MANCHESTER 1824
The University of Manchester

Some fixed-point arithmetic tricks on SpiNNaker

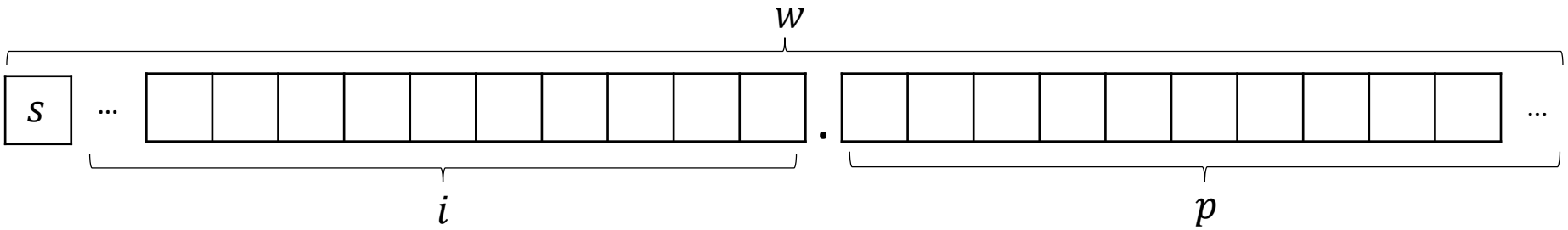
Mantas Mikaitis

SpiNNaker meeting, 26th September 2018

Contents of the presentation

1. Reminder of fixed-point arithmetic
2. Trick 1: Accum type multiplication with added rounding improves Izhikevich neuron spike lag.
3. Trick 2: Mixing accum and long fract types to improve the accuracy of exponential decay (expk() function)

Generalized fixed-point number representation



- i - number of integer bits
- p - number of fractional bits (precision)
- Range:
  $[-2^i, 2^i - 2^{-p}]$ (signed),
  $[0, 2^i - 2^{-p}]$ (unsigned)

- $\epsilon = 2^{-p}$(Machine epsilon)
- Smallest positive value is ϵ (gap between any two neighbouring numbers).
- Accuracy is measured in ϵ.
- The maximum error is: $\frac{\epsilon}{2}$

<u>Converting to decimals:</u>

Given a binary number: $< s, i, p > - sI_{i-1}I_{i-2}\cdots I_0.F_{-1}F_{-2}\cdots F_{-p}$

If signed format and s is set, invert all bits and add 1. Then:

$$value = -\left(\sum_{k=0}^{i-1} I_k 2^k + \sum_{j=1}^{p} F_j 2^{-j} + s2^i\right), else$$

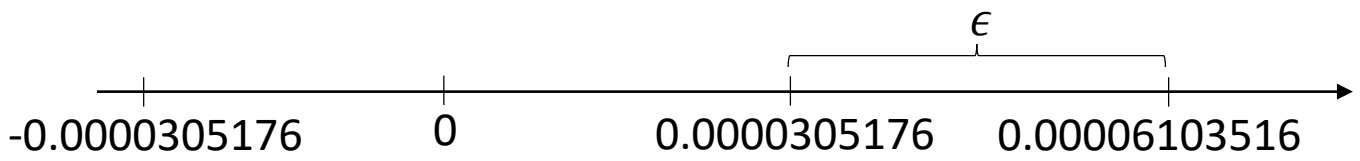
$$value = \sum_{k=0}^{i-1} I_k 2^k + \sum_{j=1}^{p} F_{-j} 2^{-j}$$

GCC types *accum* and *long fract*

- Accum <s, 16, 15>:

$$\epsilon = 2^{-15} \approx 0.0000305176\ldots$$

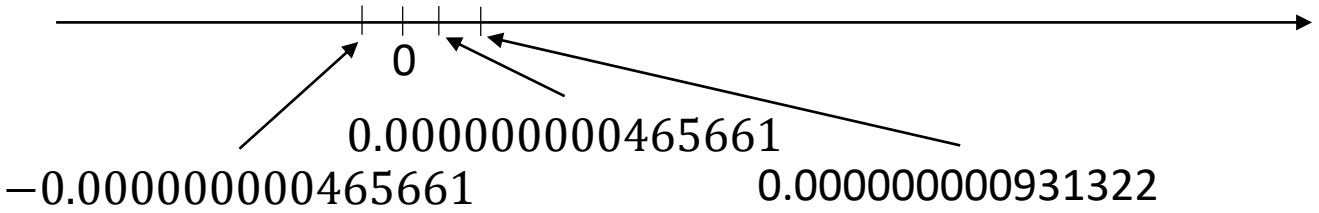
Range: $[-2^{-15} = -65536, 2^{16} - 2^{-15} \approx 65535.99996948\ldots]$



