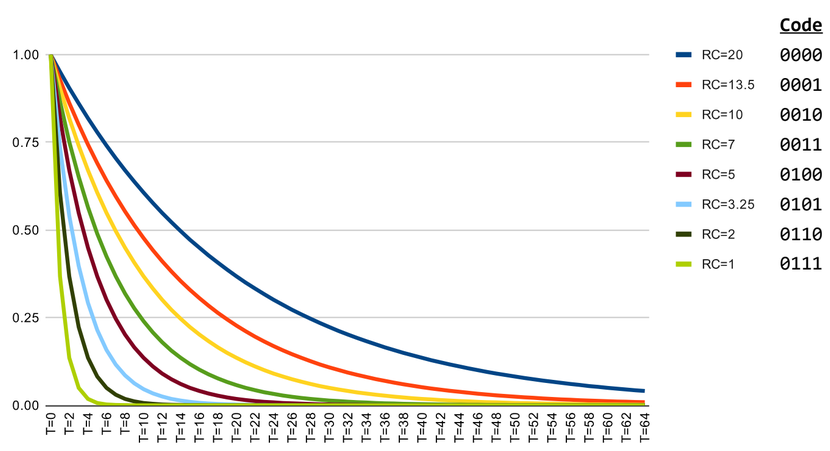


Once a dendrite charge level reaches 98% of its maximum, it is considered charged and on the next tick will automatically begin to discharge/decay according to one of eight possible rates assigned to that dendrite if the threshold of the membrane that dendrite is attached is not exceeded by the sum of all the membrane's dendrite output levels. Each rate has a 4-bit code, with code "0000" being the slowest charge rate and code "0111" being the fasted charge rate. With a code of "0111", a dendrite can be fully charged in just 4 ticks from spike input detection. With a code of "0000", the dendrite will not be fully charged until 64 ticks from spike input detection. Note that bit 3 of the code is always "0", since only 3 bits are ever used.

If the threshold of a membrane is exceeded by the sum of all of its eight dendrites, then the membrane fires, creating a spike on its output, which resets its eight dendrite levels to zero.

For discharging, the dendrite output level (V_{tick}) at any tick relative to the input spike is automatically computed according to the following equation and graph (note that the legend at the bottom of the graph is the "tick" number):

$$V_{tick}=V_{weight}(\exp(-tick/RC))$$



The charge rate codes form a 64-entry vector (one code for each of 64 dendrites), with each code residing in HedgeHog memory space on byte address boundaries starting at location 0x00080000. Thus the rate code for dendrite 00 (membrane0/dendrite0) is at location 0x00080000 and the rate code for dendrite 01 (membrane0/dendrite1) is at location 0x00080001 and so on. These locations can be read from or written to at any time by either the HedgeHog mover shell or host CPU. There are 4096 of these rate code vectors available. This means that it is possible to change the charge rate every tick. However, the usual practice is to use the same vector (and same dendrite charge rates) for all the ticks in a given layer when pushing a given spike train into it.

Dynamically Programmable Weight and Threshold Vectors

Associated with each charge rate vector are the weight and threshold vectors. Each of the 4096 weight vectors comprises qty. (64) 16-bit binary floating-point format 1|7|8 weights, one for each dendrite. Like the charge rate vectors, the weight vectors reside in asymmetrically accessible dual-port SRAM. On the "A" side of the SRAM, the weights can be individually accessed on 16-bit boundaries by the HedgeHog shell or host CPU for initialization and training purposes, while on the "B" side of the SRAM, all 64 weights are read out simultaneously by the HedgeHog shell as a 1024-bit vector during the spike-train push operation.

Weight vector memory resides in HedgeHog indirectly-addressable memory space starting at location 0x00040000. Thus the weight for dendrite 00 (membrane0/dendrite0) is at location 0x00040000 and the weight for dendrite 01 (membrane0/dendrite1) is at location 0x00040001 and so on. These locations can be read from or written to at any time by either the HedgeHog mover shell or host CPU. There are 4096 of these weight vectors available. This means that it is possible to change a given dendrite's weight every tick.

