Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

Spring Semester 2022

Design of a Floating-Point Stochastic Rounding Unit

Semester Project

Jiayong Li
jiayli@student.ethz.ch
Enci Zhang
zhange@student.ethz.ch

June 2022

Supervisors:	Luca Bertaccini, lbertaccini@iis.ee.ethz.ch
	Gianna Paulin, pauling@iis.ee.ethz.ch
	Tim Fischer, fischeti@iis.ee.ethz.ch
Professor:	Prof. Dr. Luca Benini, lbenini@iis.ee.ethz.ch

Acknowledgements

First, we would like to thank our Ph.D. supervisors, Luca Bertaccini, Gianna Paulin, and Tim Fischer, for their help, support and guidance. In addition, our senior scientist at ETH, Frank Kagan Gürkaynak gave some insightful inputs during the design review. The course units VLSI I and II arm us with relevant knowledge and guarantee a quick start for our project. We also want to thank our principal supervisor, Prof. Dr. Luca Benini, and the IIS group for providing this wonderful opportunity, during which we could get a glimpse of digital integrated circuit design. And finally, we direct our gratefulness to ETH Zürich, for bestowing us with first-class education and academic atmosphere.

Abstract

Deep Neural Networks (DNNs) have been used on a wide range of problems in our life. But training such an algorithm requires a lot of memory and energy. To save memory and be more energy-efficient, academia and industry have started to re-examine the required precision in DNNs by using limited-precision data representations. With this decrease in bitwidth, stochastic rounding is brought forward as the counteraction against accuracy degradation. Benefiting from its probability-based characteristics, stochastic rounding does not suffer from stagnation. As a result, it works better than other rounding methods when putting a small adjustment on a relative big value, which is the typical case in training DNNs. Studies of low-precision neural network training have proven accuracy enhancement when substituting the widely-used RNE with stochastic rounding. Therefore, in this project, we extend a former FPU to support stochastic rounding. The performance of this extension is compared against RNE round mode in three-operand additions and sum of dot products accumulating in a larger formats.

Declaration of Originality

Jiayong Li,
Enci Zhang,
Zurich, June 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Context and Motivation

Deep Neural Networks (DNNs) have achieved great success over a wide range of problems in the past decade. However, these models are typically very big and training a DNN is power-hungry. Since the Neural Networks (NNs) are resilient to precision losses, one approach to reduce DNN's memory footprint and achieve a more energy-efficient training process is to utilize floating-point(FP) formats with reduced precision. Under this context, Stochastic rounding(SR) is introduced as a novel floating-point rounding method that aims to mitigate the degradation of model accuracy due to sacrificing the accuracy of arithmetic operations.

Stochastic Rounding is a rounding method that randomly maps a real floating-point number to one of its two nearest values following a certain probability regime. Unlike the commonly-used RNE mode, which rounds to the nearest value and settles ties to even, stochastic rounding does not suffer from stagnation and consequently outperforms RNE during additions where the two addends have a significant difference. This merit of SR has been investigated and explored by various studies[1] [2][3] in neural network training with low-precision floating-point formats. When employing stochastic rounding over the RNE mode, all these works show accuracy enhancement in respective machine learning tasks.

In our project, we implement a stochastic rounding extension for the sum-dot-product(SDOTP) module of an existing floating-point unit(FPU), fpnew [4], which supports trans-precision computations with instructions contained in the -F and -D RISC-V instruction set architecture (ISA) extensions. Then, we adapt the rounding unit to correctly interface with the SDOTP module and exhaustively test the functionality and performance of our stochastic rounding mode. Finally, we integrate the new FPU into a microcontroller architecture–PULPissimo [5], an energy-efficient, single-core platform developed by IIS.

1.2. Related Work

Since its proposal in 1992 by Hohfeld and Fahlman[6], Stochastic Rounding has been widely applied to save hardware resources by enabling low-bitwidth Neural Network training. Gupta *et al.*[7] prove that with SR, a 16-bit fixed-point data representation incurs little or no degradation in classification accuracy. Su *et al.*[2] delve further into this path and achieve 8-bit fixed-point representation leveraging SR. Zamirai *et al.* focus on 16-bit floating-point FPU training and utilize SR to avoid RNE stagnation. The authors obtain equivalent validation accuracy to that of 32-bit across seven deep learning tasks. Na *et al.*[8] implement stochastic rounding hardware targeting limited-numerical-precision recurrent neural network training. Their results show that SR combined with dynamic fixed-point boosts hardware speed by 4.7x and decreases energy per task by 4.55x in comparison to floating-point. As we see, former studies rely on software libraries to emulate behaviors of stochastic rounding. Up to now, the few chips released, including Intel Loihi and the new Graphcore IPU, are closed-source. Our project makes the first attempt to realize open-source floating-point arithmetic hardware supporting stochastic rounding.

1.3. Our Contributions

1.3.1. Implementation of the Stochastic Rounding Extension

During implementation, we develop the Round by Stochastic Rounding (RSR) mode, add new parameters to increase its controllability, and adapt the rounding unit to the SDOTP module.

1.3.2. Functional Verification

Regarding functional verification, we first build our own golden model in Python for sanity checks. Additionally, we adopt another existing golden model implemented in Julia[9]. Then, we verify the hardware results leveraging these two models.

1.3.3. Performance Assessment

To assess RSR performance, we go through HDL simulations to test the newly developed SDOTP unit, evaluate its performance against the original SDOTP unit, and compare the effectiveness of the RSR mode with other rounding modes, especially RNE.

1.3.4. Tapeout Adaptations

In this part of the project, we first integrate our FPU with stochastic rounding extension to the PULPissimo microcontroller, then adapt back-end scripts for the Eclipse chip tapeout to TSMC65 technology. Our tapeout contributions include RTL integration, synthesis, regression tests (C test), and the physical design (Floorplan, place and route, signoff).

Chapter 2

Algorithm

2.1. Existing Stochastic Rounding Algorithms

Stochastic rounding (SR) is a rounding method that randomly maps a higher precision real number x to one of the two nearest values in a lower and finite precision system. The probability of rounding to either of these values is inversely proportional to the their distance to x, shown as follows[10]:

$$f(x) = \begin{cases} \lceil x \rceil, & \text{with probability q(x)} \\ \lfloor x \rfloor, & \text{with probability 1 - q(x)} \end{cases}$$

where $q(x) = \frac{x - \lceil x \rceil}{\lceil x \rceil - \lfloor x \rfloor}$. Thus, by encoding the discarded bits into this probability, the expectation of the rounding result $f(x)$ equals to x, namely, $E[f(x)] = x$.

One approach to implement the SR algorithm is to generate a random number in the same position as the bits to be rounded, add it to the original value, then round the sum toward zero. This method is applied in the Julia[9] and QPyTorch[11] implementation.

Another approach is shown in Ref [10]. In this method, we again generate a random number, but compare it to the original value instead of addition, then according to the comparison outcome, we choose either to round up or down. As comparisons are less area- and time-consuming than additions in the hardware perspective, this second approach is clearly a better choice for hardware implementation.

2.2. Our Hardware SR Algorithm

As mentioned in previous section 2.1, we follow the second approach to implement our SR algorithm. This rounding algorithm takes raw results in IEEE-754 format from previous calculations, round them by SR, and outputs the rounded results in our target format.

The most relevant part of the input is the mantissa bits. For simplicity, we assume the exponent part is correctly biased and the post-rounding checks are omitted here. We assume the input mantissa is N-bit long, and it needs to be rounded to m bits (apparently, $m < N$). The extra $(N - m)$ bits would be the information for us to perform rounding. A parameter introduced here is the "precision of SR", p, because it's not necessary to take all the extra bits to perform the rounding and we will use the first p bits in the extra bits (we call them rounding_bits) to help us do the rounding.

To determine which direction (round toward/away from zero, RTZ/RA) to round, we compare this rounding_bits with a p-bit random number. The random number should be evenly distributed in range $[0, 2^p - 1]$. If this random number is smaller, we need to round away from zero and otherwise round toward zero. Intuitively, if the rounding_bits is big, it's easier to satisfy the RA condition which means the absolute value of the rounded number will likely be bigger than the original one since the original number is closer to it. And this make sure the expectation value of the rounding results equals the value before rounding. This algorithm is shown here 1.

Algorithm 1: stochastic rounding

```
input      : Pre-round result consists of sign, correctly biased exponent and
             mantissa bits before round(implicit bit has already been discarded
             before this step): {pre_round_sign, pre_round_exp,
             pre_round_man}
output     : Rounded result: {rounded_sign, rounded_abs}
parameters: m, number of mantissa bits in the rounded results
             p, precision for SR, number of bits in matissa used for SR

1  pre_round_abs ← {pre_round_exp, pre_round_man[N -: m]};
   /* Use the next p bits from mantissa to determine roundup = 0 or 1.  If
      p is set too large that there's not enough bits in the mantissa, we
      will pad the LSBs of rounding_bits to be 0s to make sure the
      implementation still works.  (Not shown here for simplicity)      */
2  rounding_bits ← pre_round_man[N-m -: p];
3  random_number ← Random_gen(length = p);
4  if random_number < rounding_bits then
       // random number is smaller, we round up
5      roundup = 1;
6  else
       // random number is equal or bigger, we round down
7      roundup = 0;
8  end
9  rounded_sign ← pre_round_sign;
10 rounded_abs ← pre_round_abs + roundup;
```

Chapter 3

Hardware Architecture

3.1. Basic Setup

For the first step, we extend the five rounding modes specified in RISC-V ISA with our new rounding mode, round by stochastic rounding (RSR), and implement our rounding algorithm 1 in the rounding unit from fpnew. All rounding modes support by the new rounding unit are shown in table 3.1.

