POSIT, final report

Shay Agroskin
agroskinshay@campus.technion.ac.il
Shahaf Haller
hallershahaf@campus.technion.ac.il

Contents

1 Abstact

This project studies and implement the a floating point format called POSIT. It explains the differences between the POSIT and the current industry standard in terms of implementation complexity and computation accuracy. section 6 describes the implementation of different arithmetic units for the new format using SystemVerilog.

The project concludes that the new format is more suitable for workflows that need high accuracy and don't need to represent high range of numbers. Combined with the implementation simplicity, the new format can be a candidate to replace the current IEEE 754 standard.

2 Introduction

The project analyzes a new way to do fractions arithmetic, instead of the standard floating point implementation used today.

Fraction arithmetic efficiency and accuracy draws more attention in recent years due to the growing popularity of machine learning and more complex graphic designs (e.g. video games). While some approaches propose different implementations of the current IEEE standard, other studies suggest a different way to represent fractions all together. This project studies one of these new formats called POSIT[1].

2.1 Floating points

The current industry standard for fraction representation is the IEEE standard 754 (floats), in which a fraction is represented by three fields: sign, exponent and significant (see Figure 1). By denoting the sign as s, the exponent as e and significant as f, the decimal representation of the number is represented by:

$$(-1)^s \cdot 2^{e-bias} \cdot 1.f \tag{1}$$

The range and precision of numbers that can be represented using this format along with the value of *bias* depend on the vector's length. The standard defines 3 types of vector lengths:

	vector length	exponent bits	fraction bits	bias
half precision	16	5	10	15
single precision	32	8	23	127
double precision	64	11	52	1023

The most used lengths nowadays are a 32 and 64 bit vectors.

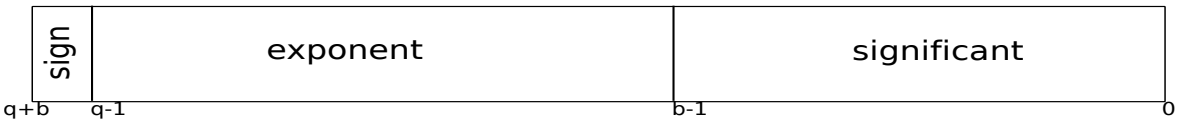

Figure 1: IEEE 754 standard fraction representation

For example, the number 1.5 can be represented by setting $s = 0, e = bias, f = 1.5$ since

$$(1.5)_{10} = (1.1)_2 = (-1)^0 \cdot 2^{bias-bias} \cdot 1.5$$

As seen in Equation 1, the *significant* only specifies the fractional part of the number "$1.significant$". In our example, this implies that the value f would be represented by 0.1 in the floating point vector.

2.1.1 Special values

The format reserves some special values with different meaning:

- Zero - zero is a special value denoted with an exponent and fraction of 0. The format distinguishes between $+0$ and -0 using the *sign* bit. Although each has its own representation, both zeroes are equal.
- Denormalised - If the *exponent* is all zeroes, but the *fraction* is not, then the value is a denormalized number. This means that this number does not have an assumed leading one before the binary point:

$$(-1)^{sign} \cdot e^{-bias} \cdot 0.fraction$$

- Infinity - The value $\pm\infty$ are denoted with an *exponent* of all ones and a *fraction* of all zeroes. The sign bit distinguishes between negative and positive infinity. Every multiplication and addition with the infinity value would result in infinity except multiplication by 0 which would result in NAN.
- Not A Number (NAN) - The value NAN is used to represent a value that is an error. This is represented when *exponent* field is all ones with a zero sign bit and *fraction* field that is not zero.

Table 1 summarizes the representation of the IEEE 754 special values.

Exponent	Fraction	value
0	0	0
all set	0	$\pm\infty$
0	not 0	denormalized
all set	not 0	Not A Number (NAN)

Table 1: Summary of IEEE 754 special values

2.2 Caveats

Although ubiquitous, the IEEE standard 754 has several downsides. Since the exponent and fraction bit lengths are fixed, the range and accuracy of a number that can be represented using this format are limited, which in turn case some problems.

2.2.1 Representation problem

The IEEE standard 754 requires choosing the base of the exponent part and the number of bits to represent it during implementation. Regardless of the chosen base, some number would always be impossible to represent exactly.