Similarly, each of the 4096 threshold vectors comprises qty. (8) 16-bit binary floating-point format 1|7|8 membrane thresholds, one for each membrane. These threshold values also reside in asymmetrically accessible dual-port SRAM. On the "A" side of the SRAM, the thresholds can be individually accessed on 16-bit boundaries by the HedgeHog shell or host CPU for initialization and training purposes, while on the "B" side of the SRAM all 8 of the thresholds are read out simultaneously by the HedgeHog shell as a 128-bit vector during the input spike-train push operation.

Threshold vector memory resides in HedgeHog indirectly-addressable memory space starting at location 0x00018000. Thus the threshold for membrane0 is at location 0x00018000 and the threshold for membrane1 is at location 0x00018001 and so on. These locations can be read from or written to at any time by either the HedgeHog mover shell or host CPU. There are 4096 of these threshold vectors available. This means that it is possible to change a given membrane's threshold every tick. But this is not the usual practice. The usual practice is to keep the thresholds constant during the entire spike train push into a given layer.

Theory of Operation

The theory of operation is really quite simple. The HedgeHog takes advantage of its instruction REPEAT capability along with its auto-post-modify indirect addressing mode. The basic format of its dual-operand instruction is as follows:

<activate/accumulate> <signal/size>:<destination> = (<signal/size>:<sourceA>, <signal/size>:<sourceB>)

Example 8-input, 8-membrane single-layer:

```
pushSpikes:
        _       _4:AR2 = _4:#binWeightsMem          ;point to weights vector
        _       _4:AR3 = _4:#spikeSrcMem            ;point to input spike train source buffer
        _       _4:AR4 = _4:#SNNinput               ;point to SNN spike train target input
        _       _2:REPEAT = _2:#31                  ;input spike train is 32 ticks deep in
                                                    ;this example
        ACT     _1:*AR4++[1] = (_1:*AR3++[1], _128:*AR3++[0])    ;push the spike train and
                                                    ;specified weights,rates, thresholds
                                                    ;into SNN input
```

In the above example, Auxiliary Registers AR2, AR3 and AR4 are initialized to point to weights/rates/thresholds vector, spike train source buffer, and fused spiking neural network input (respectively). After the REPEAT counter is loaded with the number of ticks (minus 1), the immediately following instruction is executed 32 times. "ACT" means "activate" in the mode column. ACT is set here because this is a simple single-layer push with no concatenation. During execution of this last instruction, the 32-tick spike train, along with the qty. (64) weights, qty. (64) rates, and qty. (8) thresholds, are pushed into the FSNN, all requiring only qty. (32) system clocks.

Upon completion of this last instruction, the HedgeHog will have automatically performed qty. (2304) multiply, (2304) division, (2304) exponential, (2816) addition, and (256) compare floating-point computations and will have captured in their respective trace buffers qty. (32) ticks worth of all dendrite output levels, all membrane output levels, all membrane spike train outputs, time exponentials on all membrane firings, slopes on all membrane output levels, distancece/error on all membrane output levels, a summation of all time exponentials, and a SoftMax (probability) computation on all membrane firings.

Stated another way, with just a single instruction, the HedgeHog performed qty. (9984) floating-point operations in just qty. (32) clocks. The above example presumes of course that the weights, rates and thresholds vectors have been initialized beforehand. For an example of how to do the initialization of the vectors as well as an example of how to perform concatenation of inputs to create 16, 32, 64, 256 or 512 -input/output layers, refer to the demo assembly language listing file at the SYMPL HedgeHog source code repository at GitHub: www.github.com/jerry-D

Below is an example 32-tick, 8-input spike train pushed into the HedgeHog FSNN. Note that the last half of the train is all zeros. With a charge rate of "0110" or 0x6, this is the second fastest charge rate. Thus, these ticks with all zeros in the last half of the spike train provide time for the actual input spikes that precede them to charge up and maybe cause the membrane to fire as a result of the threshold being exceeded.