Table 3.1.: Rounding modes supported by the new rounding unit

Round Mode	**Mnemonic**	**Source**	**Meaning**
000	RNE	RISC-V	Round to Nearest, ties to Even
001	RTZ	RISC-V	Round towards Zero
010	RDN	RISC-V	Round Down (towards $-\infty$)
011	RUP	RISC-V	Round Up (towards $+\infty$)
100	RMM	RISC-V	Round to Nearest, ties to Max Magnitude
101	RSR	Extension	Round by Stochastic Rounding
others	—	—	*invalid*

3.2. Implementation and Parameters

After implementing of our algorithm in the rounding unit, we integrate the rounding unit to the SDOTP module in the FPU. In the old SDOTP module, the rounding unit does not accept the whole pre-round FP number, the extra bits are first compressed to so called round-sticky bits before being passed to the rounding unit. Round-sticky bits indicate the position of the un-rounded value related to the two closest FP numbers: closer to the smaller one, closer to the bigger one or exactly in the middle. Depending on the rounding mode, we make different rounding decisions ($roundup = 0$ or 1). The output of the rounding unit is the rounded results.

In the new rounding mode, RSR, the rounding decision are made by the comparison result between the rounding_bits and a random number1, so we add a new input port to accept this input. Therefore, the interface between the rounding unit and the SDOTP module also needs to be modified to adapt this change.

Our final rounding unit and its interface in the SDOTP module are shown in Figure 3.1. The dashed lines indicate the old modules and the colored areas with solid lines are our modifications. The final mantissa after the calculation is $2p\_dst + p\_src + 4$ bits long, where p_dst and p_src stand for the total bits in the mantissa including the implicit bit for the destination and source FP formats. The first bit (MSB) is the implicit bit and will not appear in the final result. Start from the second bit, the next $p\_dst - 1$ bits are the mantissa bits before rounding. The rest part of the result will be rounded.

We add a new slice for the RSR mode. This step introduce two new parameters, p_rsr and rsr_extra_bits in the hardware. As mentioned before, p_rsr is the precision for SR, which means the number of bits that we use to perform SR. rsr_extra_bits will add extra bits in in the final mantissa and previous results. This parameter is set just in case we need a more precise final mantissa for SR. We already have a long mantissa compared to the final output and this parameter will cause a big area overhead for the SDOTP unit since it increases the size of every intermediate results. And most importantly, in the later chapters we will show $p\_rsr = 12$ is enough for our applications. So it will be set to 0 by default and this parameter is not shown in the figure.

3.3. Pseudo-random Number Generator

The pseudo random numbers in the algorithm is generated by a linear feedback shift register(LFSR). We choose the LFSR from the common cells for PULP platform. It has two parameters: $LfsrWidth$ and $CipherLayers$. $LfsrWidth$ is the internal length of the LFSR. Once you have chosen the p_rsr, the output width of the LFSR is fixed, but you still have the freedom to choose the internal length. This parameter can be set from p_rsr to 64. $CipherLayers$ is the number of cipher layers you want to use at the output. The cipher layers are from PRESENT [12], and are used here to break the shift pattern of the LFSR. The cipher layers are only available when the internal length of the LFSR is 64 bits. To save power on the chip, the LFSR is only enabled when the round mode is RSR. The state of LFSR will not change when you are not using SDOTP unit with RSR mode.

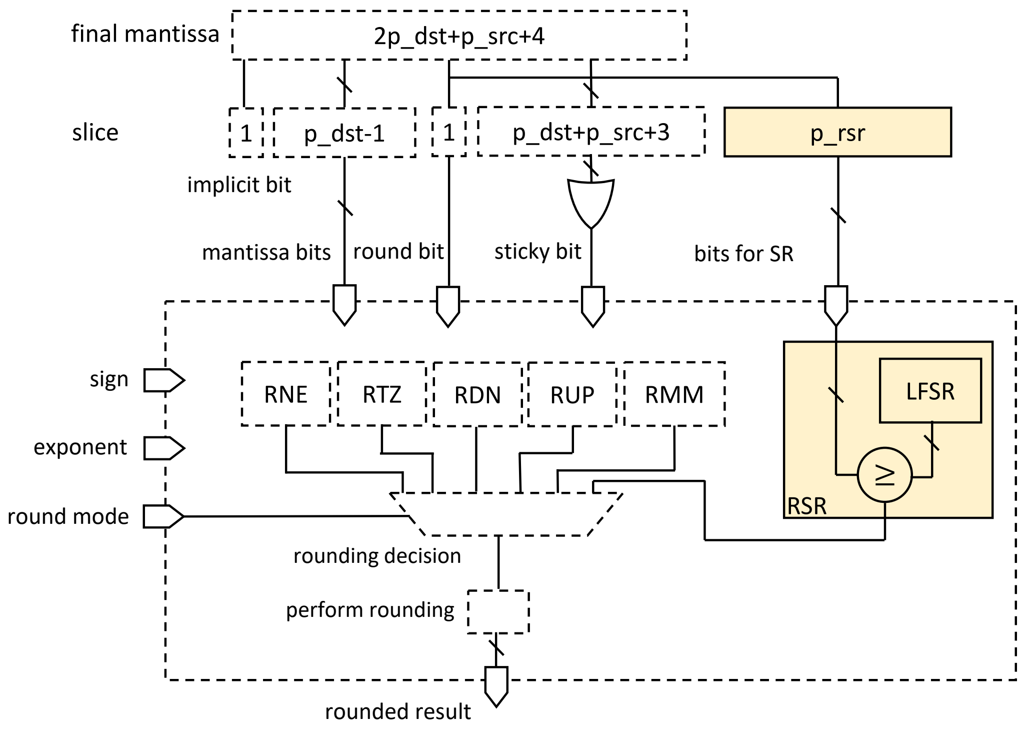

Figure 3.1.: Hardware architecture of the rounding unit and its interface with SDOTP module

Chapter 4

Verification and Results

4.1. Possible Configurations in the Hardware and Cases Tested

There are several things that may influence the final results and we need to test each combination of them to get the hardware architecture with the best results. Since the SDOTP module supports different instructions and is capable of trans-precision computing, it's obvious we need to test different instructions and go through the possible precision settings it supports. In the test, the two configuration parameters of the SDOTP module are set to: $SrcDotpFpFmtConfig = 6'b001\_111, DstDotpFpFmtConfig = 6'b101\_111$, which means the inputs can be set to any formats shorter or equal to FP16 and output can be set to different formats shorter or equal to FP32. We tested the VSUM and SDOTP instructions. For the two operations, we tested different possible input and output formats from FP8 to FP32.

Since this hardware is for general usage, we set up three different possible input distributions to test the performance of hardware in different situations: standard Gaussian distribution, $N(0, 1)$, uniform distribution in region $(-1, 1)$, $U(-1, 1)$, and uniform distribution in region $[0, 1)$, $U(0, 1)$. For the VSUM instruction, there are 3 different precision settings and we test each of them in these 3 different input distributions. For the SDOTP operation, there are 2 different precision settings and we tested both of them in the first two input distributions. Table 4.1 shows the different cases we have tested.

4.2. Input Generation

As mentioned in the previous section, we use values sampled from different distributions as stimuli to the hardware. The input data is generated in the following steps. First,

Table 4.1.: Different cases tested

Instruction	Input distribution	Input FP format	Output FP format
VSUM	$N(0,1), U(-1,1), U(0,1)$	FP8 FP16 FP32	FP8 FP16 FP32
SDOTP	$N(0,1), U(-1,1)$	FP8 FP16	FP16 FP32

generate the random inputs in the target distribution by Python's package, random. You will get the random number in FP64, Python's default FP format. Then, convert it to it's binary format (a string consisting of '0' and '1'), bias the exponent in the target format and cut all the extra bits in mantissa(or in other words, round the FP64 number to your target format by RTZ). The generated inputs are shown in Figure 4.1 and 4.2.

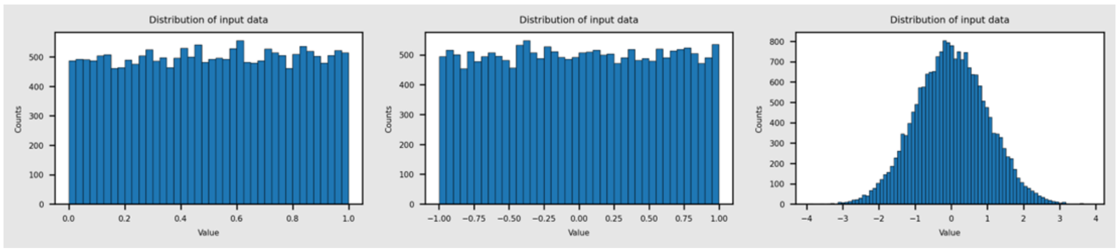

Figure 4.1.: Input distributions of FP16. Left to right: $U(0,1), U(-1,1), N(0,1)$.

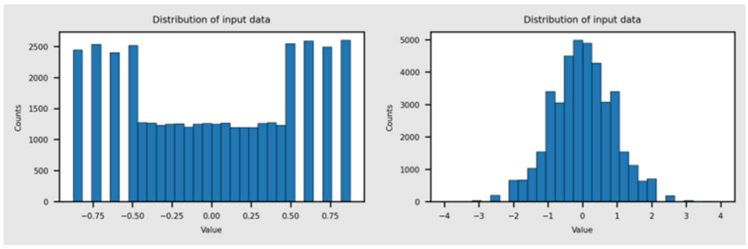

Figure 4.2.: Input distributions of FP8. Left to right: $U(-1,1), N(0,1)$

4.3. Golden models

To assess the performance of SR, we introduce two software golden models for low-precision FP formats and SR. The first model in Python is our novel contribution, the other is an open-source model written in Julia.

4.3.1. In Python

To provide a reference for our hardware implementation and get a more direct idea of the SR performance, we build our own golden model in Python. We integrate useful functions to this model, like format converters that enable the conversion between floating-point and its corresponding binary string. It is used to translate between human-readable FP values and input/output of the hardware obtained from the testbench, and also some software simulation to give us an idea of what we should expect from RSR.