- long fract <s, 0, 31>:

$$\epsilon = 2^{-31} \approx 0.000000000465661$$

Range: $[0, 2^{0} - 2^{-31} \approx 0.999999999534339\ldots]$



Arithmetic operations on fixed-point numbers

- Addition:

$$< s, i, p > + < s, i, p > = < s, i, p >$$

- Subtraction:

$$< s, i, p > - < s, i, p > = < s, i, p >$$

Note: +,- denote an integer operation, available in most of the processors, including ARM968 – special treatment is not required to implement this.

- Multiplication:

$$< s, i_a, p_a > \times < s, i_b, p_b > = < s, i_a + i_b, p_a + p_b >$$

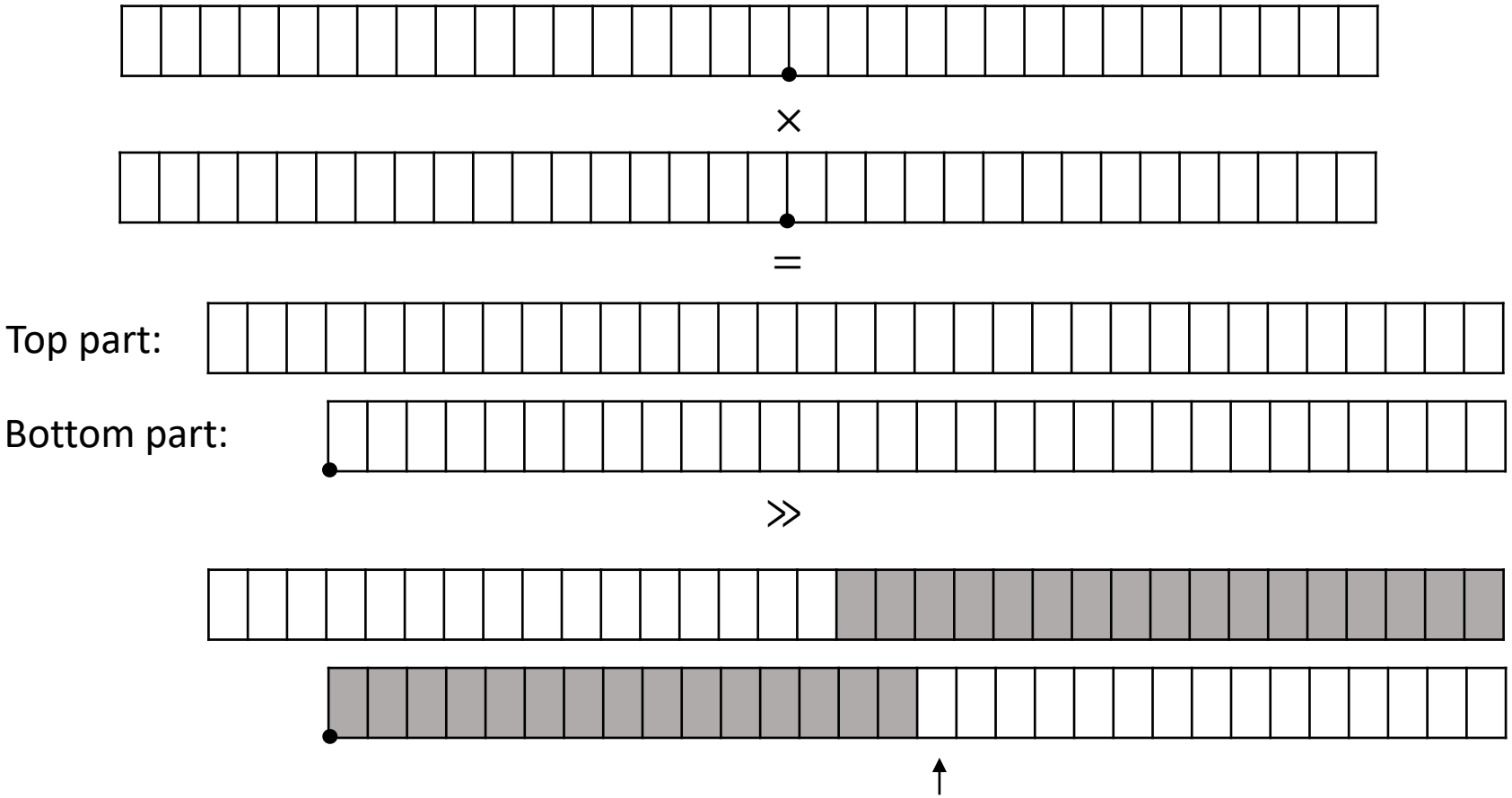
Note: x denotes integer multiplication, however, if the operands a and b had the word lengths w_a and w_b, the result will have the word length of $w_a + w_b$. Therefore, we also need to shift right to put the result back in one of the operand's formats. This results in loss of precision.