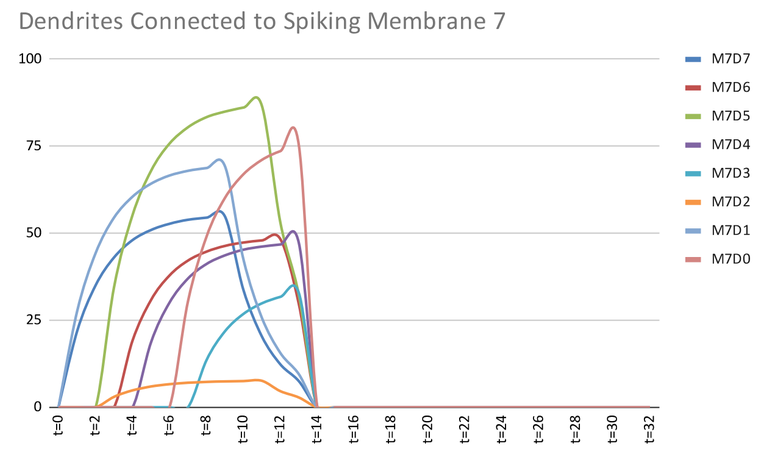
```
Input                                Spike Train
  0 - 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1 - 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  2 - 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  3 - 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  4 - 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  5 - 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  6 - 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  7 - 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      0 1 2 3 4 5 6 7 8 9 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3
                        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
                        Tick Number
```

Dendrite Trace Buffer

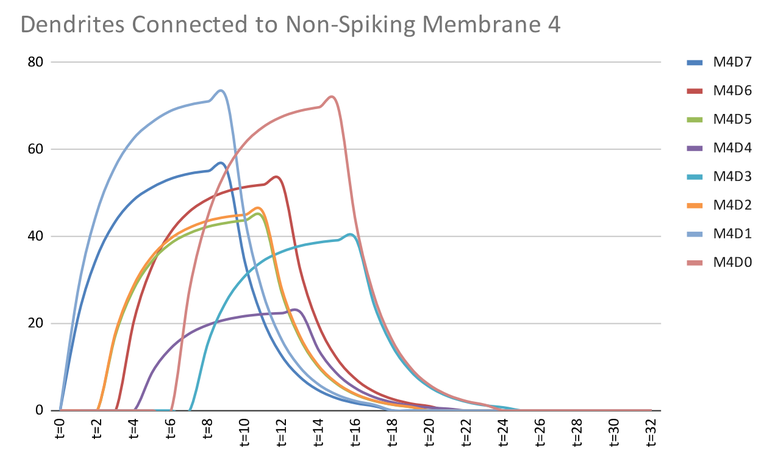
The HedgeHog FSSN Emulator/Compute Engine includes an optional 1024-bit wide by 4096-tick deep real-time trace buffer that captures all the output levels of all qty. (64) dendrites, which may be useful for debug and analysis purposes. The levels of each dendrite are stored in binary floating-point 1|7|8 format, but the Verilog test bench that is included in the HedgeHog repository at GitHub, converts these to human-readable decimal character sequences before writing it to the "dendrites.txt" file. The textual contents of this file can then be copied and pasted directly into Google Sheets online spreadsheet and plotted. Below are plots generated by Google Sheets from the actual simulation run using the 32-tick spike train shown previously.

The dendrite trace buffer begins at HedgeHog memory location 0x000C0000 and is read out 16 bits at a time with the first location corresponding to [tick0, membrane0, dendrite0], the second location 0x000C0001 corresponding to [tick0, membrane0, dendrite1] and so on. [tick0, membrane1, dendrite0] corresponds to location 0x000C0008. [tick1, membrane0, dendrite0] corresponds to location 0x000C0040, etc.

The plot below shows all the output levels of all 8 dendrites connected to Membrane 7, which fires at around tick# 14. Note how the dendrites simultaneously discharge to zero upon firing of the membrane they are connected to.



The plot below shows all the output levels of all 8 dendrites connected to Membrane 4, which never fires. Note how the dendrites gradually decay to zero over time according to the specified charge/decay rate.