We generate 10,000 random FP32 values sampled from a uniform distribution between 0 and 1, then in each step, we accumulate a fresh input to the current sum (sum is initialized to 0), and round the newly acquired sum to the target precision, FP8 or FP16. During simulation, we repeat this process for several working regimes, including FP32, which serves as the ground truth. In addition, we test the following four combinations of formats and rounding modes: FP8RSR, FP8RNE, FP16RSR, and FP16RNE. This operation is similar to the process of VSUM in the FPU, where three numbers are added instead of two. Results are shown in 4.3, and offer intuition on RSR and RNE behaviors.

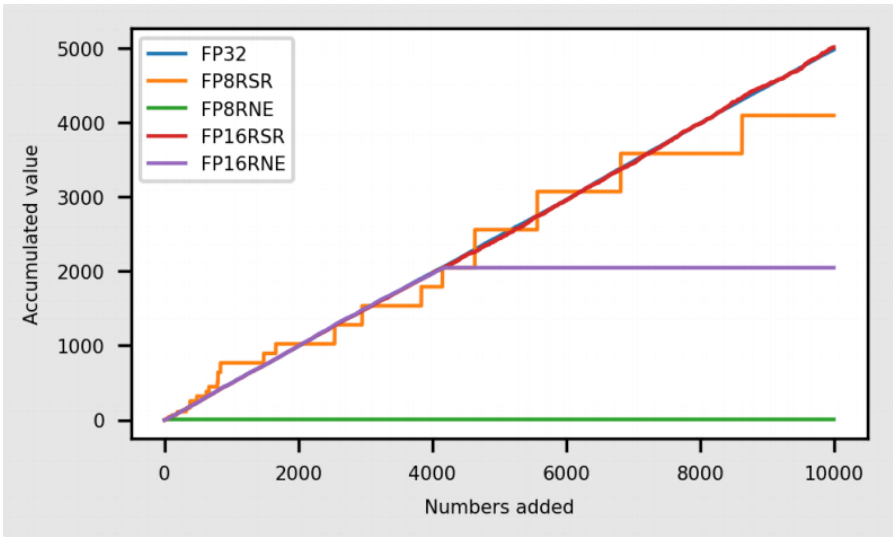

Figure 4.3.: Cross comparison between various operation modes in Python

As we can see, RNE for FP8 and FP16 both saturate after a certain point (FP8 at 8 and FP16 at 2046), but FP8 has a much lower tolerance of sum-to-addend difference. In other words, when the difference between the accumulated sum and the addend becomes too significant, RNE can no longer perceive the small addend, and thus, the sum saturates at a fixed value. As FP8 has a smaller number of mantissa bits, its saturation point naturally comes much earlier than FP16. RSR, on the other hand, maintains its accuracy to some extent thanks to its probability characteristics. It does not have any saturation point and keeps up with the accumulation result. In the FP8 case, RSR has an obvious better performance compared to RNE. Due to the highly-quantized nature of FP8, we can see clearly its stepped fluctuations around ground truth. For FP16, RSR behavior is almost indiscernible from the ground truth.

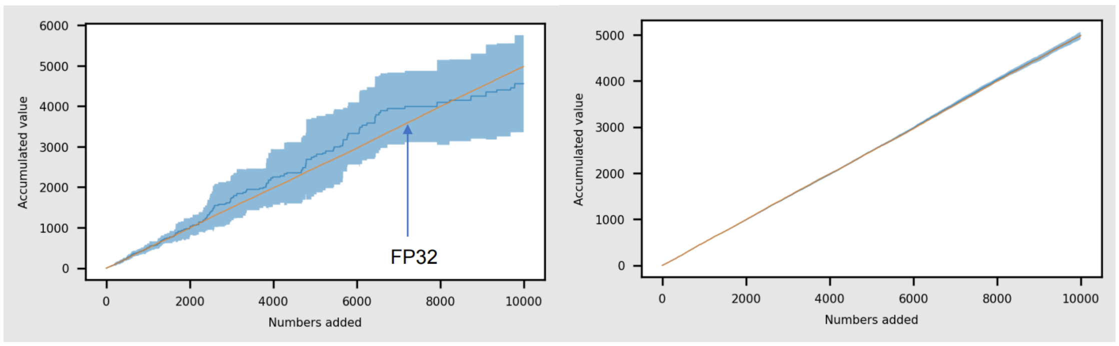

Figure 4.4.: mean and variance of RSR over 10 runs, left: FP8, right: FP16

In 4.4, we observe enhanced performances for both FP8 and FP16 when RSR results are averaged over 10 runs. This behavior is also dominated by the probability property of stochastic rounding, as discussed in 2.1, its mean value closes to ground truth after more runs.

4.3.2. In Julia

An open source implementation of stochastic rounding written in Julia [9] is also taken as a golden model in this project. We first use Python to generate multiple lists of random input with different precision and probability distribution. To keep maximum consistency with results in hardware simulation, we apply the same inputs as in 4.2, and assume the exact process as in hardware. To be specific, we dot multiply adjacent values, accumulate the product, and round the sum with SR. Finally, we evaluate the performance of SR by examining the accumulated absolute value of the error. This overall process is described in Algorithm 2. In addition, we also check that the two software models in Python and Julia give the same results.

As the previous golden model in Python already offers intuition on SR performance, it is sufficient to only do sanity checks on the VSUM mode with selective input arrays using this Julia model. Representative results are shown in 4.5, please refer to Appendix A for more data.

We can see that when accumulating floating point numbers with possibility to be either positive or negative, absolute error of RSR is much farther away from zero compared to RNE, and RNE does not saturate. This behavior of RSR and RNE matches hardware simulation results in Section 4.4.1.

Algorithm 2: Golden model

```
input     : Generated random arrays $a$.
            Precision formats: FP8, FP16, FP32.
            Probability distribution: Gaussian distribution $N(0,1)$, uniform
            distribution $U(0,1)$, uniform distribution $U(-1,1)$.
output    : Step-wise accumulation result $S$ with target precision, accurate
            accumulation result $R$ with FP64 precision, absolute errors $abs\_err$,
            average of RSR absolute errors across $n$ runs $avg\_abs\_err$.
Parameter: Number of input values $N$, number of RSR runs $n$.
1 Convert the lower precision array $a$ to a FP64 array $b$.
2 SDOTP: FP8 $\rightarrow$ FP16, FP16 $\rightarrow$ FP32.
    $N = 40000$, $S[1] = 0$,
    $S[i+1] = a[4i-3] * a[4i-2] + a[4i-1] * a[4i] + S[i]$, for $i \in (1, 10000)$, rounded to
    target precision by RSR or RNE.
    $R[i+1] = b[4i-3] * b[4i-2] + b[4i-1] * b[4i] + R[i]$, for $i \in (1, 10000)$.
3 VSUM: FP8 $\rightarrow$ FP8, FP16 $\rightarrow$ FP16, FP32 $\rightarrow$ FP32.
    $N = 20000$, $S[1] = 0$,
    $S[i+1] = a[2i-1] + a[2i] + S[i]$, for $i \in (1, 10000)$, rounded to target precision by
    RSR or RNE.
    $R[i+1] = b[2i-1] + b[2i] + R[i]$, for $i \in (1, 10000)$.
4 $abs\_err[i] = |R[i] - S[i]|$, for both RNE and RSR.
5 Exclusively for RSR, run step 2/3 and 4 for $n$ times, usually $n = 1, 10, 100$ or 1000,
    take the average across $abs\_err[i]$ to get $avg\_abs\_err[i]$.
```

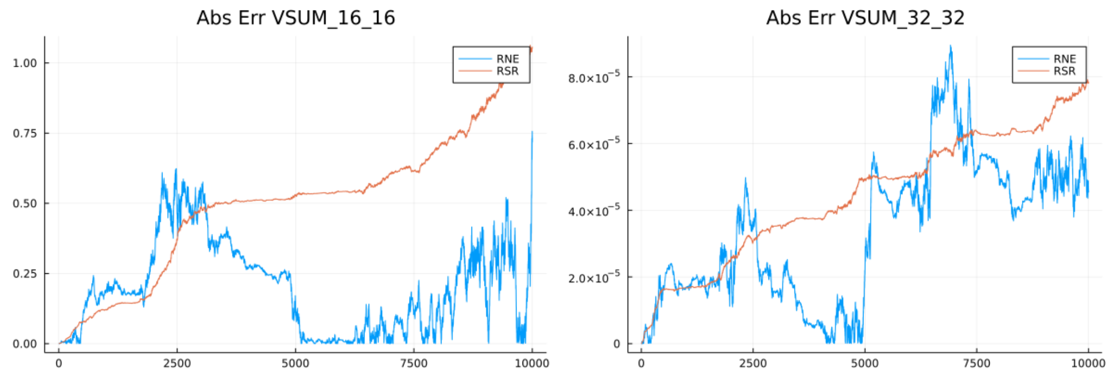

Figure 4.5.: absolute error of RSR and RNE in Julia, left: VSUM FP16 $\rightarrow$ FP16 with $U(-1,1)$ input, right: VSUM FP32 $\rightarrow$ FP32 with $N(0,1)$ input

4.4. Functional Verification of SR

Figure 4.6 illustrates the testbench setup for the functional verification of SR. After testing our parameters' compatibility with the old SDOTP module and SR unit's proper functionality, we assess the performance of SR in different cases in Table 4.1. First, we generate random inputs as described in Section 4.2, then feed them (FP8 or FP16 representable numbers) into the golden model to receive benchmark results in double precision(FP64). To test the hardware functionality, we need to convert the generated inputs to binary formats, execute the SDOTP operations, output the results, and back-convert the results to IEEE FP format. Similar to the algorithm described in 2, our testbench wrapper for SDOTP works as follows: *op* determines the operation mode, VSUM or SDOTP. If *op* is SDOTP, $is\_first = 1$ at the start of our testing, operand e takes 0 as input, operands a, b, c, d read in the first four values of the input file. After arithmetic calculations of SDOTP, specifically, $a * b + c * d + e$, the result is temporarily stored inside a flip-flop, fed back to the multiplexer as the next input for operand e, and outputted in the next clock cycle. As the process advances, operands a, b, c, d continue to read in from the input file until reaching the end, and in this way, we implement the accumulation of dot products. If *op* is VSUM, we set operands b and d to 1, and simply read in two inputs at a time for operand a and c, b and d will be bypassed and $a + c + e$ is computed.