For example, by choosing base 2, the number 0.3 cannot be represented precisely and is rounded to a value that can be represented by

$$2^{e-bias} \cdot 1.f$$

Since 0.3 representation is only and approximation of the real number, the following Python code

```
> 0.3 + 0.3 + 0.3 == 0.9
```

is mistakenly evaluated to “False”.

2.2.2 Granularity and Range problem

The range and precision of an IEEE 754 number depend on the number of bits allocated to it during implementation. Allocating more bits to the exponent part can the range of numbers, while increasing the *fraction* field can increase their accuracy. For example, to represent the number $\frac{1}{3}$ we need infinite bits for the fraction part.

Moreover, as the *exponent* value increases, the precision granularity gets worse (as seen in Figure 2) since each change in the *fraction* part is multiplied by the *exponent* value (see Equation 1 for reference).

This creates a situation where precision-sensitive workflows need to settle on the given *fraction* bits, even when the big range is not required (since the fraction part cannot use the exponent bits to increase precision).

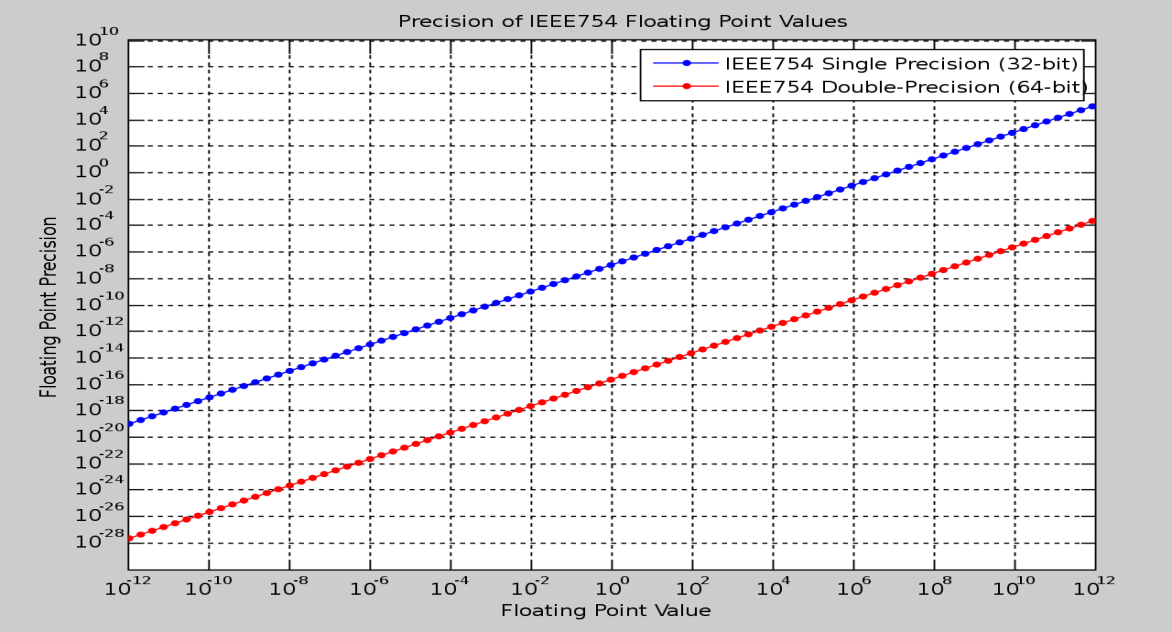

Figure 2: Precision for different float values

2.2.3 Wasteful representation

There are more than one way to represent the same number using the existing format. As explained in subsubsection 2.1.1 the NAN number are represented by setting the *exponent* bits to zero and *fraction* bits to not zero. In single precision format, this leads to $2 \cdot (2^{23} - 1)$ different representation of NANs. Since all NAN are the same, this representation is quite wasteful.

3 POSIT

In order to tackle some of the IEEE 754 format’s problems, a new format was proposed by John Gustavson which is called POSIT[1].

The new format divides the bit vector into four parts (3): *sign, regime, exponent* and *fraction*, which together form the decimal number:

$$(-1)^{sign} \cdot (useed^{regime}) \cdot 2^{exponent} \cdot 1.fraction \tag{2}$$

where $useed = 2^{2^{es}}$ and es is the number of bits used for the exponent part.