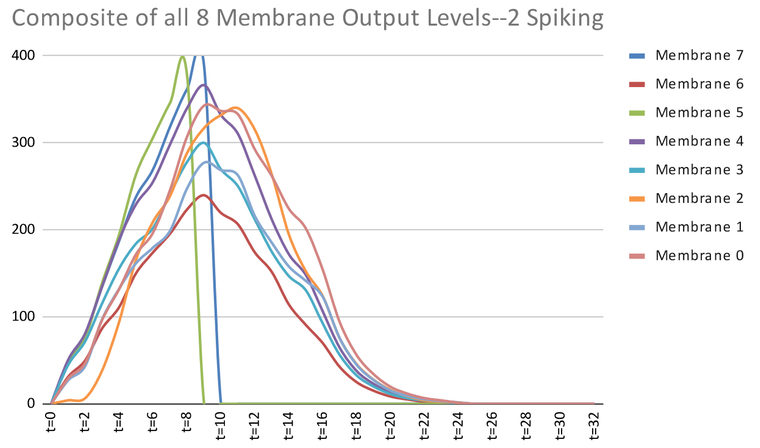
Membrane Output Level Trace Buffer

The HedgeHog FSSN Emulator/Compute Engine includes an optional 128-bit wide by 4096-tick deep real-time trace buffer that captures all the output levels of all qty. (8) membranes, which may be useful for debug, analysis and SNN training purposes. In cases where layers are concatented, these values represent an accumulated sum.

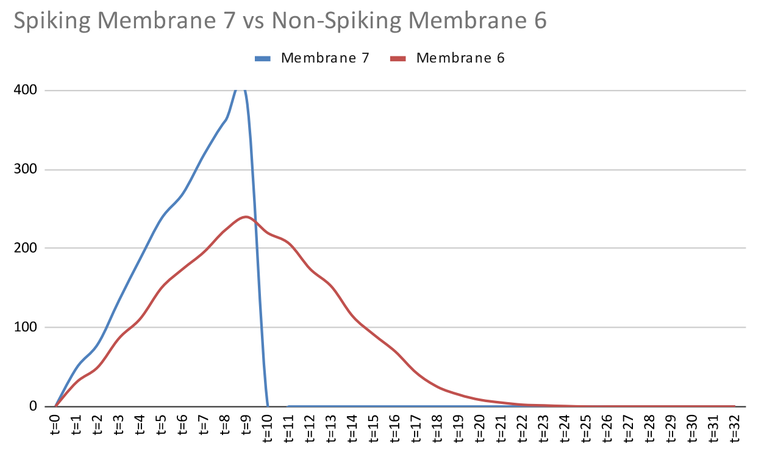
The levels of each membrane are stored in binary floating-point 1|7|8 format, but the Verilog test bench that is included in the HedgeHog repository at GitHub, converts these to human-readable decimal character sequences before writing it to the "outputLevels.txt" file. The textual contents of this file can then be copied and pasted directly into Google Sheets online spreadsheet and plotted. Below are plots generated by Google Sheets from the actual simulation run using the 32-tick spike train shown previously.

The membrane output for each tick can be accessed from HedgeHog 16-bit memory beginning at location 0x00010000, which corresponds to [tick0, membrane0]. [tick0, membrane1] membrane output is at location 0x00010001 and so forth. [tick1, membrane0] is at location 0x00010008 and so on.

The plot below shows all the output levels of all 8 membranes, with Membrane 5 and Membrane 7 firing at tick# 9 and 10 respectively along with the other membranes gradually decaying over time because their thresholds were never reached.



The plot below shows the distinction between a membrane spiking because its threshold was exceeded and a non-spiking membrane that gradually decays over time because its threshold was never exceeded.