As previous sections 3.2 and 3.3 introduced, there are four parameters inside the SDOTP module: extra bits in the intermediate results *rsr_extra_bits*, which is 0 by default, RSR precision p_rsr, LFSR internal width $LfsrWidth$, and number of cipherLayers $CipherLayers$.

In this chapter, we will show the results in different implementations and find the best p_rsr, $LfsrWidth$ and $CipherLayers$ for different input and output formats and operations.

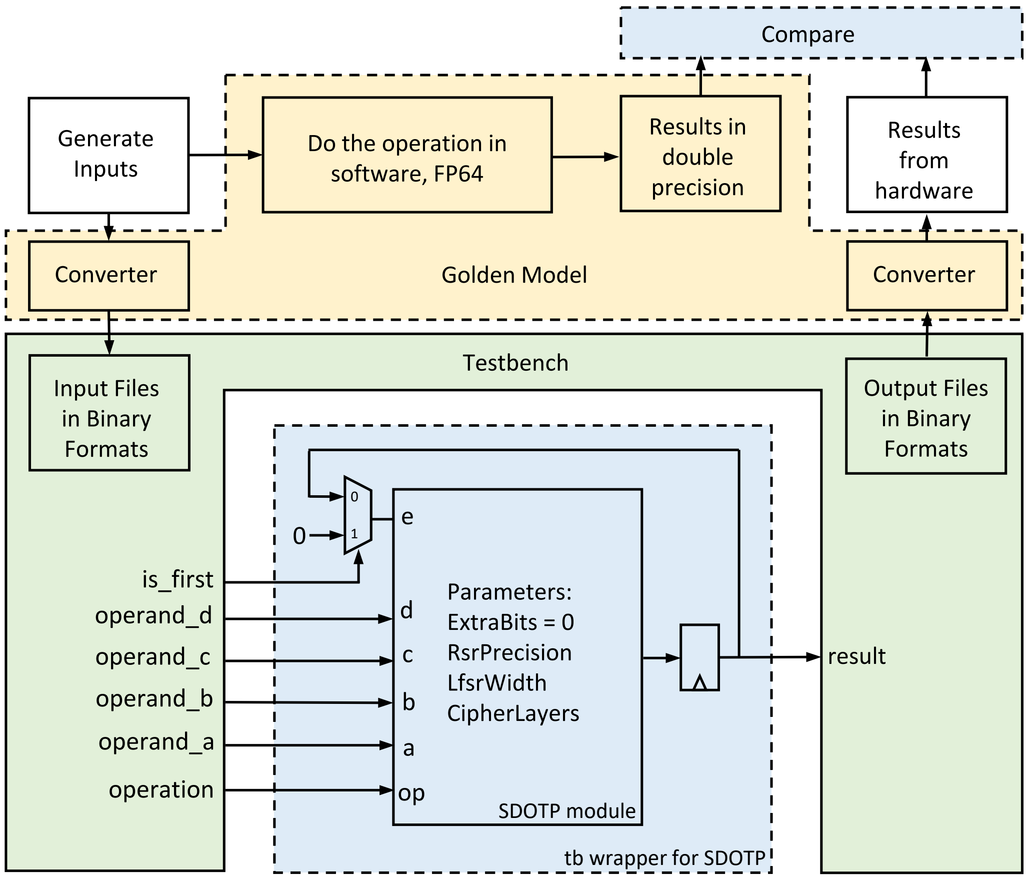

Figure 4.6.: Functional verification setup.

4.4.1. VSUM, FP16 Input, FP16 Output

The test started with the format in between, FP16. This format is not super short (FP8) or very precise (FP32). We will show all the detailed results in this section.

4.4.1.1. $U(0,1)$

For the first step, we set some reasonable values to the parameters and compare RSR with RNE, p_rsr is set to 12 bits, and we use 64-bit LFSR with 3 cipherlayers to break the shift pattern.

Figure 4.7 shows the results when input is a uniform distribution in [0,1). Top left corner shows the accumulated value in 10000 consecutive VSUM operations obtained from software golden model in FP64 (blue line), hardware simulation with different rounding

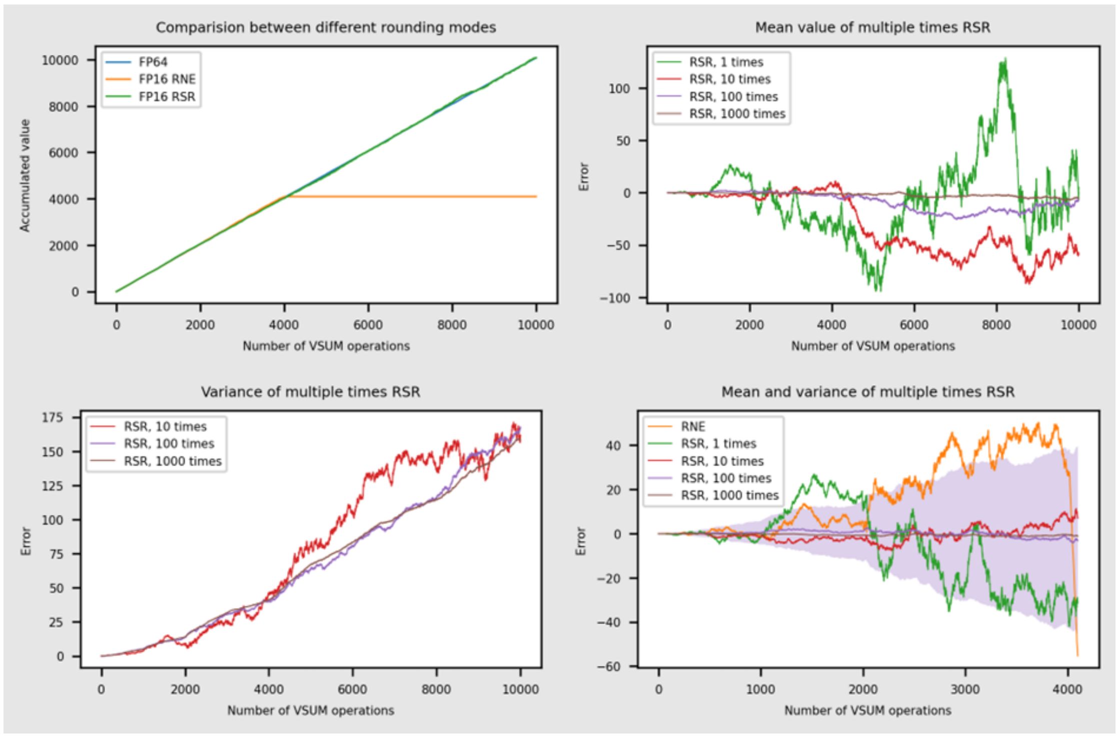

Figure 4.7.: Results of VSUM, $U(0,1)$ iutput. Top left: accumulated value over 10000 VSUM. Top right: average error of RSR compared to FP64. Bottom left: standard deviation of error. Bottom right: RSR's error disrtibution compared to RNE mode before it stagnated.

modes, RNE(yellow line) and RSR(blue line). In this input distribution, the adds are in only one direction, the RNE results get saturated after it reaches 4096. This result is consistent with our previous expectations: when you add a big number to a small number in RNE mode, the big number will swamp the small number and there would be no changes in the result. For the RSR result, since it's mechanism is diffemt from RNE, it will not stagnate but wiggle around the FP64 value.

Due to RSR's random nature, every time we will obtain a different trace of the accumulated value. We calculated the average value and standard deviation of the results obtained by multiple runs of RSR after the same number of operations (which means LFSR starts at different state in hardware simulation). The top right plot shows the average error compared to the FP64 results and the bottom left plot shows the standard deviation of the error. As we can see, the average error is significantly smaller if we average the results more than 100 times. The standard deviation of the error will not change if we average over multiple runs.

The bottom right plot in Figure 4.7 compares those errors with the error of RNE mode before RNE stagnated. The shaded area corresponds to the average value plus/minus standard deviation when we do the statistics over 100 times RSR results. Although the average value of RSR is quite close to the FP64 results, it is not better than the RNE modes if we have a look at it's distributions. But in the region where RNE stagnated (the straight line in the rightmost part in the plot), the error of the RNE mode explodes and RSR is definitely better than RNE.

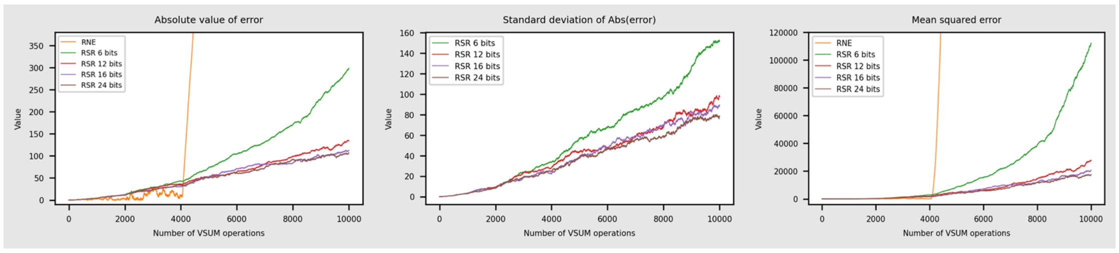

Figure 4.8.: Results of varying p_rsr. Left to right: average of abs(error) of RNE and different precision's RSR, standard deviation of abs(error), mean squared error

To analyze the error in detail and find the best p_rsr, we also calculate the mean and standard deviation for error's absolute value, and also the mean squared error. The results are shown in Figure 4.8. This shows clearly RSR has a bigger error compared to RNE mode in the "normal" region of RNE. And for the p_rsr, 6 bits has bigger error and standard deviation compared to 12 bits or more and there's no difference between 12 bits, 16 bits and 24 bits. Therefore, to save the area in the hardware, we will use $p\_rsr = 12$ for this case.