- Division:

$$< s, i, p_a > \div < s, i, p_b > = < s, i, p_a - p_b >$$

Note: loss of precision now occurs right away inside the integer divider. Usual solution is to pre-shift the dividend.

Rounding and its role in multiplication



If we truncate the number, we can introduce an error of up to $\epsilon = 0.0000305176$.

Rounding and its role in multiplication

- Round to nearest is a simple rounding mode: if the most significant bit of the truncated vector is set, add 1 to the result:

$$Round(IN: x, < s, i, p >) = \begin{cases} \lfloor x \rfloor & if\ \lfloor x \rfloor \leq x < \lfloor x \rfloor + \frac{\epsilon}{2} \\ \lfloor x \rfloor + \epsilon & if\ \lfloor x \rfloor + \frac{\epsilon}{2} \leq x < \lfloor x \rfloor + \epsilon \end{cases}$$

where $\lfloor x \rfloor$ denotes the truncation of the number x to the given fixed point format.

Rounding and its role in multiplication (Optional ARM assembly slide)

- GCC implements no rounding in the multiplication of *accums* (see comments in GCC's arm-fixed.md file):

```
SMULL   R4, R5, R7, LR
LSL R7, R5, #17
ORR R4, R7, R4, LSR #15
```

Trick1: Add rounding to GCC multiplication of two *accums* to improve the Izhikevich neuron

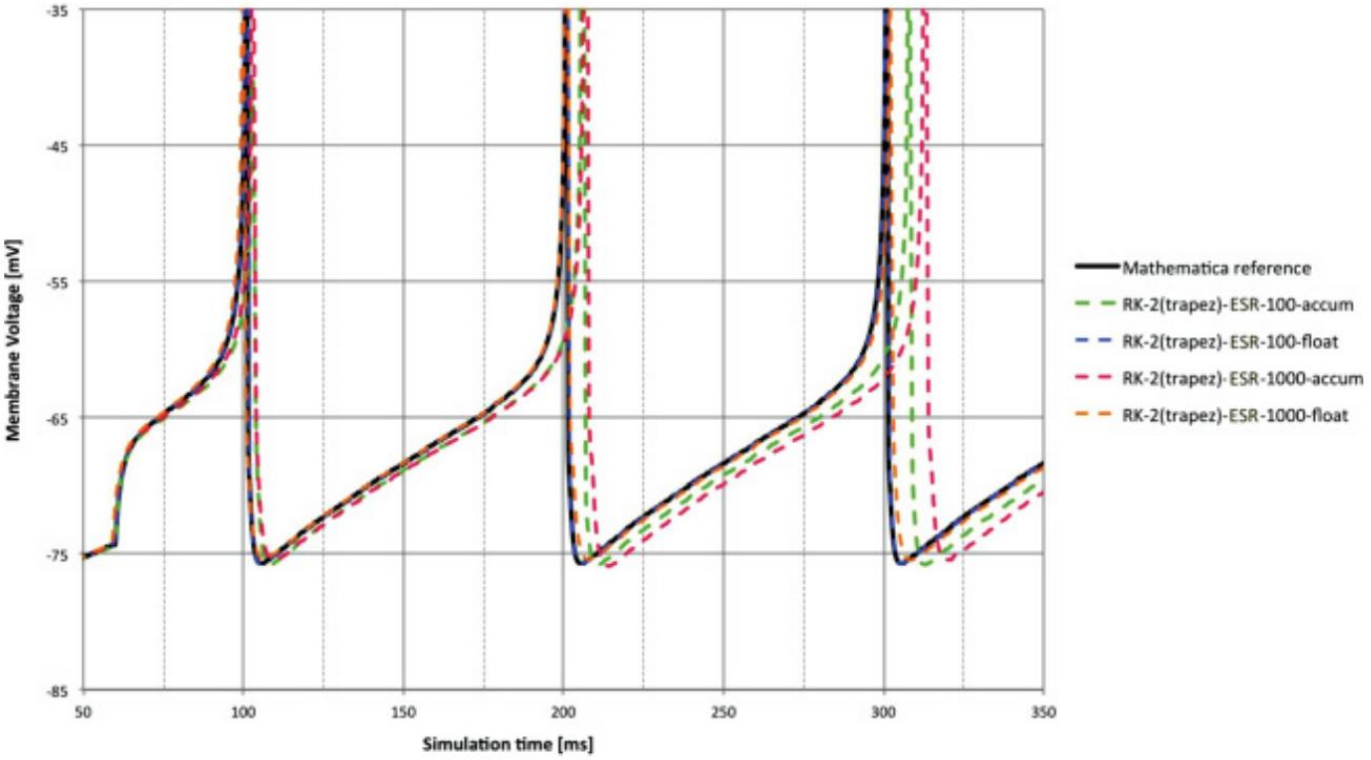
- There are 12 accum multiplications in the SpiNNaker Izhikevich neuron:

```
static inline void _rk2_kernel_midpoint(REAL h, neuron_pointer_t neuron,
                                        REAL input_this_timestep) {

    // to match Mathematica names
    REAL lastV1 = neuron->V;
    REAL lastU1 = neuron->U;
    REAL a = neuron->A;
    REAL b = neuron->B;

    REAL pre_alph = REAL_CONST(140.0) + input_this_timestep - lastU1;
    REAL alpha = pre_alph
                 + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * lastV1) * lastV1;
    REAL eta = lastV1 + REAL_HALF(h * alpha);

    // could be represented as a long fract?
    REAL beta = REAL_HALF(h * (b * lastV1 - lastU1) * a);

    neuron->V += h * (pre_alph - beta
                      + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * eta) * eta);

    neuron->U += a * h * (-lastU1 - beta + b * eta);
}
```