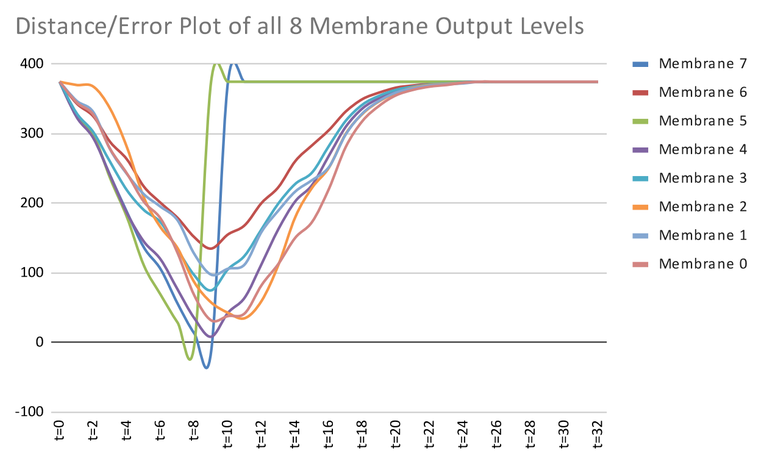


Distance/Error Trace Buffer

The HedgeHog FSSN Emulator/Compute Engine includes an optional 128-bit wide by 4096-tick deep real-time trace buffer that captures all the distance/error between of all 8 membrane output levels and their respective firing thresholds, which may be useful for debug, analysis and SNN training purposes. These distance values are stored in binary floating-point 1|7|8 format, but the Verilog test bench that is included in the HedgeHog repository at GitHub, converts these to human-readable decimal character sequences before writing it to the "errAmounts.txt" file. The textual contents of this file can then be copied and pasted directly into Google Sheets online spreadsheet and plotted. Below are plots generated by Google Sheets from the actual simulation run using the 32-tick spike train shown previously.

The error/distance of each membrane output for each tick can be accessed from HedgeHog 16-bit memory beginning at location 0x00100000, which corresponds to [tick0, membrane0]. [tick0, membrane1] error/distance is at location 0x00100001 and so forth. [tick1, membrane0] is at location 0x00100008 and so on.

The plot below shows a composite of all 8 membrane distance/error computations, which can be accessed on-the-fly by the HedgeHog mover shell or the host CPU.

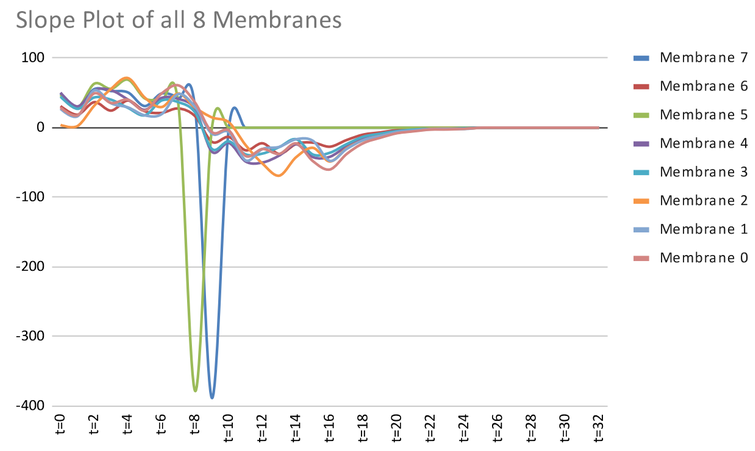


Slope Trace Buffer

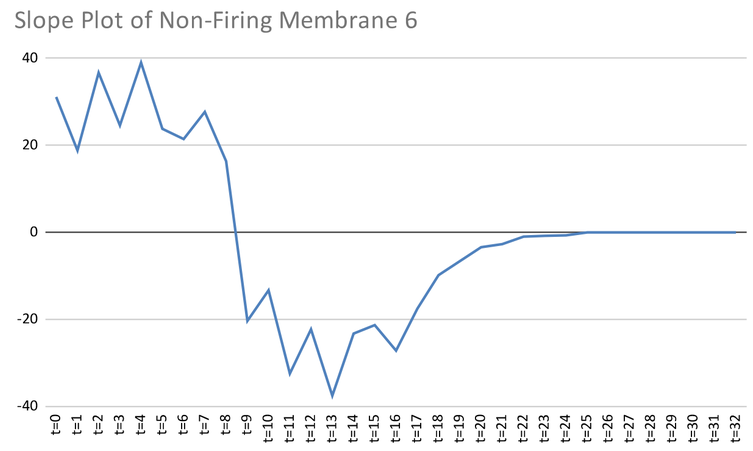
The HedgeHog FSSN Emulator/Compute Engine also includes an optional 128-bit wide by 4096-tick deep real-time trace buffer that captures all the slope and rate of change of the 8 membrane output levels, which may be useful for debug, analysis and SNN training purposes. These slope/direction/magnitude values are stored in binary floating-point 1|7|8 format, but the Verilog test bench that is included in the HedgeHog repository at GitHub, converts these to human-readable decimal character sequences before writing it to the "slopeAmounts.txt" file. The textual contents of this file can then be copied and pasted directly into Google Sheets online spreadsheet and plotted. Below are plots generated by Google Sheets from the actual simulation run using the 32-tick spike train shown previously.

The slope of each membrane output for each tick can be accessed from HedgeHog 16-bit memory beginning at location 0x00101000, which corresponds to [tick0, membrane0]. [tick0, membrane1] slope is at location 0x00101001 and so forth. [tick1, membrane0] is at location 0x00101008 and so on.