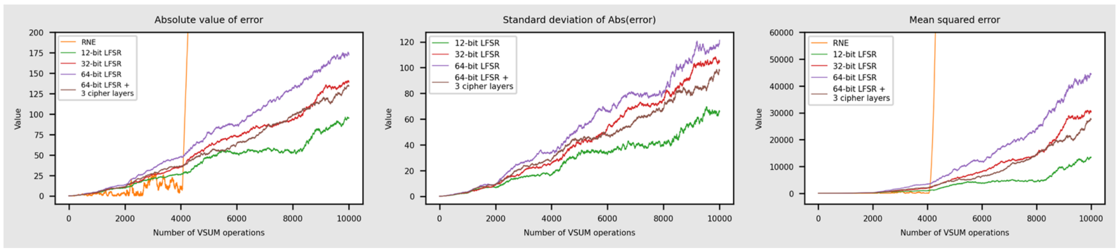

Figure 4.9.: Results of varying LFSR config. Left to right: average of abs(error) of RNE and RSR, standard deviation of abs(error), mean squared error

The next step is to find the influence of different configurations in the LFSR. So far we used a complicated config: 64 bits internal length and 3 cipherlayers to break the shift pattern. We fix $p\_rsr = 12$, tested results from 12-bit, 32-bit, 64-bit LFSR and 64-bit LFSR with 3 cipherlayers. In this case, the 12-bit LFSR has the best performance: it has the smallest error and standard deviation. It's quite interesting that the complicate configured LFSR doesn't improve the performance of the rounding results.

To give a more intuitive picture of this, we examined the output of the LFSR in different configurations. We fix the MSB of LFSR output correspond to 2^{-1}, so the output can be mapped to a number in region [0,1). We calculated the average value of the first 1 million outputs from the LFSRs, the results are shown in Figure 4.10.

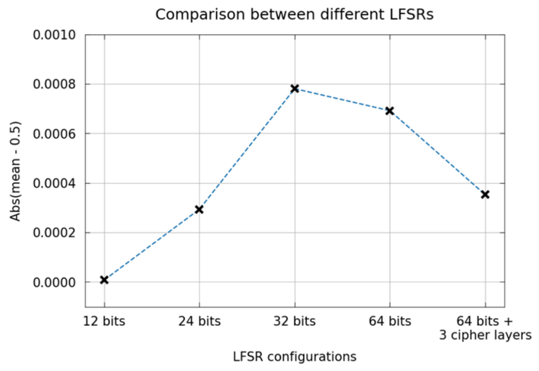

Figure 4.10.: Average value of the first 1 million outputs from the LFSR, values are obtained by setting the MSB of LFSR output to 2^{-1}

The results show the 12-bit LFSR has the output averaged closest to 0.5. The 12-bit LFSR has go through all of its possible states ($2^{12} - 1 = 4095$) in this process, so it's more "uniform". And in the SR applications we only need a uniformly distributed random number, that's why the simplest LFSR has the best performance.

4.4.1.2. U(-1,1) and N(0,1)

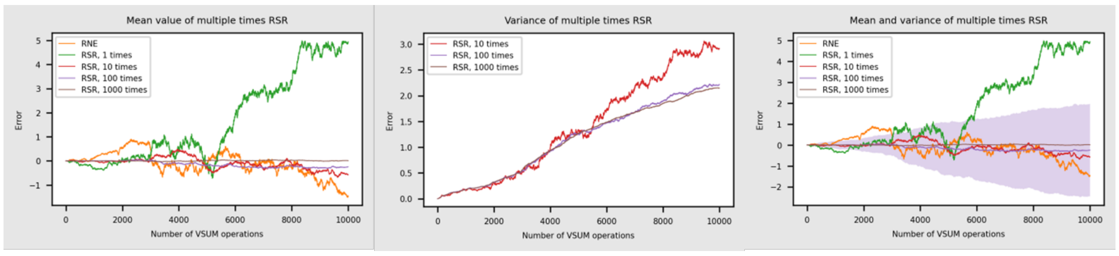

Figure 4.11.: Results of VSUM, $U(-1,1)$ input. Left to right: average error of RSR compared to FP64, standard deviation of error, RSR's error disrtibution compared to RNE.

Figure 4.11 shows the results when input is a uniform distribution in (-1,1). The results are similar to the $U(0,1)$ distribution. For multiple times RSR, the average error is smaller than RNE when you go to more 100 times and the standard deviation is also stable if you do RSR more than 100 times. In this case, the RNE mode will not stagnate and RSR is not better than RNE. If you look at the one-shot trace (green line) randomly chosen from the 1000 traces, it goes really far away from the correct results.

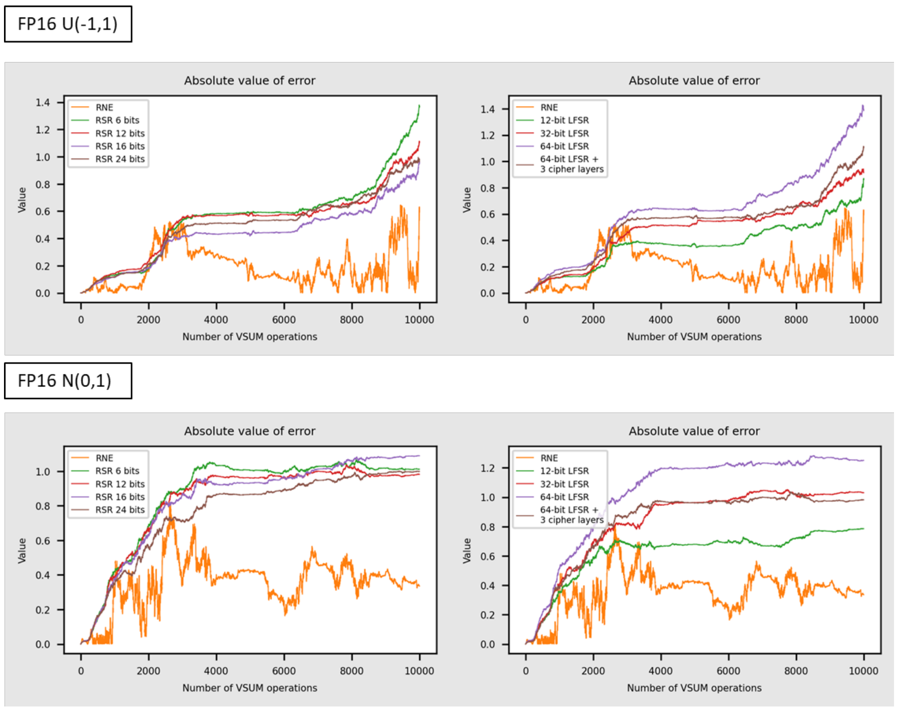

Figure 4.12.: Results of VSUM, $U(-1,1)$ and $N(0,1)$ input. Each line: left, average of abs(error) of RNE and RSR using 64-bit LFSR with 3 cipherlayers, right, average of abs(error) when fixing $p\_rsr = 12$ and varying the configurations of LFSR

Figure 4.12 shows the results when input is an uniform distribution in (-1,1) (first line) and standard Gaussian distribution (second line). The standard deviation are not shown here because they are not influenced by the parameters used in the plots. In these cases, p_rsr doesn't influence the results of RSR, so we will choose $p\_rsr = 12$ for implementation for this specific instruction and format in hardware according to previous results in $U(0,1)$. Plots on the right show the mean abs(error) when we fix $p\_rsr = 12$ and vary the configurations of LFSR. In these cases, again the 12-bit LFSR has the best performance.

4.4.1.3. Comparison with other rounding modes

Now we have our final parameter set: $p\_rsr = 12$, with 12-bit LFSR (no cipherlayers). We can compare the performance with other rounding modes. To be fare to the existing rounding modes, we will only compare the cases when the input distributions are $U(-1,1)$ and $N(0,1)$. The results are shown in Figure 4.13 and Figure 4.14. In the legend, p is the precision of FP16, which is 11 bits. We can list the results from worst performance to higher performance: RTZ, RDN, RUP < RSR 1 time < RMM < RNE ≈ RSR 10 times < RSR 100 times.

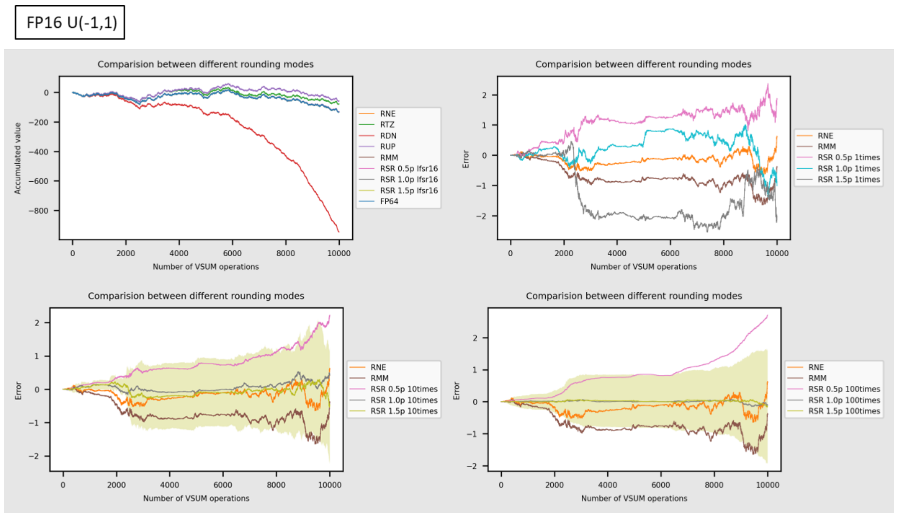

Figure 4.13.: Comparisons between different rounding modes and multiple times of RSR with different p_rsr

4.4.2. Other operations and formats

Then we move to FP8 and FP32, and also SDOTP instruction with different formats. We perform the same analysis for all of them. The results from these different cases are quite similar, so we just give the conclusion directly here and put all the results in the Appendix A.