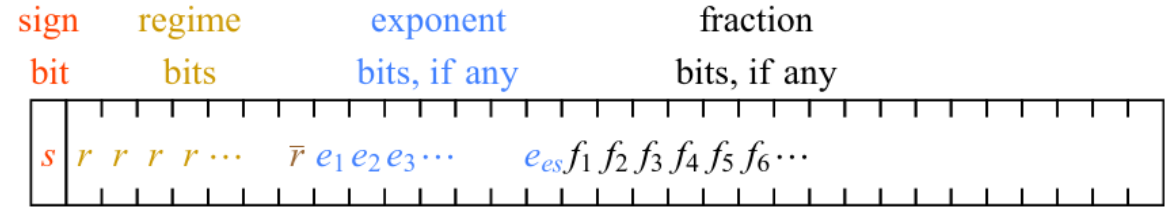

Figure 3: Generic posit format for finite, nonzero values

3.1 regime value

The *regime* value depends on the running sequence starting after the sign bit in the following manner:

$$bbbbbbb\bar{b}$$

in this sequence of n b's and one $\bar{b}$

- if b=0, the regime equals to -n
- if b=1, the regime equals to n-1

Figure 4 shows more examples of *regime* sequences and their value in the Equation 2:

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, *k*	−4	−3	−2	−1	0	1	2	3

Figure 4: Different regime sequences and their respective value

3.2 Dynamic range and accuracy

As explain the *regime* description in subsection 3.1, the *regime* field does not have a fixed size: The number of bits in the regime part are determined by the first flipped bit after the bit sequence.

This implies that both *01* and *11111111111110* are valid regime sequences (equal to −1 and 13 respectively) which can be part of a 32bit POSIT vector. In fact, the regime sequence can take up to the whole POSIT bit vector. Since the exponent field size is fixed, the fraction bit width change according the regime part.

If no flipped bit exists, meaning that the POSIT vector has to form of *sbbbbbbbb* where s is the sign bit, then the POSIT vector receives a special value depending on the value of b and s (see **??** for more information).

The dynamic field size help with the granularity and range problem (2.2.2) by allowing workflows to use large regime fields when working with very large number and small regime fields when in need of high accuracy (the size of the fraction field is implied by the regime part).

3.3 exponent and fraction fields

The *exponent* field width, denoted as *es*, is fixed and determined at implementation. The field's starts after the first flipped bit that marks the *regime* field's end (see subsection 3.1). If there are less than *es* bits left after the *regime* field, the *exponent* field is right-padded with zeroes.

Unlike the IEEE 754 standard, in POSIT the *exponent* field represent a positive number without *bias*. This implies that the *exponent* range is $0 \leq exponent \leq 2^{es} - 1$

3.3.1 fraction field

The *fraction* part of the POSIT starts after the *exponent* field (see Figure 3) and its width depends on the number of bits occupied by the *sign*, *regime* and *exponent* parts(3.1).

Like the IEEE 754 format, the POSIT format assumes that the *fraction* field only describes the fractional part of a number $1 \leq n < 2$. For example, the decimal representation of the *fraction* field $b_1, b_2, \ldots, b_n$ is:

$$1 + \sum_{i=1}^{n} b_i \cdot 2^{-i}$$

Combining the *sign*, *regime* and *exponent* parts the decimal representation of a POSIT vector is

$$(-1)^{sign} \cdot (useed^{regime}) \cdot 2^{exponent} \cdot 1.fraction \tag{3}$$

Figure 5 shows different POSIT vectors and their decimal value. As the figure shows, in case the fraction part is absent, it is assumed to be zero (i.e. its decimal representation is $1 + 0 = 1$)

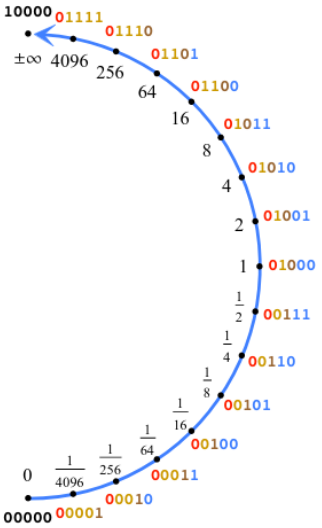

Figure 5: 5-bit POSIT vectors with es $= 2$, useed $= 2^{2^{es}} = 16$ and their decimal value

3.4 POSIT special values

Similar to IEEE 754 the POSIT format defines special values:

- infinity -POSIT format does not differentiate between $\pm\infty$ and represents them both by setting the *sign* bit and clearing the rest of the vector (as can be seen in Figure 5).
- zero - a vector with all its bits cleared represents the zero value. POSIT has only representation for it.
- regime only number - if the *regime* part consists of striding 1's and has no flipped bit, then the vector is assumed to have *exponent* and *fraction* part equal to 0, and the *regime* field is assumed to have flipped bit at its end. For example, for a 16 bit POSIT vector, this value would be represented as:

  $$s111111111111111$$

  Where s is the sign bit. In this example the number is equal to

  $$(-1)^s \cdot (useed)^{14}$$

Table 2 summarizes the special POSIT values

sign	the other	bits value
0	all cleared 0	
1	all cleared	$\pm\infty$
0	all set	$(useed)^{n-1}$
1	all set	$-(useed)^{n-1}$

Table 2: Special POSIT values

Unlike the IEEE 754 format, POSIT doesn't have a representation of NaN. In order to signal that a computational result is invalid the hardware implementation of the format should raise an interrupt.

4 Related work

Several models were proposed in order to do fraction arithmetic more efficiently. This section describes some of these methods.

4.1 Representing fractions by numerator and denominator

The use of a format that specifies the *exponent* part separately, e.g. IEEE 754(2.1) or POSIT(3), limits representation of numbers which that cannot be expressed using a chosen.
For example, when using the IEEE 754 format with base 2, one cannot represent $\frac{1}{3}$ exactly, since $\frac{1}{3}$ cannot be represented as a sum of 2^{-i} (see section 3 for more information).

One approach to represent rational numbers more accurately is to represent every number using its nominator and denominator values. For example to represent the number $\frac{1}{3}$ the implementation would hold the values 1 and 3.

Multiplication and division could be implemented using integer arithmetic. Addition and subtraction would require first finding the LCM of the two denominators before adding/subtracting the nominators.

Irrational numbers could be represented as an approximation using a rational number e.g. $\pi \approx \frac{22}{7}$.

This method already has some software implementations, for example *Fraction* data type.

4.1.1 Caveats

Frequent fraction additions with the use of *lcm* (different denominators), might result in a too large denominator and might require the use of the *gcd* algorithm to reduce it.

This would result in a very complex implementation with high latency per each arithmetic operation, and might not be suitable for many performance centered workflows.

4.2 Increasing Float size

As described in subsubsection 2.2.2, the bits in the *fraction* field results in higher accuracy. Workflows which require high accuracy can use a modified version of the IEEE 754 format with more bits for the *fraction* parts. This solution presents a somewhat "quick fix" as it does not require to change the existing HW significantly.

4.2.1 Caveats

Though simple, this solution does not solve the any of the problems listed in subsection 2.2 and merely reduces the impact of some of them. Also, increasing the total bits per float would result in increase in either latency or power consumption.

5 Environment and tools

- Icarus iverilog and vvp - Most of the development for this project was done on Linux private computers. The free *iverilog* allows for a command line compilation of System Verilog code. The *vvp* is used for running the compiled file.
- Inkscape - this tool is used to creating vector design (similar to Adobe illustrator, but free). It was used to designing the logic diagrams for various components.
- Synopsis tools for synthesis and elaboration - these tools were used to verify that the code is indeed synthesizable.

6 Implementation overview

Similar to Intel's ALU implementation, This project's implementation first divides the POSIT vector into three fields *regime*, *exponent* and *fraction*, each of the same bit length as the original POSIT vector (as depicted in Figure 6). For example, a 32bit POSIT vector would be divided into three fields that are 32bit long each.

These fields represent the value of the *regime*, *exponent* and *fraction* parts of the vector *after* they've been converted into a 2's complement representation. Figure 6 illustrates a block diagram of the POSIT adder implementation. As can be seen from the figure, the addition is done on each field separately and the results are combined ("packed") together afterwards to form a valid POSIT vector (see "Packer" subsubsection6.1.2)

6.1 Packer and unpacker modules

As explained, in this implementation, every arithmetic operation on a POSIT vector, is performed on each of its fields (*regime*, *exponent*, *fraction*) separately and the results of all operations are then combined into a new POSIT vector.