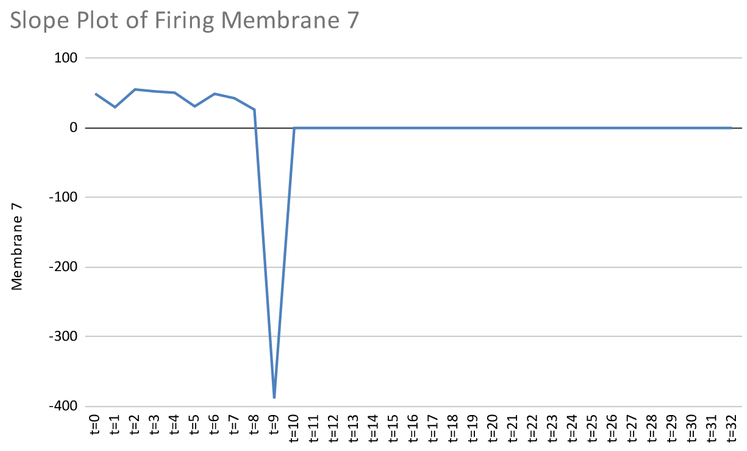
The plot below shows a composite of all 8 membrane slope computations, which can be accessed on-the-fly by the HedgeHog mover shell or the host CPU.



The plot below shows the computed slopes, tick-by-tick, of Membrane 6, which never fires because its threshold is never exceeded.



The plot below shows the computed slopes, tick-by-tick, of Membrane 7, which fires because its firing threshold is exceeded.



Time Exponential Trace Buffer

The HedgeHog FSSN Emulator/Compute Engine also includes an optional 128-bit wide real-time trace buffer that captures the automatically-computed time exponentials of membranes that have fired, which may be useful for debug, statistical analysis and classification purposes. These exponentials are stored in binary floating-point 1|7|8 format, but the Verilog test bench that is included in the HedgeHog repository at GitHub, converts these to human-readable decimal character sequences before writing it to the "exponentials.txt" file. The textual contents of this file can then be copied and pasted directly into Google Sheets online spread-sheet and plotted.

Time exponentials can be accessed beginning at HedgeHog 16-bit location 0x00110000, which corresponds to the time exponental for membrane0. The time exponential for membrane1 can be accessed at location 0x00110001 and so on.

Time Exponential Summation Register

After each activation pass for a given layer, the HedgeHog's optional 16-bit time exponential summation register will contain the results of an automatically-computed summation of all the time exponentials of membranes that have actually fired, which may be useful for debug, statistical analysis and classification purposes, like, for example a SoftMax computation. This is stored in binary floating-point 1|7|8 format, but the Verilog test bench that is included in the HedgeHog repository at GitHub, converts these to human-readable decimal character sequences before writing it to the "summation.txt" file.

The summation of the time exponentials can be accessed at HedgeHog 16-bit location 0x00110010.

SoftMax Trace Buffer

The HedgeHog FSSN Emulator/Compute Engine also includes an optional 128-bit wide real-time trace buffer that captures the automatically-computed SoftMax using time exponential computations of membranes that have fired, which may be useful for debug, statistical analysis and classification purposes. These SoftMax results are stored in binary floating-point 1|7|8 format, but the Verilog test bench that is included in the HedgeHog repository at GitHub, converts these to human-readable decimal character sequences before writing it to the "softMax.txt" file. The textual contents of this file can then be copied and pasted directly into Google Sheets online spread-sheet and plotted.

The SoftMax or probability computation for each membrane can be accessed beginning at HedgeHog 16-bit location 0x00110008, which corresponds to the SoftMax computation for membrane0. The SoftMax for membrane1 can be accessed at location 0x00110009 and so on.

Membrane Spike Train Output Trace Buffer

The HedgeHog FSSN Emulator/Compute Engine also includes a 128-bit wide real-time trace buffer that captures in Boolean form spike trains of all eight membrane outputs. For implementations that comprise more than one layer, these would be pushed back into the HedgeHog using the REPEAT instruction and pointing to the next unused block of ticks for storage, continuing in this manner until all layers have been pushed.