For the p_rsr, no matter which formats or instructions, p_rsr bigger than 12 is necessary. For LFSR, in most of the cases there's no differences between different LFSR configurations. So for saving area and energy in hardware, we will just implement the simplest LFSR: 12-bit LFSR without cipherlayers.

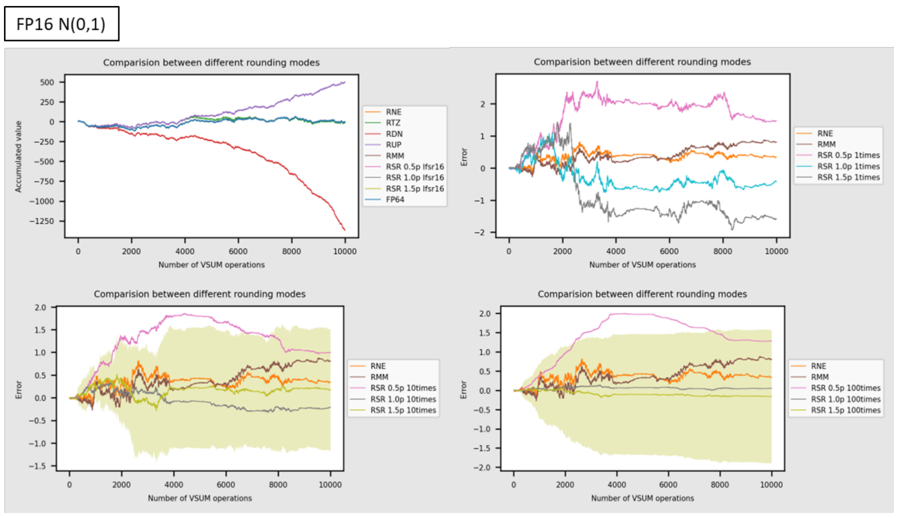

Figure 4.14.: Comparisons between different rounding modes and multiple times of RSR with different p_rsr

4.5. Implementation and AT analysis

As shown in the previous section, you can't get any improvement on the performance for $p\_rsr \geq 12$ for all the formats. And there doesn't exist a configuration of LFSR is generally better than others. To save area on the hardware, we choose $p\_rsr = 12$ and a 12-bit LFSR in the actual hardware.

To analyze the impact to the SDOTP unit by this change, we perform synthesis using Synopsys in TSMC 65nm technology. We use the worst-case corner (1.08V and 125°C) and obtain the area-timing (AT) relations for old and new SDOTP units with different number of pipeline registers. The results are shown in Figure 4.15. There is no influence on timing and the change in area is smaller than 2 kGE (smaller than 5% at 5ns, details are shown in Appendix A). The extension for RSR only add a small overhead to the existing system.

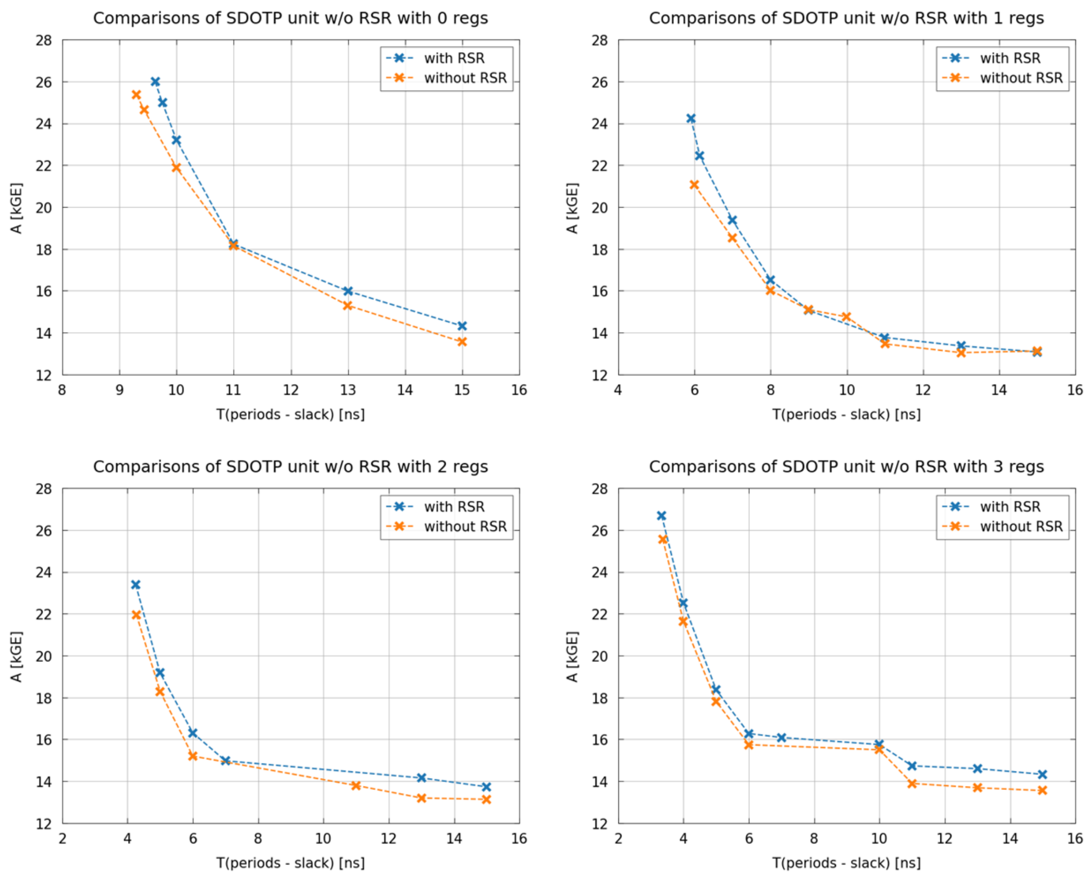

Figure 4.15.: Comparison of SDOTP unit with and without RSR implementation

Chapter 5

Conclusion and Future Work

In this project, we implement our SR algorithm in the rounding unit, and integrate it to the SDOTP module of the FPU by modifying the interface in SDOTP module. Then we verify its functionality and evaluate its performance by comparing the accumulation results to our software golden model in double precision. By testing our SR extension in different instructions and FP formats, we have proven that to reach the optimal performance (smallest error compared to FP64 results), you need at least 12 bits to perform SR with any arbitrary LFSR configuration. As a result, the final implementation in hardware uses 12 bits for rounding and a simplest LFSR which has 12-bit length internally. The AT analysis shows that this modification in SDOTP unit only adds a small ($<5\%$) overhead in area. Finally, to test the impact of this extension in a real chip, we integrate the new FPU with SR extension to a microcontroller architecture, pulpissimo, and finish the tapeout adaptions.

This is the first open-source hardware with SR extension. By testing all the different possible configurations in hardware intensively, we can make sure our final implementation has the best SR performance and at the same time, the minimal impact on the existing system.

The performance of SR varies in different application scenarios. First, if you rely more on the statistical behavior of the results, SR is definitely more advantageous. We have shown that if results are averaged over more than 10 times, SR would give a smaller error compares to RNE. Second, if you need some small adjustments to some relatively large value, SR would also be helpful. In this case, stagnation is likely to happen in RNE but SR is intrinsically immune to that. But in other cases, especially when you are not in the region where RNE stagnates, we have shown RSR has larger error compare to RNE mode and RNE would be a better choice. This is easy to understand, suppose you have a number, 0.2. In RNE case, it will be round to 0. But in SR case, you have 0.2 possibility to be rounded to 1, which has an error of 0.8. In cases where statistical properties are important, you should use SR because the expection value of this rounding is 0.2, and you have "no information loss". But if you only focus on one operation, the expentation value of abs(error) would be $0.2 \times 0.8 + 0.8 \times 0.2 = 0.36$, bigger than 0.2 in RNE case. By encoding the rounded bits in the probability, SR enhances the performance in cases where statistical information is important or results suffer from stagnation in RNE. But the price you pay is that you have some probability to go to a value with larger error, and this makes SR not a good choice in other cases.

One way to further improve our SR unit is to make application-specific adaptations, for example, typical deep learning tasks, scientific computing, quantum computing, etc. Via parameter tuning, algorithm adjustment or pseudo-random-generator upgrade, we may promote the result accuracy by a notable amount. Another possible application of SR is to algorithms that already imply randomness, like Monte Carlo methods, where SR is hopefully promised more freedom.

Up to now, the results presented in this report are only from simulations. In order to gain a more comprehensive and realistic evaluation of our stochastic rounding extension (e.g. power consumption), we need to see through the complete process of back-end design until chip tapeout. Namely, chip testing will provide the ultimate results.