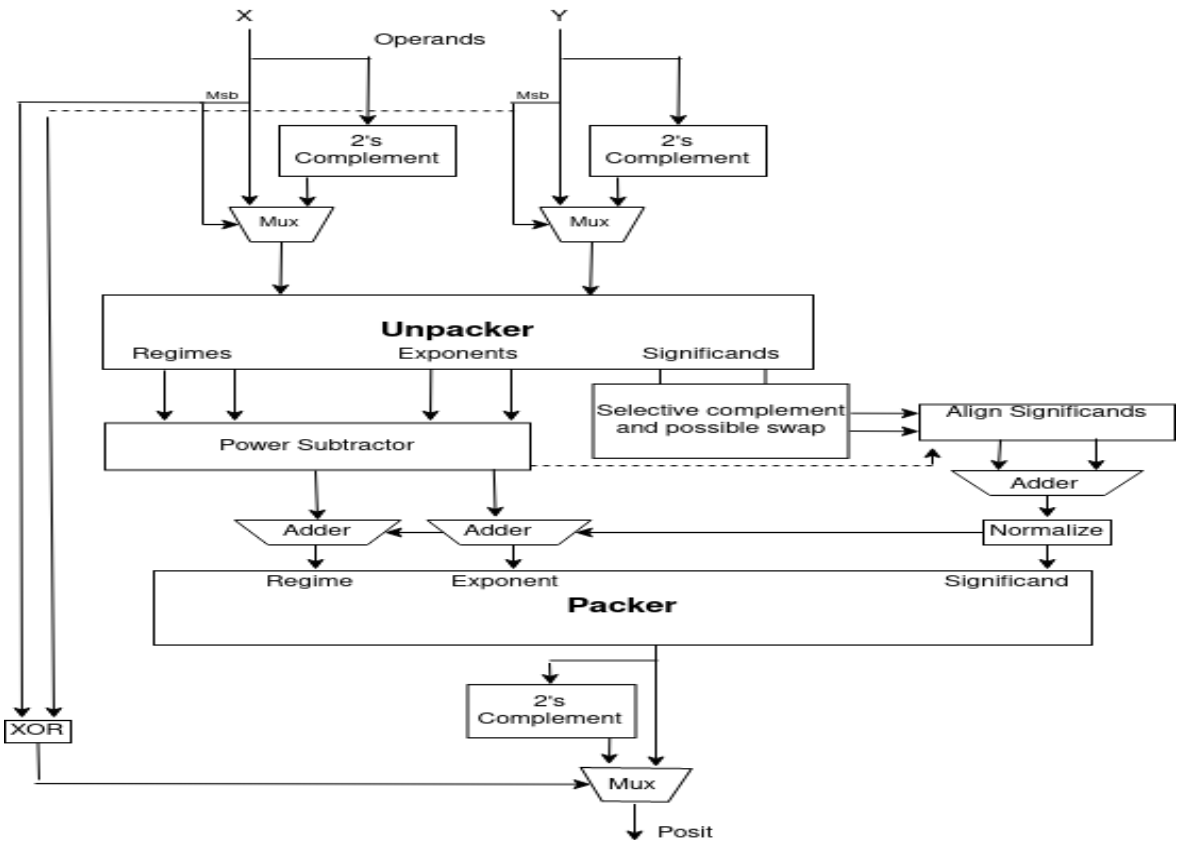

Figure 6: Block diagram of the adder module

6.1.1 Unpacker

The *unpacker* module is responsible for converting a POSIT vector into three fields that represent its *regime*, *exponent*, *fraction* parts in 2's complement representation.

The logic gate diagram (illustrated in Figure 7) shows how the *unpacker* module receives a POSIT vector of *bits* length and returns its three components.

To better understand the implementation of the *unpacker* module, Figure 8 illustrates the *seed_lookup* module which takes a POSIT vector and by XORing all bits finds the first flipped bit of the *regime*. Afterwards, the module uses the *regime* value to shift the POSIT vector left so the *exponent* part starts in the MSB. It does so to make it easier for the *unpacker* module to extract the other fields of the input vector.

The implementation avoids using any register dependant logic to achieve low latency.

6.1.2 Packer

The *packer* does the exact opposite of *unpacker* module. It receives *regime*, *exponent*, *fraction* from the *multiplier* or *adder* and creates a POSIT vector from it. The *packer* also receives an additional *zero* flag which tells it if the result of the previous module is zero.