The output spike trains (including concatentated spike trains) for each layer are recorded here on a tick-by-tick basis in Boolean form, thus a single byte read will read out all spike states for a given tick. The output spike train buffer begins at HedgeHog memory byte location 0x00008000. The LSB (D0) of location 0x00008000 corresponds to [tick0, membrane0]. D1 of location 0x00008000 corresponds to [tick0, membrane1]. and so on. Byte location 0x00008001 will contain all the outputs from the eight membranes for tick1 and so on.

Below are the contents of the membrane spike train output trace buffer of the actual simulation run using the 32-tick input spike train shown previously.

```
Output                          Spike Train
  0 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  2 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  3 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  4 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  5 - 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  6 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  7 - 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      0 1 2 3 4 5 6 7 8 9 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3
                          0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
                          Tick Number
```

Using Google Sheets Online Spreadsheet

All the above graphic plots of actual simulation results were produced using Google Sheets online spreadsheet application. To access and try out the actual spreadsheet used for the plots in this document, click on the link below:

https://docs.google.com/spreadsheets/d/1R1bdN-xH0Cou0wdah1Oj3k0VNP_2UTwp1fSHnL2JVNc/edit#gid=2851398

Note that in the spreadsheet and, consequently, in the above plots, the values were scaled up by a factor of 100 because the spreadsheet apparently will not plot values less than 1.

Simple Single-Clock Interface

Interfacing a CPU like the RISC-V to the SYMPL HedgeHog is not much different than interfacing to a synchronous true dual-port block SRAM of the type found in most FPGAs. The only real difference is that the HedgeHog provides a "Done" signal, which when active "1", indicates that the HedgeHog is available for the next task. When "0", this means the HedgeHog is still busy working on a task.

All communications between the host CPU and the HedgeHog go through this interface, with read and write transactions occurring with the same kind of timing as a synchronous block SRAM, meaning both reads and writes are synchronous with the rising edge of CLK. Through this port the host CPU has real-time, on-the-fly, access to all of the HedgeHog's resources, including PC, Auxiliary Registers, Status Register, REPEAT Counter, trace buffers, debugger, weight, rate, threshold vector memory, etc., all without the use of interrupts, monitor micro-kernals, DMA, or any special protocols.

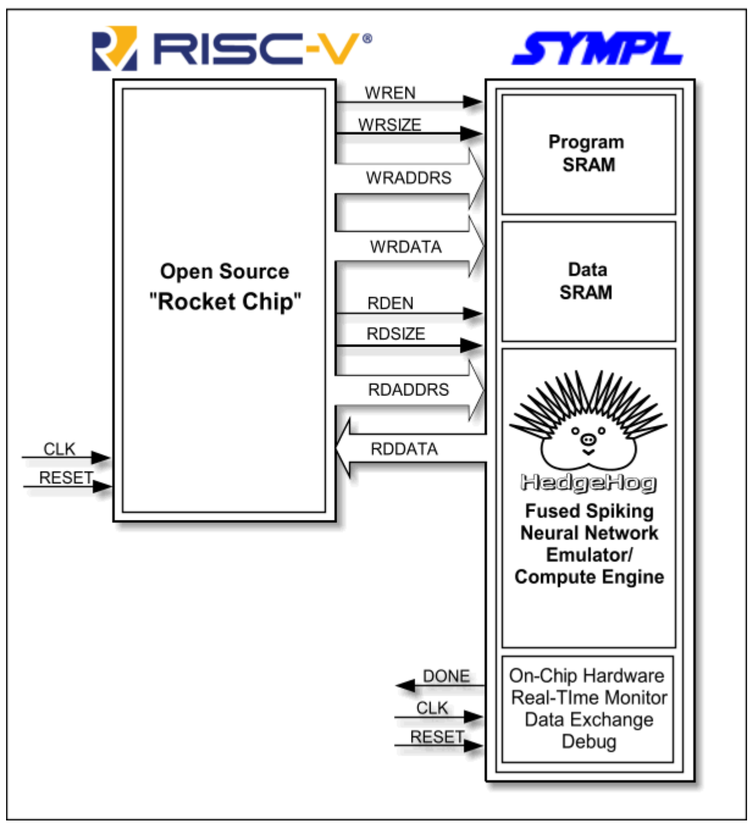
Below is the Verilog instantiation used in the test-bench included with the HedgeHog Verilog RTL source code, which emulates a host CPU like the RISC-V for these transactions.

```
HedgeHog shell(
   .CLK     (clk),
   .RESET   (reset),
   .wren    (HostWren),
   .wrSize  (HostWrSize[2:0]),
   .wrdata  (HostWrdata[63:0]),
   .wraddrs (HostWraddrs[31:0]),
   .rden    (HostRden),
   .rdSize  (HostRdSize)[2:0],
   .rdaddrs (HostRdaddrs[31:0]),
   .rddata  (HostRddata[63:0]),
   .done    (done)
   );
```

In the table below are descriptions of the signals for the module ports of the above instantiation.

Signal	Direction	Size	Description
CLK	input	1	System clock
RESET	input	1	System reset, active high, two complete clock cycles minimum
wren	input	1	Active high write enable
wrSize	input	3	Write size in bytes
wrdata	input	64	Write data
wraddrs	input	32	Write address
rden	input	1	Active high read enable
rdSize	input	3	Read size in bytes
rdaddrs	input	32	Read address
rddata	output	64	Read data
done	output	1	Done; 1=done, 0=busy

The block diagram below shows how simple a direct connection between the RISC-V CPU and the SYMPL HedgeHog can be. Other approaches such as the use of an AXI-4 or AXI-Lite master/slave arrangement can also be used. In such a configuration, the RISC-V can be employed as a training supervisor or simply be used to initialize the weight, rate and threshold vectors inside the HedgeHog in scenarios where the HedgeHog is being employed as a "PLAYER-only" SNN, wherein the vector data sets are already trained.



Unsupervised Training

As mentioned briefly at the beginning of this document, the SYMPL HedgeHog Fused Spiking Neural Network Emulator/Compute Engine comprises a "mover" shell and the HedgeHog FSNN circuit. The mover shell is a scaled-down version of the original Universal Floating-Point ISA Compute Engine published earlier this year at the SYMPL GitHub repository. To find out more about the complete hardware implementation of the IEEE® 754-20XX specification, visit the SYMPL GitHub repository and download the "UFP_ISA.pdf" documentation published there. In so doing, understand that in creating this minimum "mover" shell, even though all IEEE-754 floating-point operators have been removed from the original design, the instruction set has not changed at all.

This is because the SYMPL Universal Floating-Point ISA does not employ opcodes—it is fundamentally a "mover" architecture and all operators are memory-mapped. For the HedgeHog implementation, it's a fused spiking neural network with real-time trace capability housed in this stripped down, but very fast and efficient mover shell. This mover shell has built into its architecture a hardware real-time monitor and data exchange capability that does not rely on a software micro-kernel, but rather is all done in hardware, giving the host CPU complete real-time visibility and access into the HedgeHog without the use of interrupts, DMAs and the like.

Because the current HedgeHog implementation presently employs no IEEE 754 floating-point operators with which to perform the kinds of computations needed for training, it requires a host CPU attached to it that has such computational capability to provide the supervised training the current HedgeHog FSNN requires.

On the other hand, because the original SYMPL Universal Floating-Point ISA included the entire IEEE 754-2008 repertoire of mandated operators implemented in hardware, it would be a simple matter to re-install these operators. Once done and with the proper unsupervised training algorithms operating under it, this new Super HedgeHog can train itself.

As it turns out, with the HedgeHog fused spiking neural network presently employing 16-bit 1|7|8 binary format operators, it is highly likely it can get by without the "basic" binary32 or binary64 format operators installed, but rather with the much faster and smaller 16-bit 1|7|8 binary format operators. By doing this, not only will latencies be reduced to near zero and power consumption minimized, but such an implementation will consume fewer resources and clock faster to boot.

Below is a table showing the 1|7|8 binary format floating-point operators presently available for implementation in a Super HedgeHog:

Operation	Latency
addition	1
subtraction	1
multiplication	1
division	4
squareRoot	1
fusedMultiplyAdd	2
logB	1
scaleB	1
remainder	7
log	1
exp	1
convertFromDecimalCharacter	6
convertToDecimalCharacter	6

For more information, contact:
sympl.gpu@gmail.com
jerry.harthcock@gmail.com

Or visit:
www.github.com/jerry-D