Appendix A

More Evaluation Results

A.1. VSUM, FP8 Input, FP8 Output

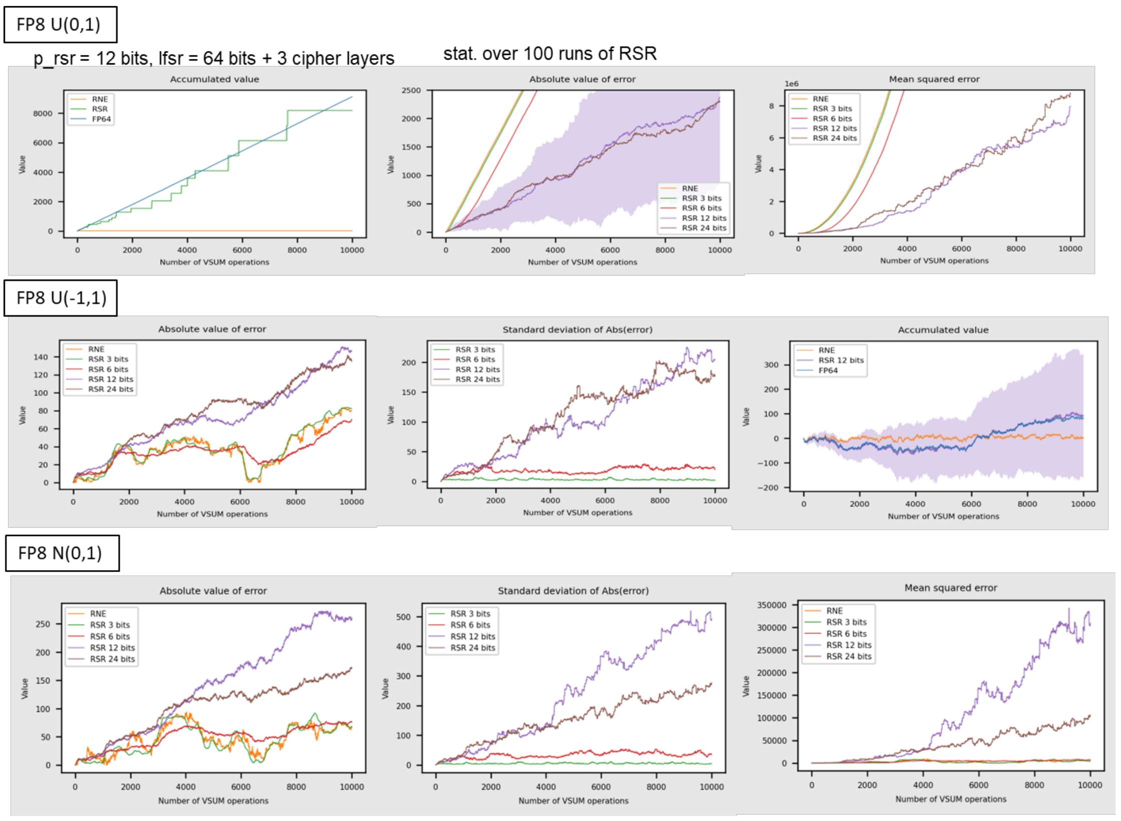

Figure A.1.: Results of changing parameters for VSUM, FP8 input.

A.2. VSUM, FP32 Input, FP32 Output

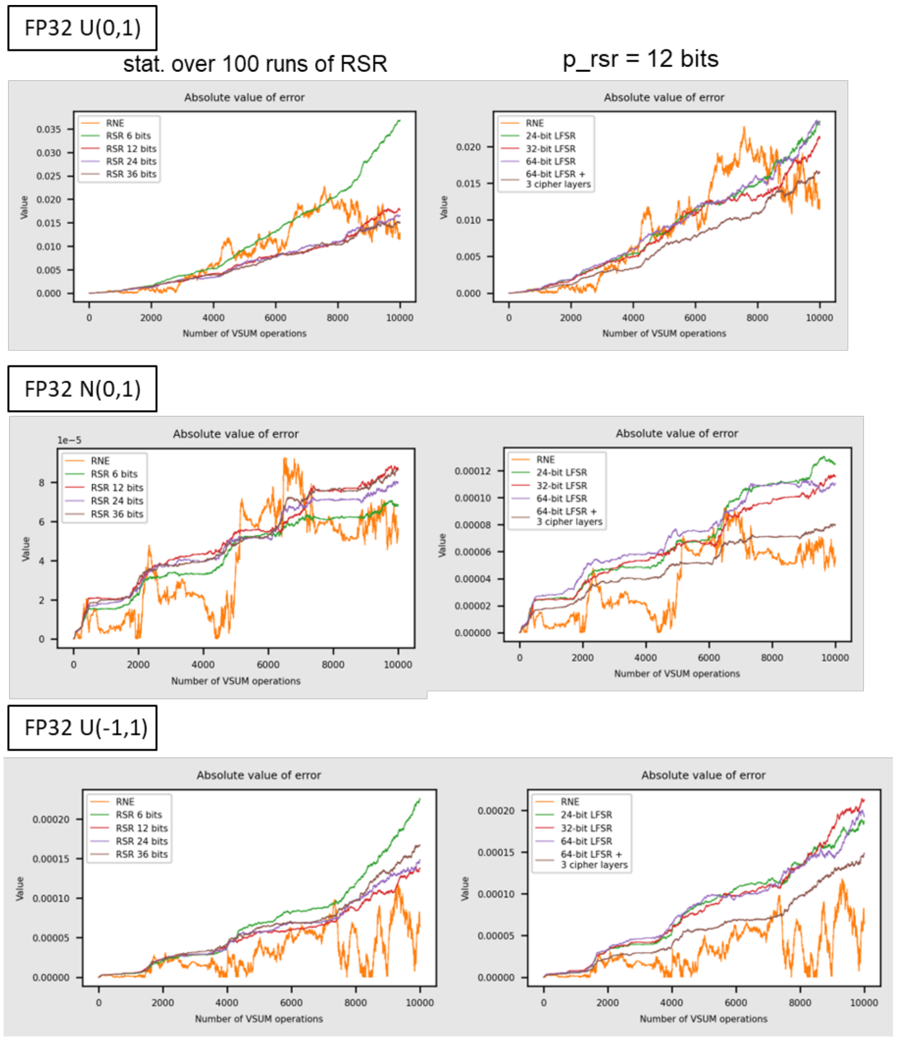

Figure A.2.: Results of changing parameters for VSUM, FP32 input.

A.3. SDOTP, FP8 Input, FP16 Output

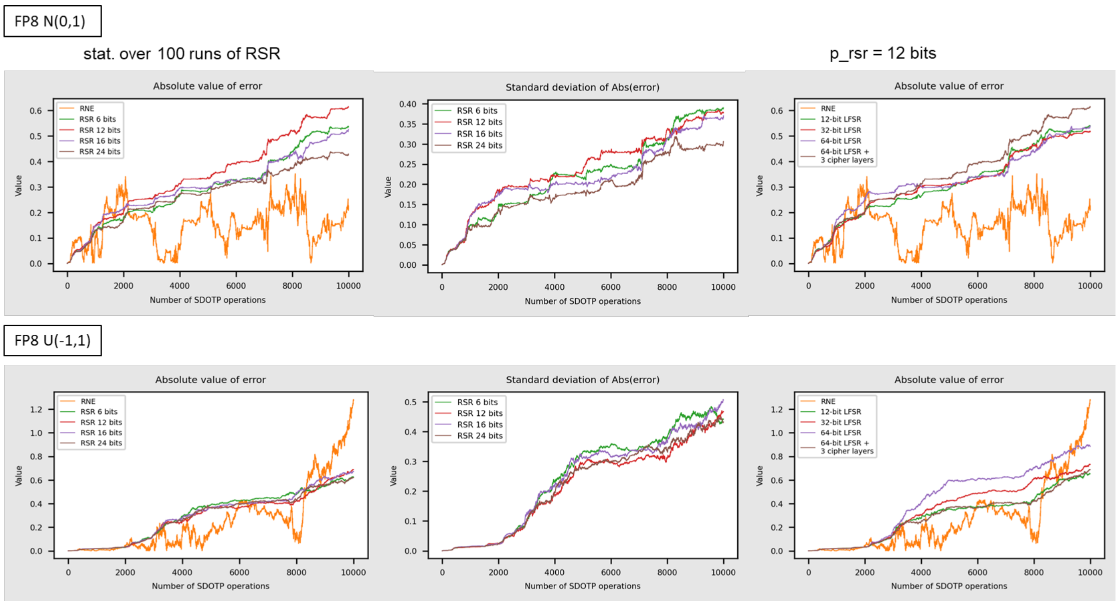

Figure A.3.: Results of changing parameters for SDOTP, FP8 input.

A.4. SDOTP, FP16 Input, FP32 Output

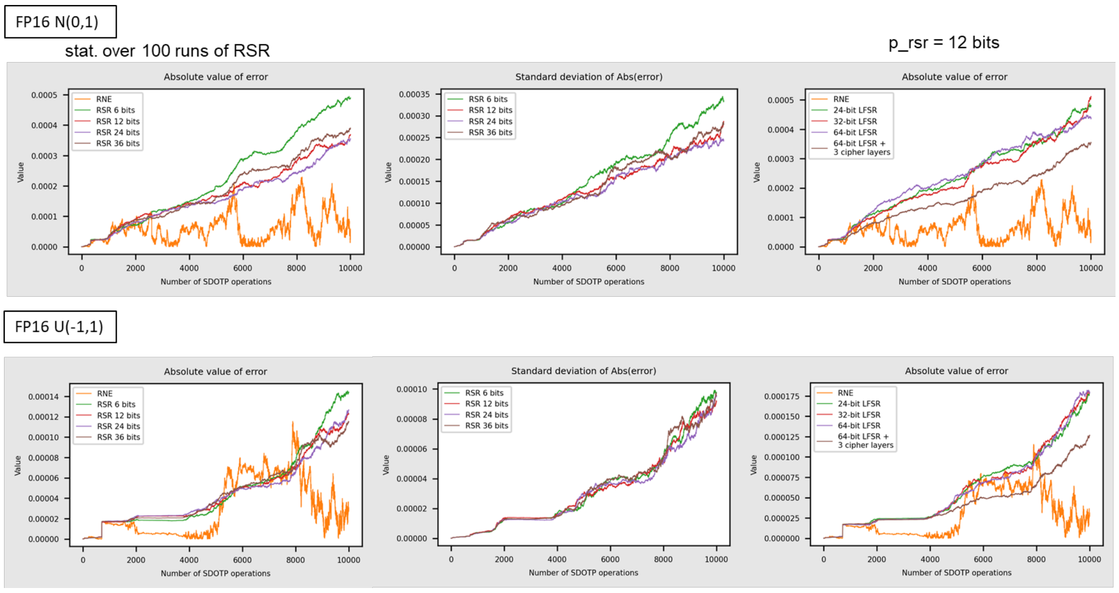

Figure A.4.: Results of changing parameters for SDOTP, FP16 input.

A.5. Comparison of SDOTP Unit with Different Pipeline Registers

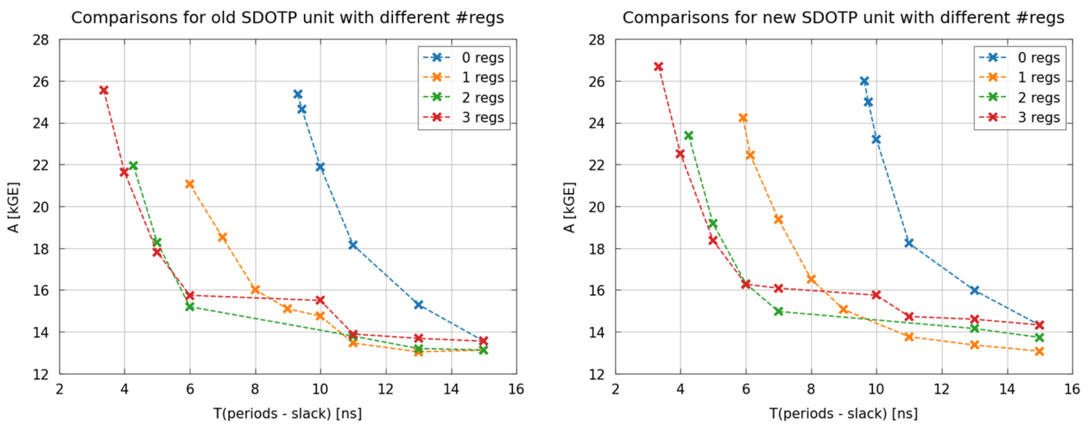

Figure A.5.: Comparison of SDOTP unit with different pipeline registers

A.6. Area of SDOTP Unit with and without RSR Extensions

Table A.1.: Area of SDOTP Unit with and without RSR Extensions

Time[ns]	**#reg**	**old[μm^2, kGE]**	**new[μm^2, kGE]**	**incr**
4	3	27700 21.6	28827 22.5	4.07%
5	3	22804 17.8	23504 18.4	3.07%
6	3	20149 15.7	20831 16.3	3.38%
5	2	23405 18.3	24534 19.2	4.82%
6	2	19449 15.2	20868 16.3	7.30%

Appendix B

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Semester Thesis at the Department of
Information Technology and Electrical Engineering

Summer Semester 2022

Design of a Floating-Point Stochastic Rounding Unit

Jayong Li and Enci Zhang

March 16, 2022

Luca Bertaccini, ETZ J 78, Tel. +41 44 632 55 58, `lbertaccini@iis.ee.ethz.ch`
Gianna Paulin, ETZ J 76.2, Tel. +41 44 632 60 87, `pauling@iis.ee.ethz.ch`
Tim Fischer, ETZ J 76.2, Tel. +41 44 632 59 12, `fischeti@iis.ee.ethz.ch`

Handout:	–
Due:	–

1 Introduction

The Internet of Things (IoT) era is characterized by billions of devices gathering data through sensors and sending them to servers, where the data can then be processed. To save bandwidth and energy across the network, a pre-processing step may also be implemented directly on the IoT device. Since transmitting data requires more energy than computing on the IoT device, pre-processing the information and sending a lower amount of data is more efficient than sending the whole information.

PULPissimo [1] is the microcontroller architecture of the more recent IoT PULP chips. PULPissimo takes care of autonomous I/O, advanced data pre-processing, external interrupts, etc. The PULPissimo architecture is built upon a single CV32E40P core [2], a small and efficient 4-stage 32-bit RISC-V core. The integer core is coupled with a floating-point unit (FPU) [3], thus providing support for single-precision (FP32) floating-point instructions.

Recently, an existing FPU has been extended to add support for low-precision floating-point formats and floating-point dot products with accumulation (SDOTP) [7]. Such a module allows for computing double the number of fused multiply-add in one cycle, a higher energy efficiency, and enables better accuracy results. Since floating-point additions are non-associative, two consecutive additions might be lossy, while the dot product unit is lossless.

Various studies on neural networks training with low-precision floating-point formats have proven accuracy benefits when employing stochastic rounding over the widely used round-to-nearest-even (RNE) mode [5]. However, such studies rely on software libraries emulating floating-point stochastic rounding. The first products supporting stochastic rounding, the Intel Loihi chip and the new Graphcore IPU, have been recently released [6]. However, those implementations are closed-source.

In this project, we investigate hardware support for stochastic rounding and explore the benefits of including it into PULPissimo.

2 Project Goals

The main tasks of this project are:

- **T1: Implement a stochastic rounding hardware unit** In this task, the students will develop a stochastic rounding unit and integrate it into the open-source floating-point dot product unit.

- **T2: Test the newly added stochastic rounding unit** In this task, the students will test their stochastic rounding unit comparing its result with a software golden model.

- **T3: Comparison with the native SDOTP unit using RNE** In this task, the students will compare the SDOTP unit with support for stochastic rounding unit against its native implementation in terms of area, and timing and performing an error analysis.

- **T4: Tapeout contribution** In this task, the students will contribute to a tapeout of the PULPissimo chip including the stochastic rounding unit.

3 Deliverables

1. **D1:** Working SDOTP module including stochastic rounding support.
2. **D2:** Comparison with baseline system (SDOTP unit using RNE).
3. **D3:** Backend scripts for tapeout.

4 Project Realization

4.1 Project Plan

Within the first month of the project you will be asked to prepare a project plan. This plan should identify the tasks to be performed during the project and sets deadlines for those tasks. The prepared plan will be a topic of discussion of the first week's meeting between you and your advisers. Note that the project plan should be updated constantly depending on the project's status.

4.2 Meetings

Weekly meetings will be held between the student and the assistants. The exact time and location of these meetings will be determined within the first week of the project in order to fit the students and the assistants schedule. These meetings will be used to evaluate the status and progress of the project. Beside these regular meetings, additional meetings can be organized to address urgent issues as well.

4.3 Coding Guidelines

HDL Code Style Adapting a consistent code style is one of the most important steps in order to make your code easy to understand. If signals, processes, and modules are always named consistently, any inconsistency can be detected more easily. Moreover, if a design group shares the same naming and formatting conventions, all members immediately *feel at home* with each other's code. At IIS, we use lowRISC's style guide for SystemVerilog HDL: `https://github.com/lowRISC/style-guides/`.

Software Code Style We generally suggest that you use style guides or code formatters provided by the language's developers or community. For example, we recommend LLVM's or Google's code styles with `clang-format` for C/C++, PEP-8 and `pylint` for Python, and the official style guide with `rustfmt` for Rust.

Version Control Even in the context of a student project, keeping a precise history of changes is *essential* to a maintainable codebase. You may also need to collaborate with others, adopt their changes to existing code, or work on different versions of your code concurrently. For all of these purposes, we heavily use *Git* as a version control system at IIS. If you have no previous experience with Git, we *strongly* advise you to familiarize yourself with the basic Git workflow before you start your project.

4.4 Report

Documentation is an important and often overlooked aspect of engineering. One final report has to be completed within this project. The common language of engineering is de facto English. Therefore, the final report of the work is preferred to be written in English. Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Tgif[1] or any other vector drawing software (for block diagrams) is strongly encouraged by the IIS staff.

Final Report The final report has to be presented at the end of the project and a digital copy need to be handed in. Note that this task description is part of your report and has to be attached to your final report.

4.5 Presentation

There will be a presentation (15 min presentation and 5 min Q&A) at the end of this project in order to present your results to a wider audience. The exact date will be determined towards the end of the work.

References

References

[1] https://github.com/pulp-platform/pulpissimo

[2] https://github.com/openhwgroup/cv32e40p

[3] Mach, Stefan, et al. "Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 29.4 (2020): 774-787.

[4] https://github.com/T-head-Semi/opene906

[5] Wang, Naigang, et al. "Training deep neural networks with 8-bit floating point numbers." Advances in neural information processing systems 31 (2018).

[6] Croci, Matteo, et al. "Stochastic Rounding: Implementation, Error Analysis, and Applications." (2021).

[7] https://github.com/pulp-platform/fpnew/tree/feature/expanding_dotp

Zürich, March 16, 2022 Prof. Dr. Luca Benini

[1]See: `http://bourbon.usc.edu:8001/tgif/index.html` and `http://www.dz.ee.ethz.ch/en/information/how-to/drawing-schematics.html`.

Appendix C

Declaration of Originality

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Design of a Floating-Point Stochastic Rounding Unit

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Li	Jiayong
Zhang	Enxi

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 08.06.2022	Jiayong Li
	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Bibliography

[1] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/335d3d1cd7ef05ec77714a215134914c-Paper.pdf

[2] C. Su, S. Zhou, L. Feng, and W. Zhang, "Towards high performance low bitwidth training for deep neural networks," *Journal of Semiconductors*, vol. 41, no. 2, p. 022404, 2020.

[3] L. K. Muller and G. Indiveri, "Rounding methods for neural networks with low resolution synaptic weights," 2015. [Online]. Available: https://arxiv.org/abs/1504.05767

[4] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 774–787, 2020.

[5] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an ultra-low-power pulpissimo soc in 22nm fdx," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, 2018, pp. 1–3.

[6] M. Höhfeld and S. E. Fahlman, "Probabilistic rounding in neural network learning with limited precision," *Neurocomputing*, vol. 4, no. 6, pp. 291–299, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/092523129290014G

[7] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and

D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 1737–1746. [Online]. Available: https://proceedings.mlr.press/v37/gupta15.html

[8] T. Na, J. H. Ko, J. Kung, and S. Mukhopadhyay, "On-chip training of recurrent neural networks with limited numerical precision," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 3716–3723.

[9] M. K, "Stochasticrounding.jl," 2020. [Online]. Available: https://github.com/milankl/StochasticRounding.jl

[10] M. Croci, M. Fasi, N. J. Higham, T. Mary, and M. Mikaitis, "Stochastic Rounding: Implementation, Error Analysis, and Applications," Oct. 2021, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03378080

[11] T. Zhang, Z. Lin, G. Yang, and C. D. Sa, "Qpytorch: A low-precision arithmetic simulation framework," 2019.

[12] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: an ultra-lightweight block cipher," vol. 4727, 09 2007, pp. 450–466.