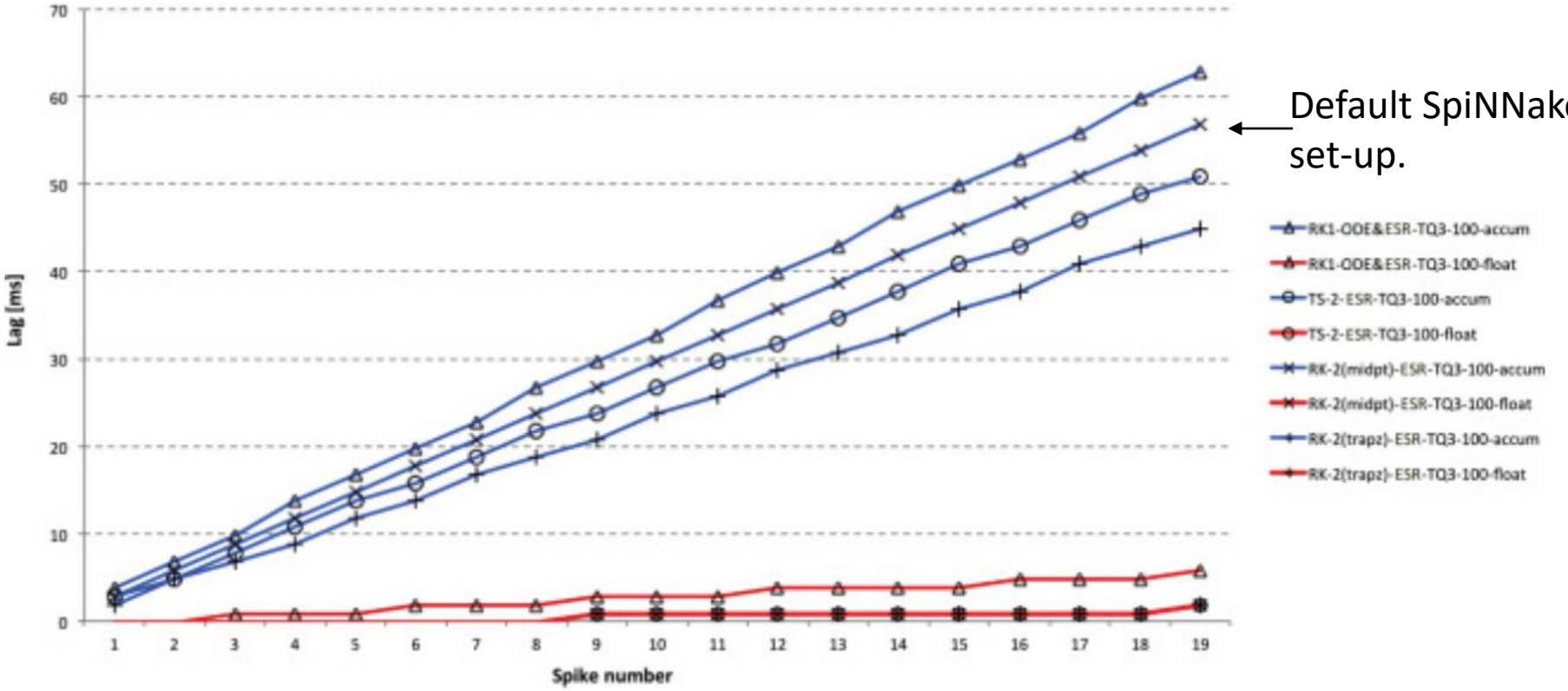
- The basic trick is to replace all of these stars with multiplications and rounding from spinn_common/include/stdfix-full-iso.h in the SpiNNaker code base (Careful with the order of multiplications!).