The following pseudocode describes the verilog implementation of this module:

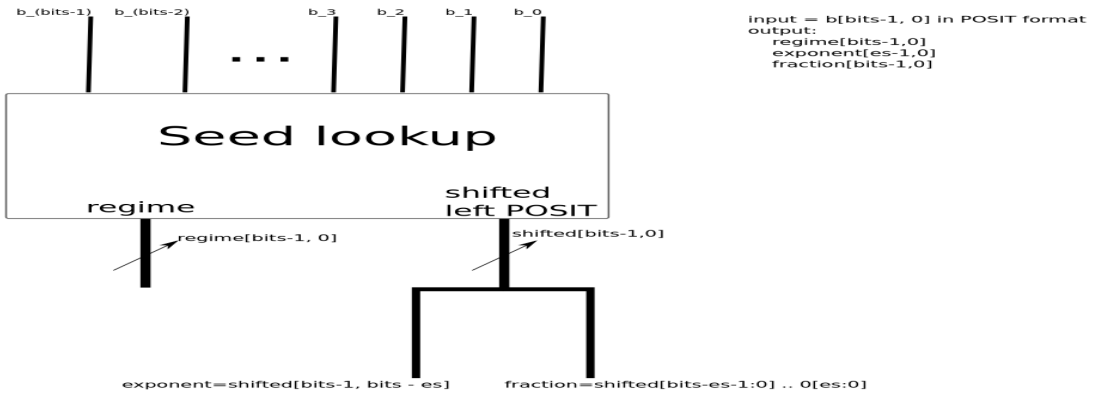

Figure 7: Unpacker diagram

Data: seed, exponent, fraction and zero
Result: a POSIT vector
if *zero is set* **then**
  return POSIT = zero vector
else
  set seed_seq = sequence of running 1s or 0s depending on sign(seed)
  set POSIT = seed_seq concatenated with exponent and fraction parts
  return the first n bits of the POSIT vector, where n is the desired POSIT length
end

The extra *zero* input is needed because the numbers one and zero both have all three parts equal to 0 (i.e. $seed = 0,\ exponent = 0,\ fraction = 0$). This issue is discussed in subsection 7.1.

6.2 Multiplication and Addition

The *multiplier* and *adder* modules receive two POSIT vector and return their product and sum accordingly (as shown in Figure 9 for the adder). Each of them uses the *unpacker* and *packer* modules internally to decompose each of the received vectors into their respective *regime*, *exponent* and *fraction* fields. The addition/multiplication is done on each of the fields separately and the three results are combined into a single POSIT vector.

6.2.1 Adder

As explained in subsection 6.2, the addition starts by decomposing the input POSIT vectors into their three components.

The *adder* module can be expressed by the following pseudo-code:

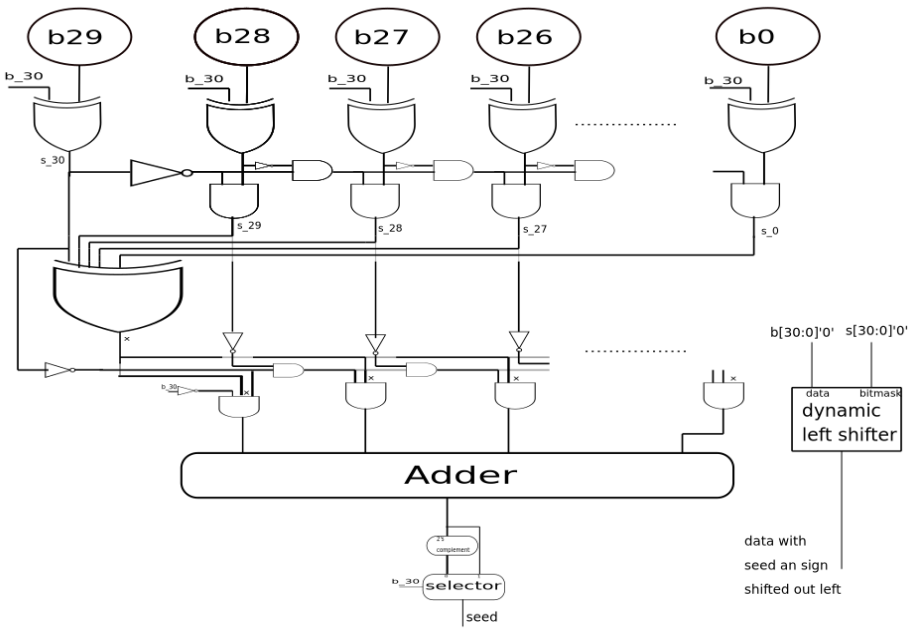

Figure 8: seed_lookup logic gate diagram

Data: Two POSIT vectors p_1, p_2
Result: vectors sum in POSIT format p_r

denote seed, exponent and fraction fields of the two vectors as $s_1, e_1, f_1, s_2, e_2, f_2$
Also denote the fields of the p_r as s_r, e_r, f_r.
if $p_1 = \pm\infty$ *or* $p_2 = \pm\infty$ **then**
    return $p_r = \pm\infty$
else
    $sum_i = s_i << ES + e_i \quad i = 1, 2$
    if $sum_1 > sum_2$ *or* $(sum_1 = sum_2$ *and* $f_1 > f_2)$ **then**
        bigFrac = f_1; bigExpSum = sum_1
        smallFrac = f_2; smallExpSum = sum_2
    else
        bigFrac = f_2; bigExpSum = sum_2
        smallFrac = f_1; smallExpSum = sum_1
    end
    $resultExp = bigExpSum$
    $sumDiff = bigExpSum - smallExpSum$
    $resultFrac = bigFrac + (smallFrac >> sumDiff)$
    if $resultFrac$ *overflows* **then**
        resultExp = resultExp + 1
        resultFrac = resultFrac / 2
    else
        resultExp -= number of time to shift $resultFrac$ left until its $MSB = 1$
    end
    s_r = resultExp << es
    e_r = first es bits of resultExp
    f_r = resultFrac
end

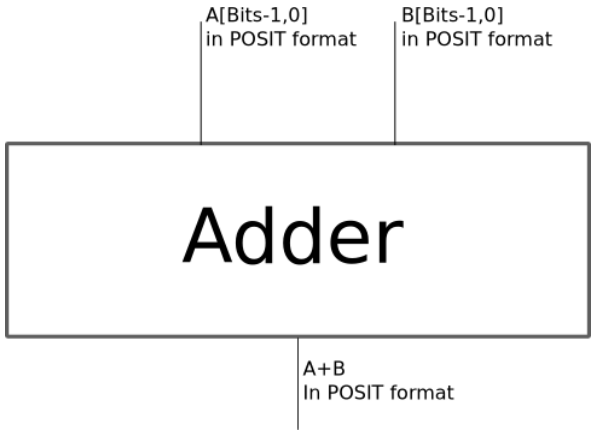

Figure 9: Adder blackbox scheme

To put the algorithm in words:

1. if either of the POSITs represent $\pm\infty$ (same representation, see subsection 3.4), set the result to be $\pm\infty$ (lines 3-4)
2. the *adder* normalizes the received vectors so that they both have the same exponent (seed + exponent field). The equation in line 6 equals to $s_i \cdot 2^{es} + e_i$ (which is equal to the total exponent of the POSIT). (lines 6-16)
3. add the normalized fraction parts and adjust the common exponent to reflect an overflow in the result. (lines 18-19)
4. adjust the common exponent so that the fraction sum would be between 1 and 2. This is needed because the fraction part in POSIT is assumed to be in that range. Therefore the exponent needs to be adjusted to scale the fraction part as needed. (line 21)

6.2.2 Multiplier

As oppose to the *adder* module, multiplication doesn't require normalizing the exponents before operation. The following pseudo-code describe its operation:

Data: Two POSIT vectors p_1, p_2
Result: vectors product in POSIT format p_r
1 denote seed, exponent and fraction fields of the two vectors as $s_1, e_1, f_1, s_2, e_2, f_2$
2 Also denote the fields of the p_r as s_r, e_r, f_r.
3 **if** $p_1 = \pm\infty$ *or* $p_2 = \pm\infty$ **then**
4 | return $p_r = \pm\infty$
5 **else**
6 | $s_r = s_1 + s_2$
7 | $e_r = e_1 + e_2$
8 | $f_r = f_1 * f_2$
9 | **if** f_r *overflows* **then**
10 | | $e_r = e_r + 1$
11 | **end**
12 | **if** e_r *overflows* **then**
13 | | $s_r = s_r + 1$
14 | **end**
15 **end**

note that is the *seed* value overflows (meaning that the POSIT vector is a *sign* bit followed by a sequence of the bit), then the output POSIT would be rounded to $\pm\infty$ or zero, depending on the sign of the seed.