Spike lag in the Izhikevich neuron model (M.Hopkins and S.Furber, Neur. Comp. 2015)

- The parameters of the neuron are: a = 0.02, b = 0.2, c =−65 mV, d = 8. V=−75 mV U=0. Stimulation at 60 ms, a 4.775 nA DC current is delivered and sustained (DC input). Timestep is 0.1ms.

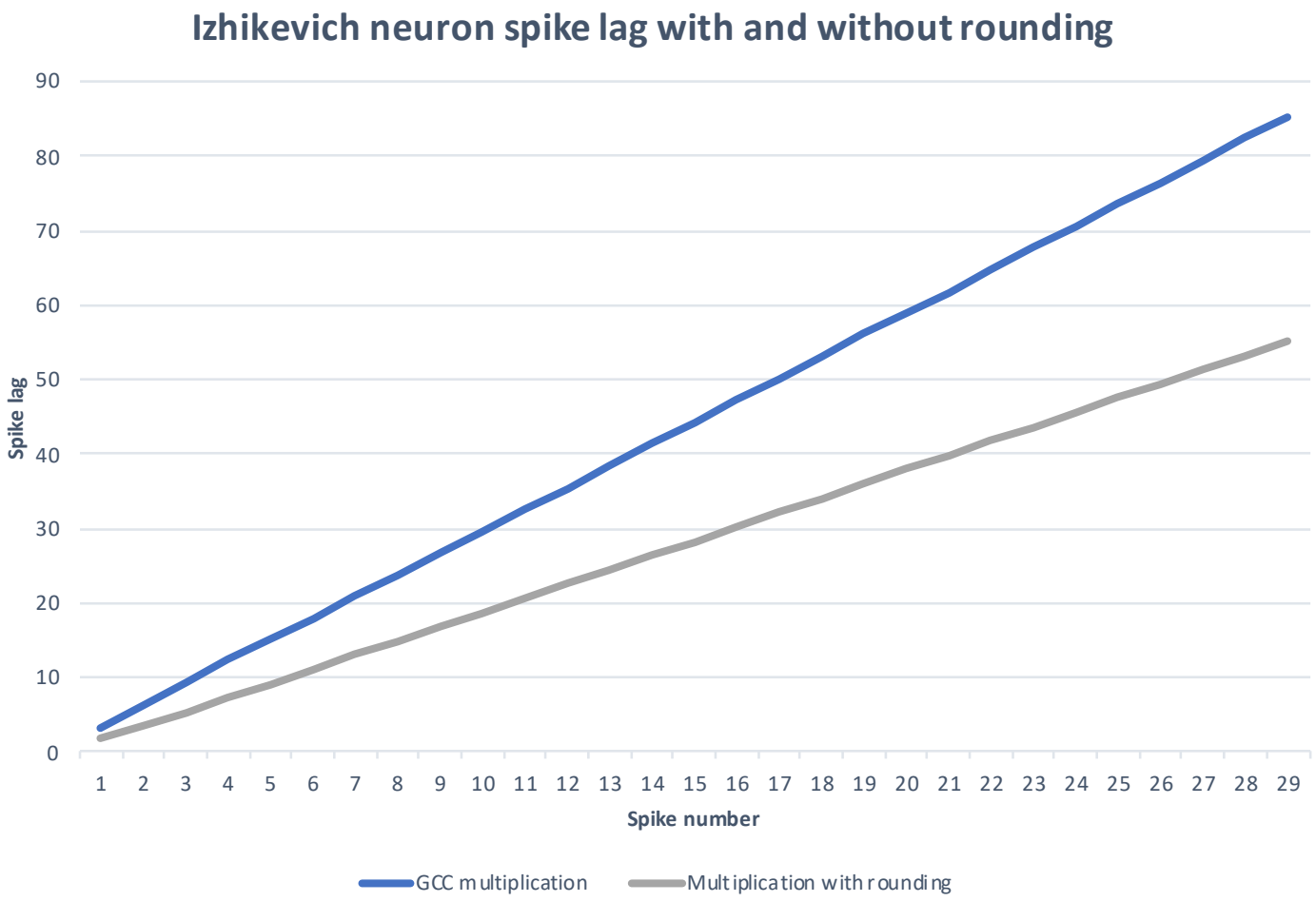


Spike lag in the Izhikevich neuron model (M.Hopkins and S.Furber, Neur. Comp. 2015)



Default SpiNNaker set-up.

Reduced spike lag after adding multiplication with rounding

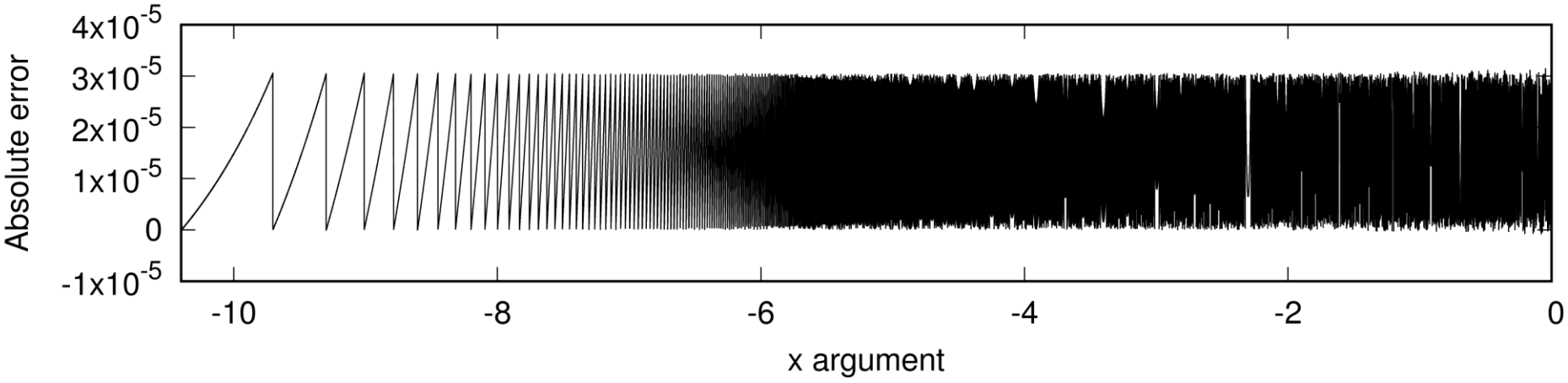


Performance overhead

- Performance of neuron state update routine:
  No rounding (GCC '*' multiplication): **1065ns**
  No rounding SpiNNaker spin_common (__stdfix_smul_k): **2000ns**
  Round to nearest (modified __stdfix_smul_k): **3100ns** (possible to reduce by writing in assembly)
  Float (Data from M. Hopkins and Furber, Neur. Comp. 2015): **6970ns**
- Future work: further improvements are possible for this neuron model by solving some part of the ODE in long fract format.

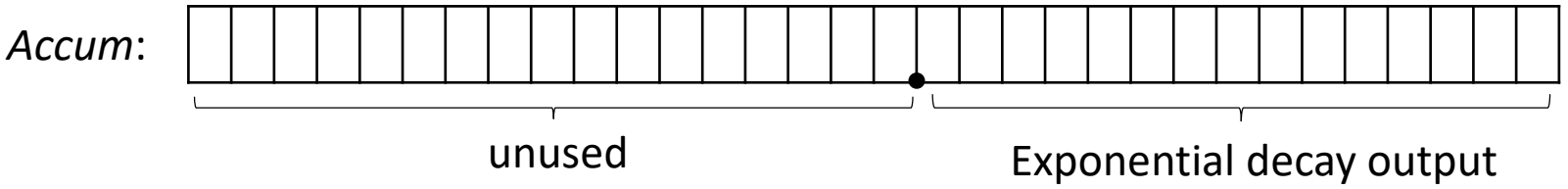
Description of expk() function

- Part of spinn_common package
- Exponential function for *accum* type with the range $x \in [\sim -10.4, \sim 11.1]$.
- If x is out of range, saturate to 0 or the maximum *accum* - ~ 65535.99996948 …
- Absolute error measured in comparison to exp() from the C double precision library
- In the range $x \in [\sim -10.4, 0]$, in steps of $10\epsilon = 0.00030517578125$.
- The maximum error is $0.000031738281250000002 \approx 1.04\epsilon$.
- Similar hardware implementation is available in SpiNNaker2 prototype Santos



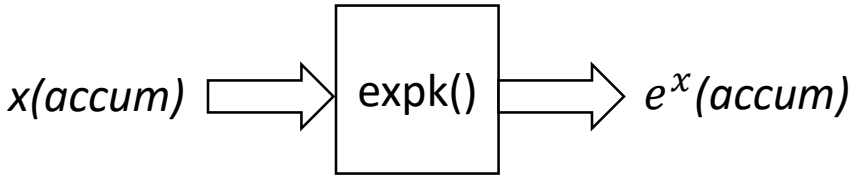
Trick2: mixing *accum* and *long frac*t to improve exponential decay

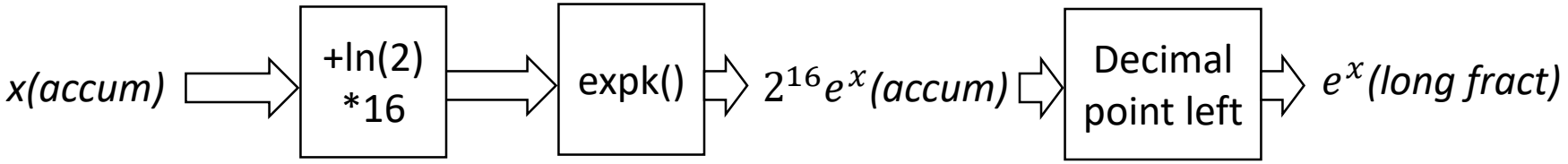
- Exponential decay is described as $X(t) = X_0 e^{-\frac{t}{\tau_x}}$, where X_0 is some initial value to be decayed in time t with the time constant τ_x.
- It is clear that only arguments $x < 0$ are of interest and the output is between $[0,1)$.
- Therefore, the top part of the *acccum* word is unused, *long fract* would be a better type for the output – gives 2x more bits of precision.
- How to use an *accum*-only function (expk() in SpiNNaker, or the accelerator in Santos chip) to get more precision allowed by the *long fract* type?



Trick2: mixing *accum* and *long fract* to improve exponential decay

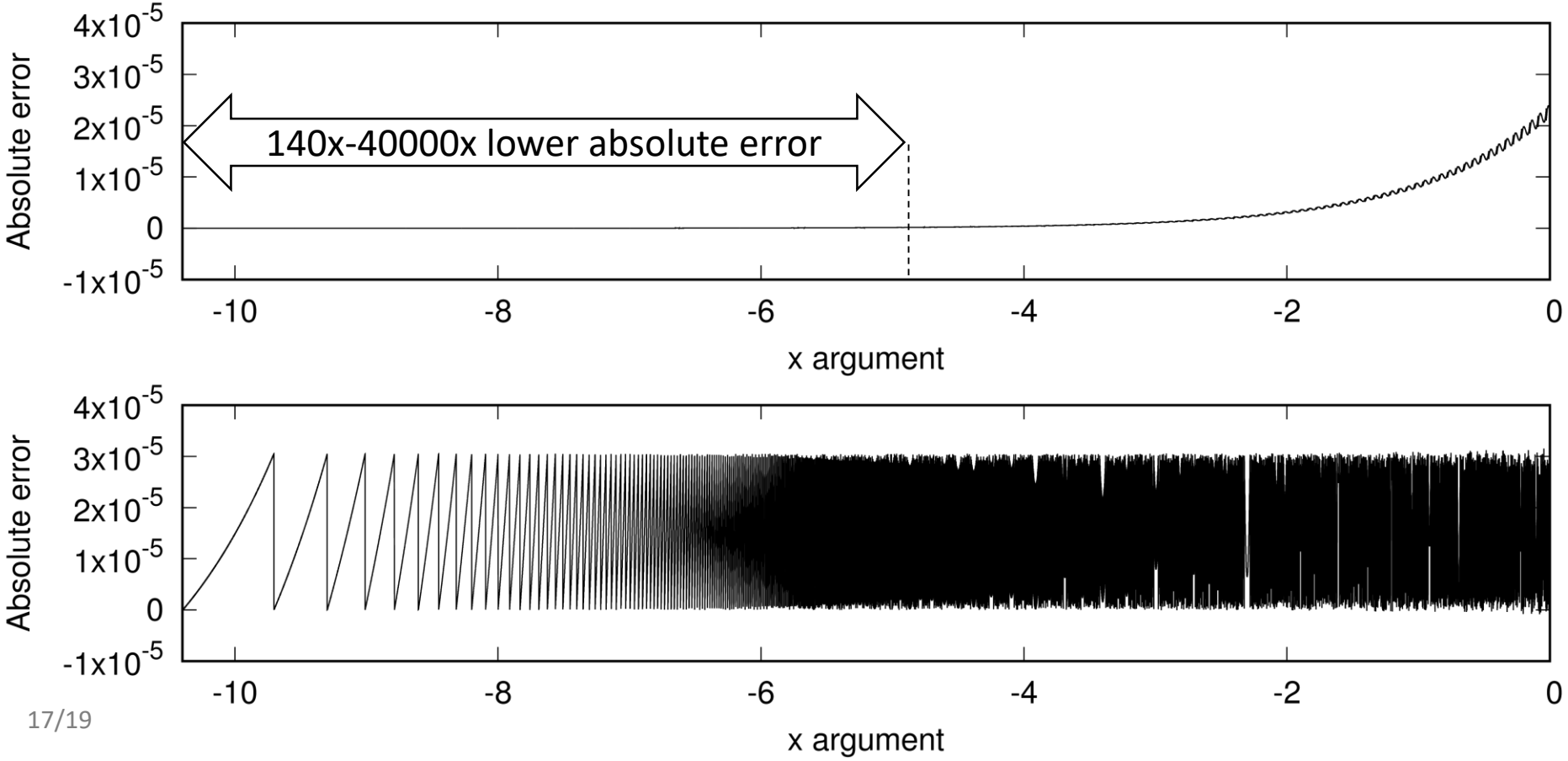
- To get exponential decay as fract, we need to arrive at $2^{16}e^{x}$ (Not by shifting).
- Note the property: $2^{16}e^{x} = e^{\ln(2^{16})}e^{x} = e^{16\times\ln(2)}e^{x} = e^{x+16\times\ln(2)}$.
- Now we have $2^{16}e^{x}$ in *accum* or e^{x} in *long fract*!
- I.e. the input range $x \in [\sim -10.4, 0]$ is transformed to $x \in [\sim -10.4 + \ln(2)\times 16, \ln(2)\times 16]$.
- The output range is transformed to y $\in [0, 2^{16}]$ in *accum* or y $\in [0, 1]$ in *long fract.*





Absolute error of exponential decay when mixing formats

- Max error of $\sim 0.79\epsilon_{accum} \approx 0.000024108886718750 01$
- Average error is lower $\sim$7x



Advantages and disadvantages

Disadvantages:

- Extra cost is 1x addition and some C pointer game to change the decimal point location (Do not simply cast to a *long fract*! Convert through pointers).
- Requires working in *long fract* – try to utilize the bottom bits in a useful way. For example rounding to *accum*, or mixed format multiplications.

Advantages:

- Does not require implementation of another exp function for *long fract*.
- Wider input range of $x \in [\sim -21.4, 0]$ - does not saturate to 0 below -10.4 because more precision is available at the bottom of the word.
- Significantly lower error in most of the function domain.

Summary

- Two simple methods of improved accuracy of fixed-point arithmetic have been shown
- First trick works for all multiplications, not just accum.
- Exponential function trick only works for e^x - for other types of functions other properties would have to be found
- It is an example of only a small fraction of what is available if some code tidiness can be sacrificed

- Next presentation on 18th December: Floating point exponential and logarithm hardware accelerator

Questions?