6.3 Division and subtraction

Subtraction can be implemented using the same logic as addition. Similarly, division can be regarded as a multiplication with the reciprocal of the dividing number:

$$A/B = A \cdot \frac{1}{B}$$

The POSIT format allows an easy transformation between a POSIT and its reciprocal value (as seen in Figure 5). The reciprocal is obtained by 2's complementing all exponent bits (*regime* + *exponent* field). To reciprocate the *fraction* a different logic should be implemented, such as translation tables (POSIT doesn't offer a fast method to reciprocate this field). As with IEEE 754 format, the reciprocal is only accurate up to the closest exponent of 2.

7 Issues encountered during the project

7.1 Packer representation of 1

The packer receives the value of its fields as three signed vectors which are identical to the unpacker's output (see subsubsection 6.1.1). This creates a problem in distinguishing between the POSIT representation of 1, 01000..., and 0, which is 0000..., as both numbers satisfy $seed = exponent = fraction = 0$.

The identical representation is the result of the hidden bit in the fraction part (since $fraction = 0$ actually represents the number 1.00...). We solved this problem by passing a flag as the *fourth* input to the packer which signal whether the POSIT vector represents a 0 or not. This way the we get

- packer(seed = 0, exponent = 0, fraction = 0, zero_flag=0) = 00000...
- packer(seed = 0, exponent = 0, fraction = 0, zero_flag=1) = 01000...

7.2 Non-synthesizable code

As was explained in section 5, most of the time the project was compiled using iVerilog tool to check for correctness. This tools, which compiles SystemVerilog, doesn't give any indication to whether the code is synthesizable or not.

After presenting the mid-semester presentation, the lab engineer offered to review our code. He Pointed out that some instructions in the code cannot be synthesized using real hardware. This led to a redesign or some of our modules. To make sure that the new code can be created using HW, for some of our modules we drew logic gate diagrams.

8 Implementation Verification

8.1 Arithmetic correctness

For each of the modules a test bench was written in SystemVerilog to test its correctness and output for special values. To test the mathematical correctness of the modules, each of the modules was implemented in Python as well and the output of the two version (SUV and Python) was compared.

Figure 10: Waveform representation of multiplier module

8.2 Waveform presentation

To check how the different signals in the modules react over time a waveform diagram was produced for every module (see Figure 10 for an example the multiplier module). They were especially useful in problems caused by signals that change undesirably and triggering code to run.

9 Conclusion

The tests show that when doing small number (between -8 and 8 in our tests) arithmetic using POSIT format, most of the bits are reserved for the *fraction* part, 26 bits out of a 32bit, while still leaving enough *exponent* bits to handle overflows.

Since more *fraction* bits give higher accuracy, this result proves the accuracy superiority of POSITs in comparison to the IEEE 754 format (as claimed by *gustavson* et al.[1]).

The added accuracy makes the POSIT more suitable for some workflows which work with small numbers, e.g. probability calculations, than the IEEE 754 format. POSIT also allows general purpose calculations thanks to its dynamic field widths. Since the implementation complexity of POSIT computational unit is the same of the IEEE 754 for most operations, this makes it a viable alternatives for some machines.

Nevertheless, the format shows some disadvantages compared to the current industry standard. As mentioned in subsection 6.3, reciprocating the *fraction* part of a POSIT is a non-trivial task, and might be implemented using translation tables. Regardless of the chosen method, the varying size if the *fraction* field, might make the reciprocating unit much more complex (and possibly less efficient) than its IEEE 754 counterpart.

Another issue, caused by the dynamic filed sizes, is the accuracy loss in high numbers. Since high numbers in POSIT representation have a small (or non-existent) *fraction* field, arithmetic using such numbers wouldn't have a good accuracy. This means that it POSIT format the following expression can be true:

$$2^{31} + 2^{-31} = 2^{31}$$

This might not be a very big issue, as this equation can be done with a 32bit vector only in POSIT format (its IEEE 754 counterpart only has 23 bits for the *exponent* field).

The new format should be checked with more workflows to verify that the claim made by *gustavson* at al.[1], regarding that arithmetic in programs is mostly done with number of the same range is true. If so, many workflows can gain by switching to the POSIT format in terms of accuracy and range.

References

[1] Gustavson, Yonemato *Beating Floating Point at its Own Game: Posit Arithmetic* http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf