Melodica: Posit Processing Unit

Report

submitted in partial fulfillment of the requirements
for the degree of

Bachelor & Master of Technology

by

Riya Jain
Roll No: 150110019

under the guidance of

Prof. Sachin Patkar

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

2020

Declaration of the Academic Ethics

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this thesis.

I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/ data/ fact/ source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have not been properly cited or from whom proper permission has not been taken when needed.

Riya Jain

(Roll No. 150110019)

Abstract

Nowadays, most technological advancements in the field of computational are being directed towards high precision arithmetic implementations. The IEEE 754-2008 compliant (floating-point) is being utilized extensively in the existing hardware arithmetic systems. Posit number system with its higher accuracy and parameterizable dynamic range has been claimed to vanquish the shortcomings of floating point number system.

I present to you a fully parameterizable posit arithmetic unit, `Melodica`. Melodica not only implements the basic arithmetic operations like Add, Sub, Mul, Div, Mac but also supports quire functionality. Melodica also features fused operations and type converters between posit, quire and float data types. The Posit ALU is integrated as an accelerator to Flute, a RISC-V based processor,. Further, Melodica is incorporated into Flute to form `Clarinet` framework. Clarinet is the `first-ever integration of quire to a RISC-V core` and the paper for the same has been archived in [1]. Clarinet enables the user to work in an environment where posits and floats co-exist and are seamlessly interchangeable. This allows the user to experiment with the two data types and find an optimum system for any scientific application.

Melodica has been extensively tested with SoftPosits [2] and extensive application study and bench-marking on computer vision kernels has been performed for Clarinet. ASIC synthesis of Clarinet and Melodica is performed on a 90 nm-CMOS Faraday process. Finally, based on our analysis and synthesis results, a quality metric is defined for different instances of Clarinet that gives us initial recommendations on the goodness of the instances. Clarinet-Melodica is an easy-to-experiment platform that will be made available in open-source for posit arithmetic empiricism.

Acknowledgments

I would like to extend my gratitude and regards to my guide, **Prof. Sachin Patkar**, for his consistent guidance and encouragement throughout my Dual Degree project. It was a great privilege and pleasure for me to do research under his supervision. I would also like to thank **Niraj Nayan Sharma** for his help throughout the project. I would also like to thank my colleagues working in HPC Lab for helping me with various difficulties. I would like to thank the entire Electrical Engineering department for allowing me to access its various facilities during the duration of my project.

Riya Jain
IIT Bombay

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Steady development in the paradigm of computation has led to an increase in demand for higher precision arithmetic units encouraging computer architects to dwell into reconsidering previously unquestioned design choices. The IEEE Standard for Floating-Point (IEEE 754) is most widely used in modern day computational systems. Though the number system is evidently a good standard for floating point number system, it has multiple short comings. A set of those shortcomings listed below [4][5].

- Its not mandatory for the floating point number system implementations using the same format to produce the same result due to the introduction of hidden guard digits to improve precision of the numbers that are rounded. Thus, inconsistent results can occur across different computing platforms.
- The basic mathematical laws of distributivity, commutativity and associativity are not applicable since rounding is performed after every individual operations of a specific calculation.
- A lot of the available bit patterns are utilised to represent an undefined or unrepresentable number i.e. Not-A-Number (NaN) value.

In 2017, Gustafson et al.[6][7] proposed a new representation for real numbers called Posits. Posits are claimed to offer a wider dynamic range or more accuracy, and contain fewer unused (e.g. Not-A-Number) states than IEEE-754 floating point formats. Whether this new proposed number system format will supplant or complement the current IEEE-754 depends on many aspects. One of these aspects is the impact Posits have on modern hardware, which is currently unknown, making it hard to reason about the gains and expenses that Posits offer compared to IEEE-754.

Due to the advantages of the posit number system, several academic and industrial research labs have started exploring and studying applications that can benefit due to posits. The *SoftPosit* library supports early-stage investigation of posit for different applications in software [2]. However, no such framework exists for hardware exploration.

There is a dire need for an easily re-configurable hardware platform for early-stage design space exploration of posit arithmetic for various applications.

Chapter 2

Posits

2.1 Floating Point v/s Unum number system format

IEEE 754 format is a set of representations of numerical values and symbols. The standard defines five basic formats ranging from the small 16-bit half-precision format, all the way up to a 256-bit octuple precision format.

The universal number(Unum), introduced by John L. Gustafson is aimed to overcome the blemishes of the IEEE 754 standard that has been in use for many years now. The Unum format has improved from type I to type II and further to type III (which is also known as Posit).

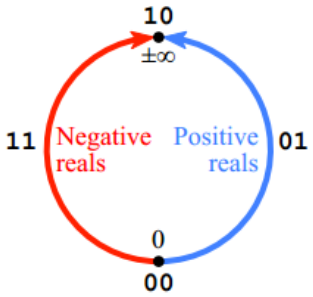


Figure 2.1: SORN, Subsets of the projective real numbers.

This initial candidate, originally called Unum (from now: Unum-I), introduced a variable-length format that includes a uncertainty-bit (called the u bit) that indicates whether the number is precise or within an interval.But it was also criticized for being hardware-unfriendly, mainly due to the variable length of the format. Following the criticism of Unum-I, Gustafson et al. proposed Unum-II. The type II specification introduces the Sets Of Real Numbers (SORN) as shown in Fig 2.1. The SORN is a loop that circles through all the numbers and connects $+\infty$ to $-\infty$. A SORN is therefore able to define a specific interval using subsets of the projective real numbers. Relaxing some of the mathematical properties of Unum-II finally led to POSITs. The advantages and disadvantages

of each Unum type is described in Table 2.1.

Unum	Advantages	Disadvantages
Type I	Exact/Open interval to 1 ULP, No approximation	Variable width, redundant forms, difficult hardware implementation
Type II	Maximum information per bit, perfect reciprocals, allows decimal representations	Hardware implementation is difficult and impractical
Type III	Hardware friendly, faster, more accuracy, lesser area than float	Too young to have business market

Table 2.1: Advantages and disadvantages of Unum

2.2 Posit Fields

In IEEE-754 format, we have a sign bit indicates if the number is negative or positive ,a constant number of exponent bits together with an implicit (format-dependent) bias indicate the range of the number, and the remaining fixed number of bits are allocated to the mantissa, which when combined with an implicit 1 yields a number between 1.0 and 2.0. The value for float is computed using equation 2.1

$$\texttt{float value} = (-1)^{sign} * 2^{exp-bias} * 1.mantissa \tag{2.1}$$

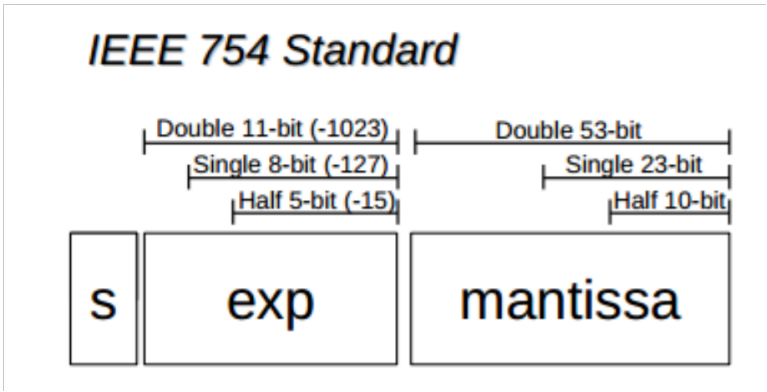


Figure 2.2: Float format

The POSIT format has four bit-fields: a sign bit indicating positive or negative numbers, a regime and exponent field that together represent the scale factor, and finally a mantissa or fraction as shown in Fig 2.2. The following table gives the division of the Posit.

Field Name	Field Representation	Field Size Representation	Field Size Bounds
Entire Posit	Posit	N	N
Sign	sign	s	1
Regime	k	r	[2,N-1]
Exponent	expo	ex	[0,min(es,N-1-r)]
Fraction	frac	f	[0,N-1-r-ex]

Table 2.2: Field representation and sizes

The POSIT format is defined by two (preset) variables: the size Posit (N) and the maximum width exponent (also called es). Thus the size of regime, exponent and fractional fields can all be variable for a given set of es and N as shown in Fig 2.3. Let posit val be the value of the datum, then: equation 2.2.

$$\texttt{posit val} = (-1)^{sign} * (2^{2^{es}})^{k} * 2^{expo} * 1.frac \tag{2.2}$$

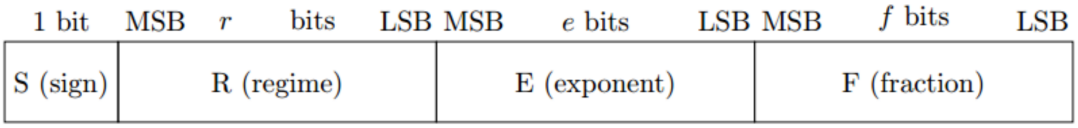


Figure 2.3: Posit format

The posit format has four fields: a sign bit indicating positive or negative numbers, a regime and exponent field that together represent the scale, and finally, a fraction.

- Sign (`s`): The MSB of the number. If the bit is set, the posit value is negative. In this case all remaining fields are represented in two's complement notation.
- Regime Field (`r`): The regime is used to compute the scale factor, `k`. In a posit number this field starts just after the sign bit and is terminated by a bit opposite to its leading bits. The computation of `k` is as per the equation 2.2, where l is the number of leading number of bits, $1 \leq l \leq N-2$.

$$k = \begin{cases} l-1, & \text{if regime has leading ones} \\ -l, & \text{if regime has leading zeros} \end{cases}$$

- Exponent Field (`exp`): The exponent begins after the regime field and the maximum width of the exponent field is `es`.
- Fraction Field (`f`): The remaining number of bits after the exponent make up the fraction. The fractional field is preceded by an implied hidden bit which is always

1.

2.3 Bit Size Bounds for different fields of posit

We look at the bounds for the various fields in the POSIT format. Recall that a POSIT number of size N is made of four bit-fields. Out of these four bits-fields, only the s is constant having value 1 while all other fields sizes are a function of the r which has variable size due to the way it is encoded.

The bounds for regime field is $2 \leq r \leq N-1$ because there will be an immediate bit flip of leading bit (giving $r = 2$ as the minimum) or there may not be bit flip of the leading bit where regime-bits occupy nearly the entire width of the Posit other than the sign bit (giving $r = N-1$ as the maximum).

Similarly, despite the fact that the number of exponent bits are fixed for a certain POSIT format, these can eventually end up having lesser than es bits or even not being present in the representation.The bounds for exponent field is $ex = min(es, N-1-r)$ because there will be no exponent field if regime field has no bit flip of the leading bit (giving $ex = 0$, $r = N-1$ as the minimum) or there is bit flip of the leading bit but the number of bits left to form the exponent field is a function of the number of bits used to form the regime field(giving $ex = min(es, N-1-r)$) or there is bit flip and more than es number of bits are remaining to form the exponent and fraction field then the exponent field cannot have size more than es.

The bounds on the fractional bits are arguably the most important and impact-full, since they decide the magnitude of the basic integer operations within the compute units. The bounds for fraction field is $f = N-1-r-ex$ because the remaining number of bits after forming the sign, regime and exponent field belong to the fraction field.

2.4 NaN Representations

The IEEE 754 standard represents numbers such as NaN or $\pm\infty$ by having the exponent field set to one, while the fraction field then determines the exception cases.
The posit number format employs a different view on the use of NaN values. Instead of encoding different types of NaN values as done in floating point system, the posit arithmetic framework doesn't allow any bits to be used for their representation. Therefore, in Posit Arithmetic whenever a case for NaN is encountered an interrupt should be asserted to handle these exception cases.Moreover, Posits have a single representation for the value $\pm\infty$ whereas IEEE floats have separate values for ∞ and -∞. Examples:

- $\pm\infty + \pm\infty$

- $\pm\infty - \pm\infty$
- 0 x ∞
- $\pm\infty$ x 0
- 0 / 0
- $\pm\infty / \pm\infty$

2.5 Quire and its Fields

A Quire is a huge fixed size scratchpad that can be used to perform fused operations without the need to round the intermediate values which can help reduce loss of precision in the resultant value. In posit arithmetic, the quire data type is accessible to the programmer through fused operations.

John Gustafson claims that this will allow us to significantly reduce the the size of posit representation to get the same dynamic range and accuracy of inputs. If we are able to reduce the size of representation of input, it will be beneficial for several reasons: (I) reduction in format-size leads to a reduction in silicon real estate cost for Floating-Point Units (FPUs), allowing the freed silicon to be spent on more compute (wider vector, more cores)or on-chip store (larger caches), (II) reduction in format size leads to more compute per unit bandwidth (more values fetched per unit bandwidth), and (III) reduction in format-size leads to lower power- and energy-consumption of FPUs. But for larger hardware implementations using quire support is not expected to be cheap due to the huge size of quire.

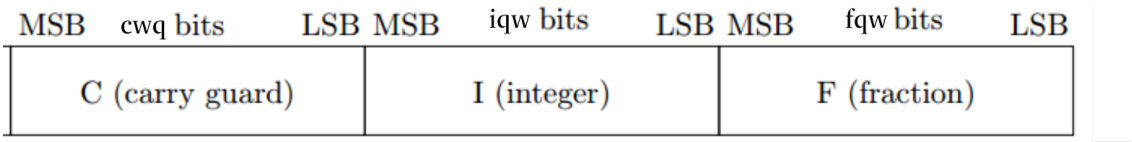


Figure 2.4: Quire format

For an N posit the quire size is fixed. The Quire format is a fixed-point 2's complement that has four bit-fields: a sign bit indicating positive or negative numbers, carry field that stores the overflow integer bits, the integer field and fraction fields. Unlike the posit representation the quire doesn't have a scale field. For NaN representation in Quire the sign bit is 1 and all the other quire fields bits contain only zeros which is equivalent to its representation of $\pm\infty$. The Sizes of the individual fields in quire is fixed unlike posits. The Table 2.3 gives the division of the Quire for a given Posit size.

Field Name	Field Size Representation	Field Size
Entire Posit	N	N
Entire Quire	qw	$\frac{(N)^2}{2}$
Sign	1	1
Carry	cqw	(N) - 1
Integer	iqw = fqw	$\frac{(N)^2}{4} - \frac{(N)}{2} = nq$
Fraction	fqw	$\frac{(N)^2}{4} - \frac{(N)}{2} = nq$

Table 2.3: Quire field Sizes for an N-bit Posit

2.6 Ubit

A ubit which is the uncertainty bit is appended to the fraction field. The ubit is 0 for all the values that can be derived from equation 2, and 1 for all the open intervals between the values that can be derived from equation 2. The ubit is defined for each tile, giving the flexibility to control both ends of the intervals independently.

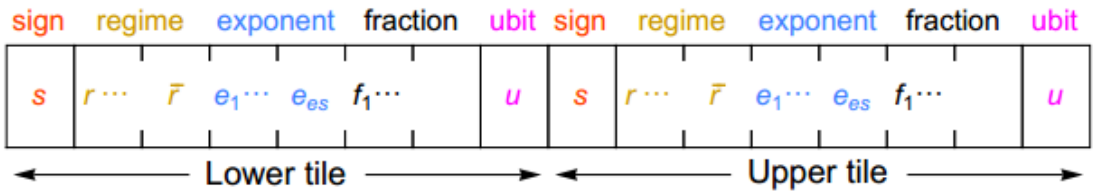


Figure 2.5: Ubit Representation

Each number is now determined with the help of two tiles. Conventional interval arithmetic is in the form of closed intervals [a, b] where a and b are floats, and a $\leq$ b. To find the interval one starts from the lower tile and moves in the counterclockwise direction until the upper tile is reached, it wraps around the circle, crossing $\pm\infty$ if necessary.

2.7 Dynamic range and precision

The dynamic range and precision of a posit number depends on the value of es and obviously the number of bits in given posit. As es value increases, the contribution of the regime and exponent bits will become larger, and so larger ranges of values can be attained and represent. So increasing es increases dynamic range. Dynamic range, measured in decades, is the log base 10 of the ratio between the largest and smallest representable positive values. We look into the maximum and minimum value that can be represented. These values will reach same absolute value in the positive and negative domain as can be seen from the SORN. So we look into the positive domain and extrapolate the value

in the negative domain.

The largest representable finite posit value is labeled maxpos. This value occurs when k is as large as possible i.e. when all the bits after the sign bit are 1's. In this case regime field maximum width = (N-1) and so k(regime field value) = (N-2). So maxpos equals:

$$\texttt{maxpos} = (2^{2^{es}})^{N-2} \tag{2.3}$$

The smallest representable positive number, minpos, occurs when k is as negative as possible, i.e. when the largest possible number of bits after the sign bit are 0's. They can't all be zeros or else we have the representation for the number 0, so there must be a 1 on the end. In this case regime field maximum width = (N-2) and k = (2-N).

$$\texttt{minpos} = (2^{2^{es}})^{2-(N)} = 1/maxpos \tag{2.4}$$

However, increasing es value means decreasing the number of bits available to the fraction, and so decreases precision of posit . So, one of the benefits of posit numbers is this ability to pick a desired es keeping in mind the trade-off between dynamic range and precision to meet your needs. The "golden zone" [3] defined as shown in the figure is where the Posits have better accuracy than floats. Posits are a good option for Machine learning applications since these involve convolutions or neuron weight summations where the summation terms are small enough to keep the resultant value in the golden zone. Moreover, we can always scale the values to ensure they lie in the more accuracy region.

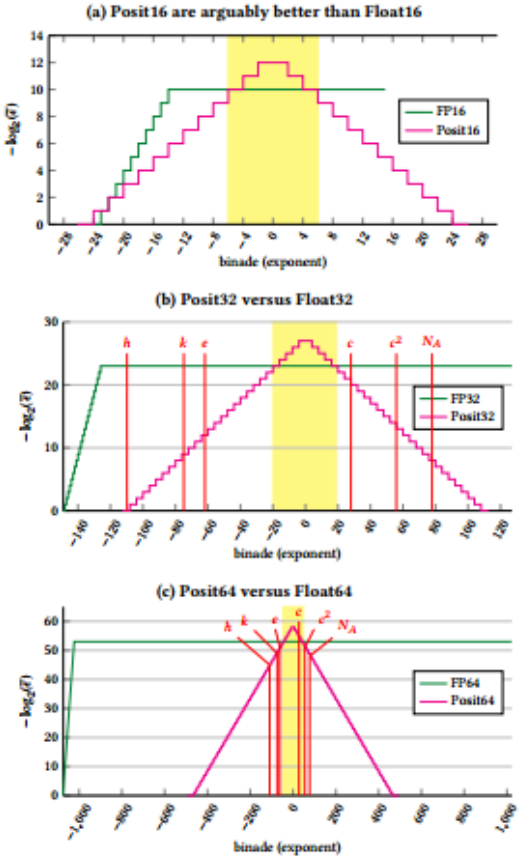


Figure 2.6: Golden zone representation [3]

2.8 Rounding

Rounding is basically the way by which u can represent a real number which can't directly be mapped to a expressible posit of a particular length. The results of all operations are regarded as mathematically exact prior to rounding. So the rounding conditions followed by Posits for a real number value say p are:

1. If p can exactly be represented in posit number system, the value is not rounded
2. If p has its value such that $|p| > maxpos$, p is rounded to $sign(p) * maxpos$
3. If p has its value such that $|p| < minpos$, p is rounded to $sign(p) * minpos$
4. For all other values, the posits are rounded using only one rounding mode which is round-to-nearest with tie-to-even i.e. round the value of p to whichever posit is closer but if two posits are equally near, the one with binary encoding ending in 0 is selected.

2.9 Standard Posit Types

There are a few standard sizes defined in equivalence to Floating Point numbers. They are described in Table 2.4

property	Posit 8	Posit 16	Posit 32	Posit 64
`N`	8	16	32	64
Max Significand bits	6	13	28	59
`es`	0	1	2	3
`minpos`	$2^{-6} \approx 1.5x10^{-2}$	$2^{-28} \approx 3.7x10^{-9}$	$2^{-120} \approx 7.5x10^{-37}$	$2^{-496} \approx 4.9x10^{-150}$
`maxpos`	$2^{6} \approx 6.4x10^{1}$	$2^{28} \approx 2.7x10^{8}$	$2^{120} \approx 1.3x10^{36}$	$2^{496} \approx 2.0x10^{149}$
Number of quire bits (`qw`)	32	256	512	2048
Exact sum quire limit	32767	$2^{43} - 1$	$2^{151} - 1$	$2^{559} - 1$

Table 2.4: Standard Posit Sizes

2.10 Related work

Since the inception of posit data representation and arithmetic, there have been several implementations of posit arithmetic in the literature. The early and open-source hardware implementations of posit adder and multiplier were presented in [8] and [9]. In [9], the authors have covered the design of a parametric adder/subtractor, while in [8], the authors have presented parametric designs of float-to-posit and posit-to-float converters, and multiplier along with the design of adder/subtractor. The PACoGen open-source framework that can generate a pipelined adder/subtractor, multiplier, and divider are presented in [10]. The PACoGen is capable of generating the hardware units that can adapt precision at run-time. A more reliable implementation of a parametric posit adder and multiplier generator is presented in [11]. A major drawback of the generator presented in [11] is that it is a non-pipelined design resulting in low operating frequency for large bit-width adders and multipliers.

Cheetah presented in [12] discusses the training of deep neural network (DNN) using posits. I believe that the architecture presented in [12] is promising and some of the features can be incorporated in Melodica in the future.

Apart from the mentioned efforts, there have been several other implementations of posit hardware units [13][14]. More recently, [15] integrated a posit numeric unit as a functional unit with the Shakti C-Class RISC-V processor. The implementation does not support quire and reuses the floating-point infrastructure (including register file) to implement posit arithmetic. This limits the system to using 32-bit or 64-bit posits.

Unfortunately, none of the previous efforts are directed toward the consolidation of posit research. Further, they do not include an easy-to-use software framework which

allows floating-point and posit types to cohabit in an application cleanly. We see here a need and an opportunity to consolidate the research in the domain of computer arithmetic by providing an open-source test-bed, Clarinet. We also address the need for a software framework by introducing a programming model that allows floating-point and posit types to coexist as independent types in an application.

Chapter 3

Melodica: Posit Arithematic in Hardware

Melodica is a parameterizable posit arithmetic unit implemented using BSV HL-HDL. Melodica accepts three high-level parameters: the posit-width (`N`), the maximum width of the exponent field (`es`) and float-width (`FW`). For an `N`-bit Melodica architecture $\frac{N^2}{2}$ (`qw`) sized quire is integrated with the operation pipelines as a special-purpose register. Depending on the size of `N`, it is possible that the quire may not be sized to a multiple of byte. It delivers accumulator functionality using posit fused-multiply-accumulate (FMA), fused-multiply-subtraction (FMS), fused-divide-accumulate (FDA), fused-divide-subtraction (FDS) into the quire with basic operations like addition, subtraction, multiplication and division, and is meant to be used alongside a single-precision floating point implementation for all other compute operations. In addition to the FMA computation Melodica implements a complete set of type-converters between floating-point and posit formats, and between quire and posit formats.

Melodica's organization is illustrated in Fig. 3.1. There are three computational steps involved in Melodica's operation: i) *extract:* interpret the posit operands to extract the sign, regime, exponent, fraction fields and infinite/zero flag, ii) *operate:* perform the appropriate mathematical operation using one or more of the extracted posits or float operand, and iii) *normalize:* convert the output posit-fields back into an `N`-bit posit word [16].

3.1 Extract

The extractor unpacks posit operand into sign, scale and fraction bit fields, essentially converting from a format with variable-width fields to one with fixed-width fields. This conversion is essential for the subsequent pipelines to efficiently compute on posit fields. The scaling factor, `scale`, is determined using the regime field and the exponent field as given by equation 2.2, where maximum posit scale width is `psw` and maximum posit fraction width is `pfw`. The maximum value of regime field is equal to $N2$ and for exponent

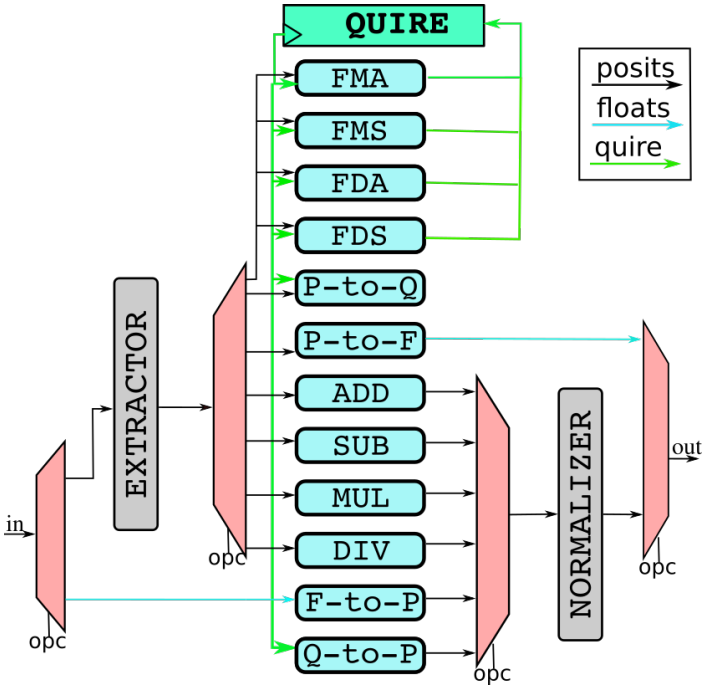


Figure 3.1: Melodica

field $2^{es}1$, the total number of bits required to represent any scale is equal to psw. The minimum size of r is 2 and the es bits for exponent field so the number of bits required to represent any fraction is equal to pfw.

$$\begin{aligned} scale &= (2^{2^{es}})^k * 2^{exponent} \\ &= 2^{2^{es}*k+exponent} \\ \texttt{psw} &= log_2((N-1)*(2^{es}-1))+1 \\ \texttt{pfw} &= N-3-es \end{aligned} \tag{3.1}$$

Extraction operates on the `N`-bit input posit word and generates four outputs as illustrated in Fig. 3.2. The steps involved are:

1. Check for special cases like 0 and $\pm\infty$ to determine the zero-infinity flag (zif). If the sign of the posit number is negative, perform a two's complement of the remaining `N-1` bits.

2. Compute regime field using equation 2.2. The regime field in general ends with a flipping bit but in the case when the number of exponent and fraction bits are zero then there may not be a flipping bit.

3. Determine the value of the exponent. The exponent field may have up to `es` bits. I multiplex between the two cases when its field size is exactly `es`, and when size is variable (lesser than `es`). In the latter case the location of the exponent field continues until the end of the posit and it forms the MSB part of the exponent.

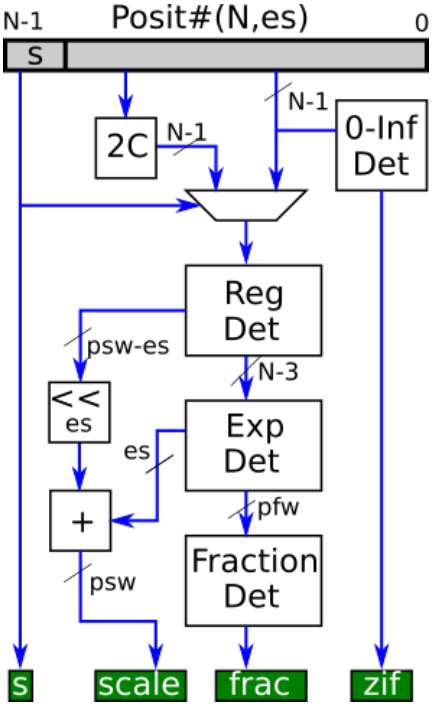


Figure 3.2: Extractor

4. Calculate `scale` using equation 3.1. Calculate the fraction field by extracting remaining bits (if any) after the exponent field.

Blocks in Fig. 3.2 are 2C : 2's complement, 0-Inf Det : zero and infinity detection, Reg Det : Regime Detection, << es : left shift by es bits, Exp Det : Exponent Detection. Sizes of inputs and outputs from the block are mentioned along the arrows. N bit posit input and data structure of s(sign), scale, frac(fraction) and zif (zero-infinity flag) forms the output.

3.2 Normalize

The normalization illustrated in Fig. 3.3, is the reverse operation of extraction. It constructs a posit value from the constituent fields available after computation on the operands. There may be a loss in accuracy due to the rounding of fractional bits. The four steps involved in normalization are:

1. Computation of regime field and the exponent bits from the `scale` value based on the equation 3.1 i.e. k is calculated as scale/2es, while the exponent is determined by the remainder (scale mod 2es).

2. Construction and concatenation of the regime field and exponent fields where regime bits are calculated based on run-length of 0s and 1s

3. Shift the fraction field by shifting regime field and exponent field. The concatenated

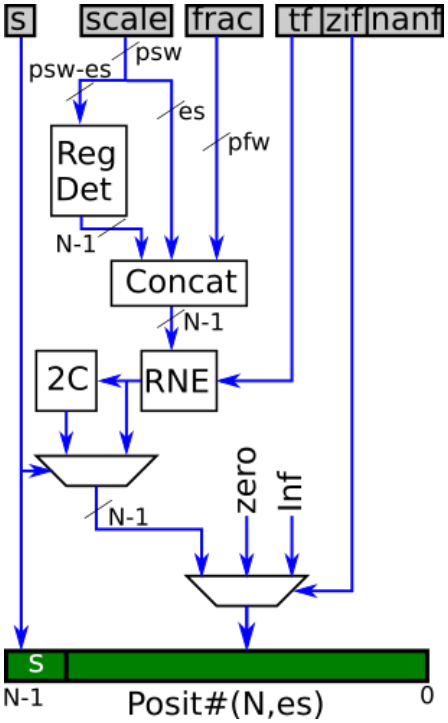


Figure 3.3: Normalizer

value is rounded to the nearest even depending on the fraction bits truncated in the previous stage and the truncation flag (`tf`)

4. Check for special cases (zero, $\pm\infty$ and NaN). If the sign bit is set, the final value is the two's complement of the remaining `N-1` bits.

Blocks in Fig. 3.3 are Reg Det : Regime Detection, Concat : Concatenate the two entries, 2C : 2's complement, RNE : Round-to-nearest even, Inf,zero : posit bit pattern corresponding to infinity, zero respectively. Sizes of inputs and outputs from the block are mentioned along the arrows. N bit posit output and data structure of s(sign), scale, frac(fraction) tf(truncation flag), zif(zero-infinity flag) and nanf(Nan flag) forms the input.

3.3 Basic Operations

The basic arithmetic operations include Addition, Subtraction, Multiplication, Division and multiply-accumulate.

3.3.1 Add and Subtract

Addition and subtraction is conceptually very similar to how they are performed in the IEEE-754 format, but with the extensions of extraction and normalization the value. The adder consists of two input operands OP1 and OP2, one output = OP1 + OP2. We start by identifying if the expect operation is a subtraction; if it is, is is inverted

(by taking 2s complement) the second operand (OP2) and proceed with adding them ($OP1 - OP2 = OP1 + (-OP2)$) using the adder.

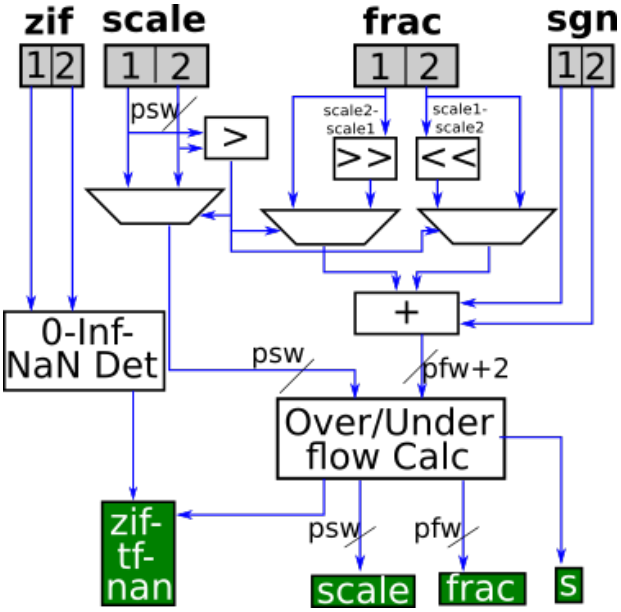


Figure 3.4: Adder

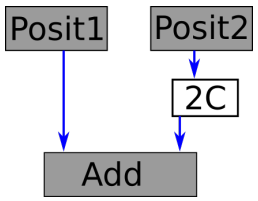


Figure 3.5: Subtracter

Addition/Subtraction is a Three staged pipeline. Consider only looking at adder (as shown in Fig 3.4) because subtracter (as shown in Fig 3.5) is 2s complement followed by adder. The input operands need to be extracted using the Extractor before entering the Adder/Subtracter and the final Adder/Subtracter output has to be normalised into a Posit. Stages involved are :

1. The hidden bit is pre-added to the input fraction fields depending on the posit value being zero or otherwise. Also check for NaN and $\pm\infty$ corner cases.

2. - Match the scale of both operands by shifting , the smallest operand right to match the scale of the two input posit operands. Therefore, after determining the larger operand, the smallest operand's fraction bits are shifted right by the difference of the scales of the two operand.
   - But the fraction field of the smallest operand might lose lsb side bits due to the shifting performed to match both operand scales. Hence, a truncated flag

is introduced by the adder that can be retained until rounding is performed and use it when result needs to be normalized.

3. Both operand fractions are added (equal or unequal operand signs) using an signed integer adder.

4. Modify the scale to accommodate for the shifting of fraction bits back into the 1.xxx form and remove the hidden bit.

Blocks in Fig. 3.4 and Fig. 3.5 are $>$: control signal generation after comparing the two inputs, $>>$, $<<$: right and left shift, 0-Inf-NaN Det : zero, infinity and NaN detection, Over/underflow Calc : Calculate the Overflow and Underflow with respect to the maximum attainable scale and fraction values, 2C : 2's complement. Sizes of inputs and outputs from the block are mentioned along the arrows. Data structure of sgn(sign), scale, frac(fraction), zif(zero-infinity flag) each for the two input posits and data structure of s(sign), scale, frac(fraction) tf(truncation flag), zif(zero-infinity flag) and nanf(Nan flag) forms the output.

3.3.2 Multiply

Multiply (MUL) is an instruction designed to compute the product of two input numbers. The input operands need to be extracted using the Extractor before entering the Multiplier (as shown in 3.6) and the final Multiplier output has to be normalised into a Posit. Stages involved in a posit multiplier is as follows:

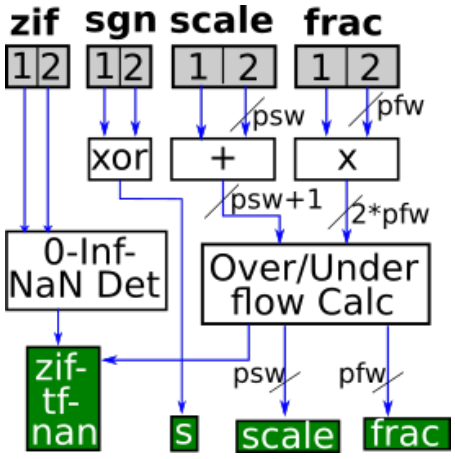


Figure 3.6: Multiplier

1. The hidden bit is pre-added to the input fraction fields depending on the posit value being zero or otherwise. Also check for NaN and $\pm\infty$ corner cases.

2. The fraction field of the input posit operands are multiplied using an unsigned integer multiplier. The resultant sign is computed by the xor of the input sign fields of the operands.

3. The scale of the output product is determined by signed addition of the scales of both input operands. This scale is then compensated in case of an overflow or underflow in multiplied fraction fields.

4. There may be an overflow in the scale bits too. So, the resulting product fraction is shifted in order to obtain the normalized form 1.xxxx and further the scale over flow is compensated.

5. But the fraction field of the smallest operand might lose lsb side bits due to the shifting performed to match both operand scales. Hence, a truncated flag is introduced by the adder that can be retained until rounding is performed and use it when result needs to be normalized.

6. Modify the scale to accommodate for the shifting of fraction bits back into the 1.xxx form and remove the hidden bit.

Blocks in Fig. 3.6 are 0-Inf-NaN Det : zero, infinity and NaN detection, Over/underflow Calc : Calculate the Overflow and Underflow with respect to the maximum attainable scale and fraction values. Sizes of inputs and outputs from the block are mentioned along the arrows. Data structure of sgn(sign), scale, frac(fraction), zif(zero-infinity flag) each for the two input posits and data structure of s(sign), scale, frac(fraction) tf(truncation flag), zif(zero-infinity flag) and nanf(Nan flag) forms the output.

3.3.3 Divide

Divide (DIV) is an instruction designed to compute the product of two input numbers. The input operands need to be extracted using the Extractor before entering the Divider (as shown in 3.6) and the final Divider output has to be normalised into a Posit. Stages involved in a posit divider is as follows:

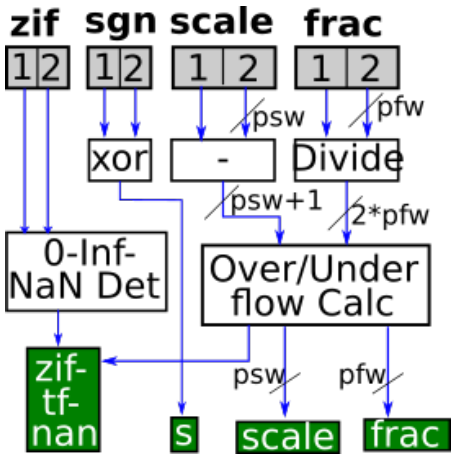


Figure 3.7: Divider

1. The hidden bit is pre-added to the input fraction fields depending on the posit value being zero or otherwise. Also check for NaN and $\pm\infty$ corner cases.

2. The fraction field of the input posit operands are divided using an unsigned integer divider made using the FSM shown in Fig 3.8. The user can pre-define the number of accuracy bits it requires in the divider. But to not lose any bits in the fraction a minimum quotient of pfw should be ensured. The resultant sign is computed by the xor of the input sign fields of the operands.

3. The scale of the output product is determined by signed addition of the scales of both input operands. This scale is then compensated in case of an overflow or underflow in divided fraction fields.

4. There may be an overflow in the scale bits too. So, the resulting product fraction is shifted in order to obtain the normalized form 1.xxxx and further the scale over flow is compensated.

5. But the fraction field of the smallest operand might lose lsb side bits due to the shifting performed to match both operand scales. Hence, a truncated flag is introduced by the adder that can be retained until rounding is performed and use it when result needs to be normalized.

6. Modify the scale to accommodate for the shifting of fraction bits back into the 1.xxx form and remove the hidden bit.

Blocks in Fig. 3.7 are 0-Inf-NaN Det : zero, infinity and NaN detection, Over/underflow Calc : Calculate the Overflow and Underflow with respect to the maximum attainable scale and fraction values, Divide : Integer divider using fsm in Fig. 3.8. Sizes of inputs and outputs from the block are mentioned along the arrows. Data structure of sgn(sign), scale, frac(fraction), zif(zero-infinity flag) each for the two input posits and data structure of s(sign), scale, frac(fraction) tf(truncation flag), zif(zero-infinity flag) and nanf(Nan flag) forms the output. In Fig. 3.8 denom : denominator, numer : numerator, quo : quotient, rem : remainder, denom2 : shifted denominator, n : is to calculate the shift. The inputs for the integer divider are denominator and numerator while outputs are quotient and remainder.

3.3.4 Multiply Accumulate

Multiply-Accumulate (MAC) is an instruction where to the product of two input operands is added with the third input to get an accumulated value. The MAC block uses the above mentioned Addition and Multiplication blocks. In this instruction the result of the product is not rounded, followed by an addition to get the output accumulated value, after which rounding is performed. This procedure is alternatively called double rounding.

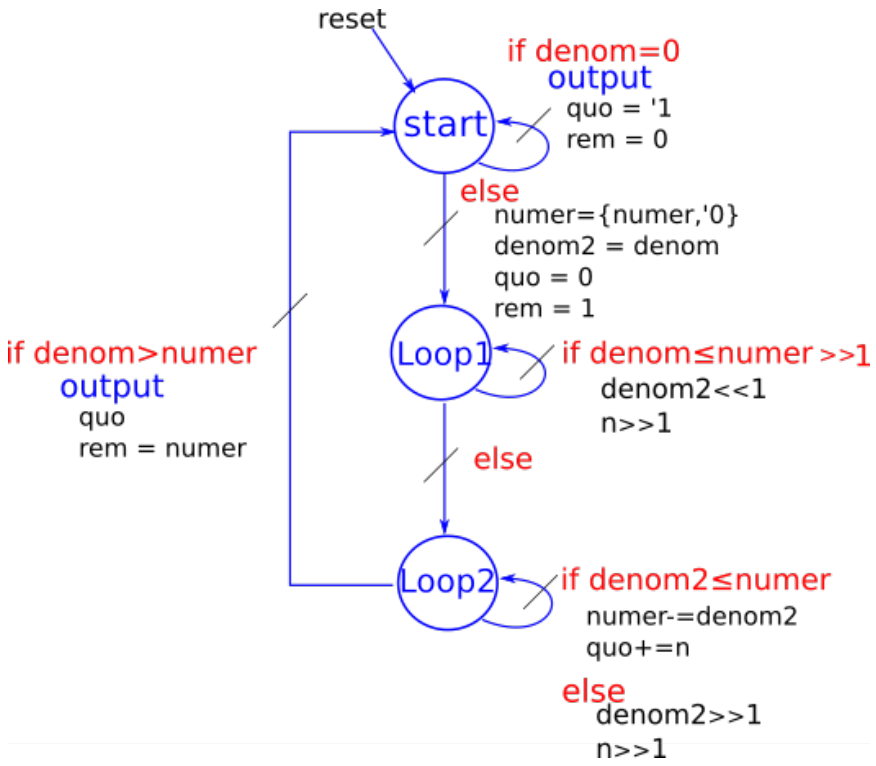


Figure 3.8: FSM diagram for Divide

3.4 Fused Operations

Fused operations are instructions that takes only two posit numbers as input, performs two or more operations on them and is not rounded until the entire expression is computed using the quire. The fused operations include Fused Multiply Accumulate (FMA), Fused Multiply Subtract (FMS), Fused Divide Accumulate (FDA) and Fused Divide Subtract (FDS).

Lets first look at why FMA is beneficial over MAC. Recall that in multiplier to compensate the overflow scale the output fraction bits may be shifted or in adder in order to match the scale of both input operands, the fraction bits of the smaller operand is truncated due to shift in fraction field. It is therefore possible that multiple fraction bits are lost even before the fraction field addition. This is unacceptable when the motive for designing an implementation is able to achieve better accuracy than its preceding implementations. In order to avoid any input information to be discarded, the use of accumulator is proposed. This instruction uses the quire as the accumulator.
The quire can be seen as a scratchpad register that can be used for different fused operations. The size of this quire should be wide enough to eliminate the need of having to perform rounding of intermediate results. While existing scratchpad are not controllable by the user, a quire register will be convenient for the user to manipulate. Hence, the user can instantiate a quire and use the quire to perform any operation without rounding

of results.

3.4.1 Fused Multiply Accumulate (FMA) and Fused Multiply Subtract (FMS)

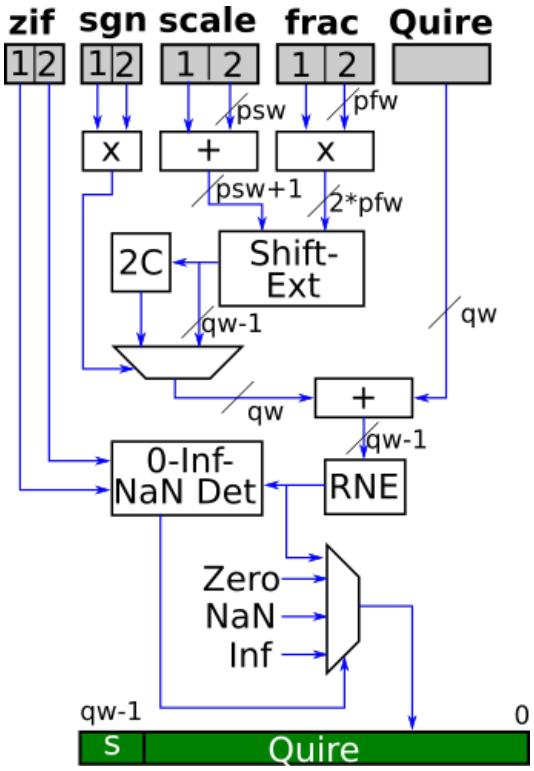


Figure 3.9: Fused Multiply Accumulate (FMA)

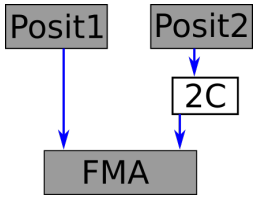


Figure 3.10: Fused Multiply Subtract (FMS)

FMA operation performs the following calculation: $Quire = Quire + X * Y$, while FMS operation performs the following calculation: $Quire = Quire - X * Y$, where Quire denotes the contents of the quire and X,Y denote the input posits. We start by identifying if the expect operation is a FMS; if it is, it is inverted by taking 2s complement of the second operand (OP2) and proceed with multiplying them $(-(X * Y) = X * (-Y))$ using the multiplier. So, Fused Multiply Accumulate (FMA) and Fused Multiply Subtract (FMS) use the same hardware except that FMS requires an extra N-bit 2's complement for the second input.

The FMA, illustrated in Fig. 3.9 computes the product of two input posit numbers

and adds the result to the quire. The FMS, illustrated in Fig. 3.10 computes the product of two input posit numbers and subtracts the result from the quire. Using quire as an accumulator helps preserve the overflow or underflow bits without the need to round the results. Each of the input operands need to be extracted using the Extractor before entering the FMA block. The FMA is performed as follows:

- The hidden bit is pre-added to the input fraction fields depending on the posit value being zero or otherwise. Also check for NaN, 0 and $\pm\infty$ corner cases.
- The fraction field and sign fields are multiplied using integer multipliers, and the scales are added to create the `scale` of the output. There is no rounding
- The product fraction is shifted using the new scale value to align with the quire's integer and fraction fields. If the product is negative, appropriate twos complement is performed.
- Quire is added to the product of the operands using signed addition, and if there is overflow or underflow, the sum is rounded to the nearest even.

Blocks in Fig. 3.9 and Fig. 3.10 are shift-Ext : shift and accommodate scale to get integer and fraction part equivalent for quire, 0-Inf-NaN Det : zero, infinity and NaN detection, 2C : 2's complement, RNE : Round-to nearest- even. Sizes of inputs and outputs from the block are mentioned along the arrows. Data structure of sgn(sign), scale, frac(fraction), zif(zero-infinity flag) each for the two input posits and quire as the output.

3.4.2 Fused Divide Accumulate (FDA) and Fused Divide Subtract (FDS)

FDA operation performs the following calculation: $Quire = Quire + X/Y$, while FDS operation performs the following calculation: $Quire = Quire - X/Y$, where Quire denotes the contents of the quire and X,Y denote the input posits. We start by identifying if the expect operation is a FMS; if it is, it is inverted by taking 2s complement of the second operand (OP2) and proceed with dividing them ($-(X/Y) = X/(-Y)$) using the divider. So, Fused Divide Accumulate (FDA) and Fused Divide Subtract (FDS) use the same hardware except that FDS requires an extra N-bit 2's complement for the second input.

The FDA, illustrated in Fig. 3.11 computes the division of two input posit numbers and adds the result to the quire. The FMS, illustrated in Fig. 3.12 computes the division of two input posit numbers and subtracts the result from the quire. Using quire as an accumulator helps preserve the overflow or underflow bits without the need to round the results.Each of the input operands need to be extracted using the Extractor before entering the FMA block. The FDA is performed as follows:

- The hidden bit is pre-added to the input fraction fields depending on the posit value being zero or otherwise. Also check for NaN, 0 and $\pm\infty$ corner cases.

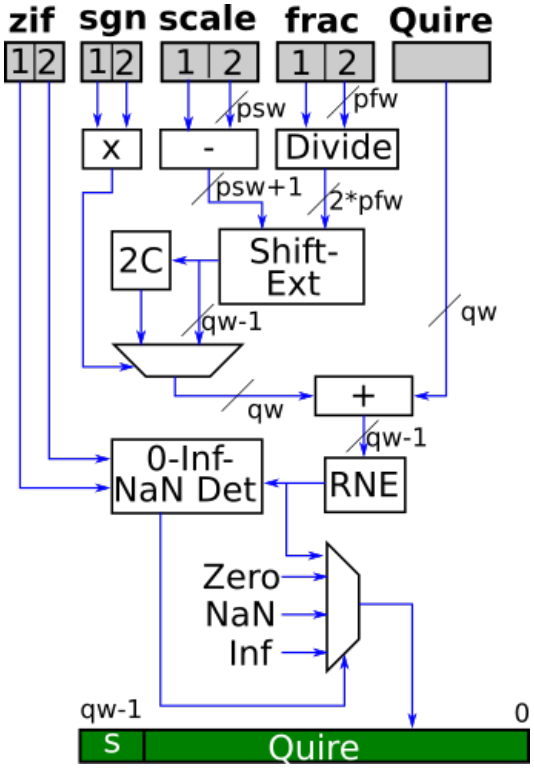


Figure 3.11: Fused Divide Accumulate (FDA)

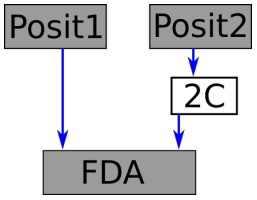


Figure 3.12: Fused Divide Subtract (FDS)

- The fraction fields are divided using integer divider formed by the FSM described in Fig 3.8. The user can pre-define the number of accuracy bits it requires in the divider. The sign fields are xored using 1-bit xor. The scales are subtracted to create the `scale` of the output. There is no rounding
- The product fraction is shifted using the new scale value to align with the quire's integer and fraction fields. If the product is negative, appropriate twos complement is performed.
- Quire is added to the product of the operands using signed addition, and if there is overflow or underflow, the sum is rounded to the nearest even.

Blocks in Fig. 3.11 and Fig. 3.12 are shift-Ext : shift and accommodate scale to get integer and fraction part equivalent for quire, 0-Inf-NaN Det : zero, infinity and NaN detection, 2C : 2's complement, RNE : Round-to nearest- even, Divide : integer divider using fsm in 3.8. Sizes of inputs and outputs from the block are mentioned along the arrows. Data

structure of sgn(sign), scale, frac(fraction), zif(zero-infinity flag) each for the two input posits and quire as the output.

3.5 Type Converters

3.5.1 Float-to-Posit (F-to-P)

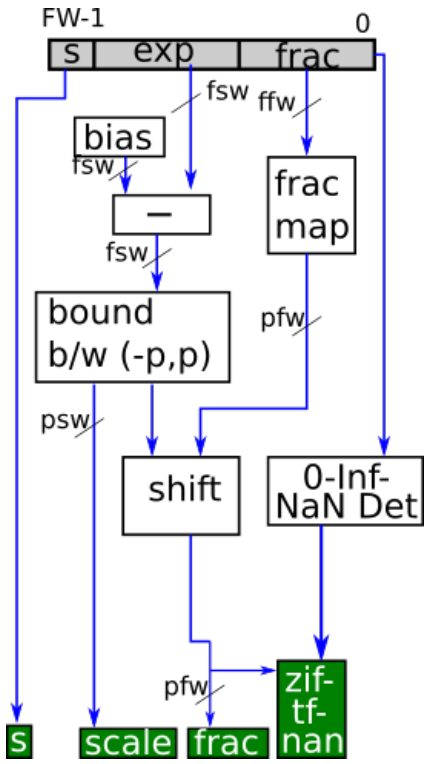


Figure 3.13: Float-to-Posit (F-to-P)

The F-to-P block converts a floating point input to posit format as depicted in Fig. 3.13. The conversion may result in a loss of precision when using narrower posit types. For a number represented in the posit format, its value is as per equation 3.2, where `fsw` is the exponent width of the float and `ffw` is the fraction width of float.
The steps involved in F to P conversion are:

- The fixed length fraction field field is interpreted directly from the input operand and mapped to the field in posit format using equation 2.2 and 3.2.
- The `scale` field after subtracting the bias is bounded between `(-p, p)`, where p equals $(N-2) * 2^{es}$.
- A truncation flag (`tf`) is asserted by the block depending on conversion between different size float and posit values. These flags are retained to perform rounding in later stages.

- In addition the converter also checks for special cases for the target format (zero, NaN, and $\pm\infty$).

The output of the F-to-P needs normalization before write-back.

$$value = (-1)^{sign} * 2^{exp-bias} * 1.f \tag{3.2}$$

Blocks in Fig. 3.13 are frac map : map fraction fields for float and posit and , bound b/w (-p,p) : bound between (-p,p), 0-Inf-NaN Det : zero, infinity and NaN detection,. Sizes of inputs and outputs from the block are mentioned along the arrows. Data structure of sgn(sign), exp(exponent), frac(fraction), zif(zero-infinity flag) each for the input float and data structure of s(sign), scale, frac(fraction), tf(truncation flag), zif(zero-infinity flag) and nanf(Nan flag) forms the output.

3.5.2 Posit-to-Float (P-to-F)

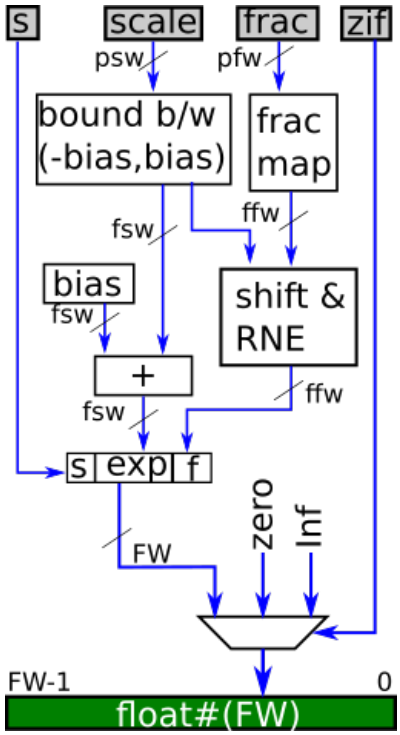


Figure 3.14: Posit-to-Float (P-to-F)

The P-to-F block converts a posit input to float format as illustrated in Fig. 3.14. Depending on the configuration parameters for Melodica this operation may result in a change in widths between source and target types. For a number represented in the posit format, its value is as per equation 3.2, where `fsw` is the exponent width of the float and `ffw` is the fraction width of float.
The input of the P to F block has to be extracted before conversion to floating point number. The steps involved in P to F conversion are:

- The fixed length fraction field field is interpreted directly from the input operand and mapped to the field in posit format using equation 3.2 and 2.2.
- The `scale` field is bound between (-bias, bias) for the target float format, and truncation of the fraction field if the field-width for the target format is narrower than the source format using equation 2.2 and 3.2.
- A truncation flag (`tf`) is asserted by the block depending on conversion between different size float and posit values. These flags are retained to perform rounding in later stages.
- In addition the converter also checks for special cases for the target format (zero, NaN, and $\pm\infty$).

Blocks in Fig. 3.14 are frac map : map fraction fields for float and posit and , bound b/w (-p,p) : bound between (-p,p), 0-Inf-NaN Det : zero, infinity and NaN detection,. Sizes of inputs and outputs from the block are mentioned along the arrows. Data structure of sgn(sign), exp(exponent), frac(fraction), zif(zero-infinity flag) each for the input posit and float forms the output.

3.5.3 Quire-to-Posit (Q-to-P)

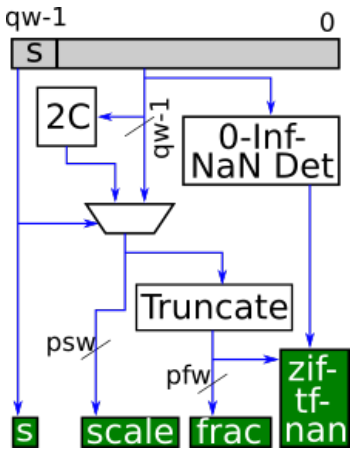


Figure 3.15: Quire-to-Posit (Q-to-P)

The Q-to-P block converts a value in the quire format to posit format as illustrated in Fig. 3.15. The Q-to-P block converts a it to a posit format so that it can be converted to Posit after normalization. After adjusting for a negative value, the `scale` and fraction are extracted from the quire as illustrated in Fig. 3.15. The truncation flag (`tf`) which is generated from truncating the fraction value, is sent to the Normalize block to control rounding to `RNE`.

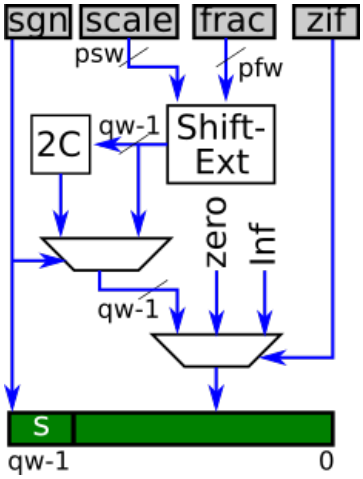


Figure 3.16: Posit-to-Quire (P-to-Q)

3.5.4 Posit-to-Quire (P-to-Q)

The P-to-Q block converts an input posit number after extraction to quire format, thereby initializing the quire. The fraction from the extractor block is shifted (based on the scale) and extended to occupy the corresponding field in quire as shown in Fig. 3.16.

Chapter 4

Melodica as an Accelerator

Bluespec Flute is a 5-stage in-order open source RISC-V processor with a synthesizable Verilog. Flute is intended for low-end to medium applications that require 64-bit operation, an MMU (Virtual Memory) and more performance than Piccolo-class processors. Since floats are already so extensively used, adding Posit ALU as an accelerator will help us bring the plus points of posits to float. I have integrated Melodica using a DMA engine to form Posit Core. This Posit Core is integrated as a hardware system accelerator with Flute.

4.1 Flute - A RISC-V CPU

The Flute is an in-order open-source CPU based on the RISC-V ISA, implemented using the BSV HL-HDL. The Flute pipeline is nominally 5-stages (as shown in 4.1) but longer for instructions like memory load-stores, integer-multiply, or floating-point operations. The core is parameterized and can be configured to operate at 32-bit or 64-bit and supports the RV64GC variant of the RISC-V ISA [17]. The Flute core also supports a memory management unit (MMU) and is capable of booting the Linux operating system. The pipeline stages in Flute are:

F: Issue fetch requests to the instruction memory. The fetch stage can also handle compressed instructions.

D: Decode the fetched instruction. Checks for illegal instructions.

E1: The first execution stage. Reads the register files or accept forwarded values from earlier instructions. Execute all single-cycle opcodes meant for the integer ALU. Branches are resolved here. Discard speculative instructions.

E2: Execute multi-cycle operations, including floating-point operations. Multi-cycle operations are dispatched to their individual pipelines from this stage. If the instruction was executed in E1, this stage is just a pass-through.

WB: Collects responses from various multi-cycle pipelines, handle exceptions and asynchronous events like interrupts, and commit the instruction.

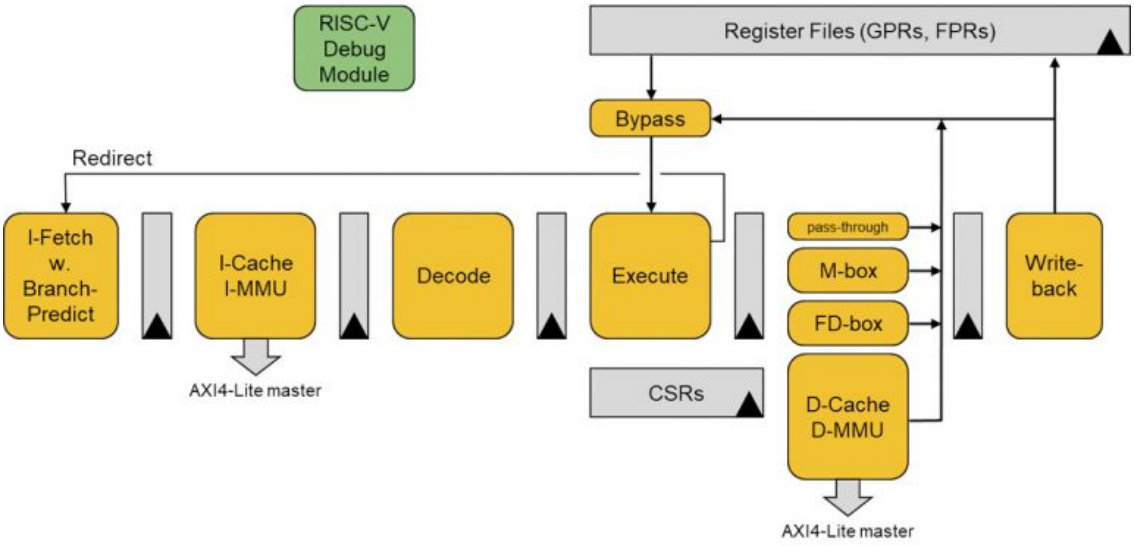


Figure 4.1: FLute RISC V CORE

A list of components substitutable for our use inside a Flute core:

1. Memory components: Memory Management Units(MMUs), L1 caches, L2 caches
2. SoC components: Interconnect fabric, memory controller, Posit Core, a few devices with easy expandability and substitutability to incorporate your own components

The Flute core has specific memory locations allocated for each of the above mentioned entities. The base address and the maximum size of the memory it can use is also fixed a priori. So basically all the entities are Memory Mapped I/O.

4.2 Components to implement the Accelerator

The Fig 4.2 depicts the main components used to implement the accelerator and the communication between them. The components can be described as follows:

4.2.1 AXI4

AXI4 (Advanced eXtensible Interface 4) is the fourth generation of the AMBA interface specification from ARM. The essence of the AXI protocol is that it provides a framework for how different blocks inside each chip communicate with each other. The AXI protocol is based on a point to point interconnect to avoid bus sharing and therefore allow higher bandwidth and lower latency. The procedure for the AXI protocol is as follows:

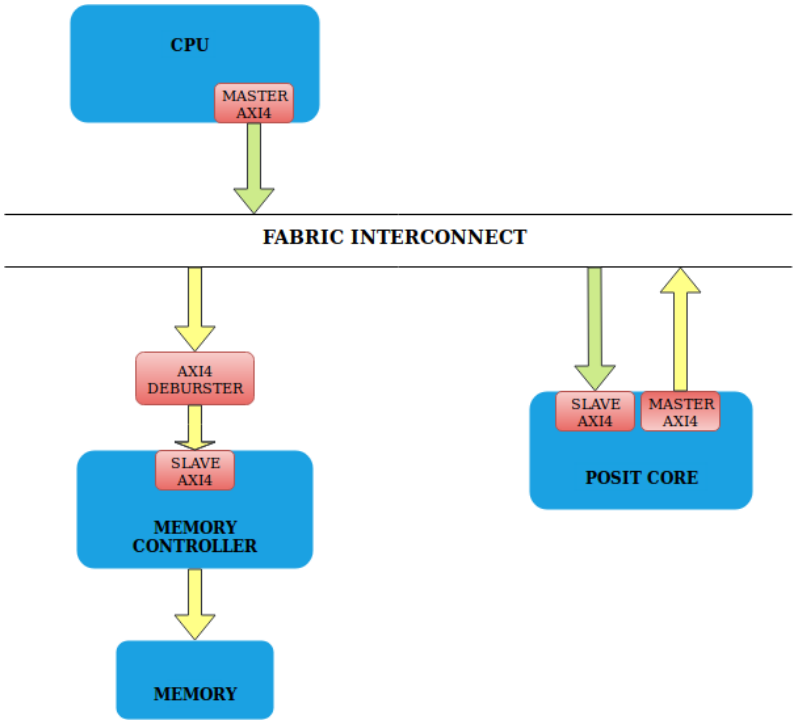


Figure 4.2: Flute Integration with Posit Core

- Master & slave must "handshake" to confirm valid signals
- Transmission of control signal must be in separate phases
- Separate channels for transmission of signals
- Continuous transfer may be accomplished through burst-type communication

By working with the master and slave devices, the AXI protocol works across five addresses that include read and write address, read and write data, and write response. Each write data channel is used to transfer data from the master to the slave upon write request from master. After the write transaction, the slave uses the write response channel to signal the completion of the transfer to the master. A read data channel to transfer data from the slave to the master upon read request from master. Since each channel has its own unique signal, it can send the handshake response uninterrupted so that it can be received and put into order. Moreover, there are parallel crossbar pathways for Write data and Read data channels. So,when more than one Write or Read data source has data to send to different destinations, data transfers can occur independently and concurrently, provided AXI ordering rules are met. The AXI protocol is burst-based. A burst can comprise multiple data transfer. Hence, each burst transactions are split into multiple transactions.

AXI4 Interconnect is a shared-Address, Multiple-Data (SAMD) crossbar type architecture. AXI Interconnect connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. The AXI Interconnect allows any mixture of

AXI master and slave devices to be connected to it. When the interface characteristics of any connected master or slave device differ from those of the crossbar switch inside the interconnect, the appropriate infrastructure cores are automatically inferred and connected within the interconnect to perform the necessary conversions.

4.2.2 Posit Core

The Posit number system format implementation, Melodica is integrated as a system accelerator with the RISC-V core. The posit core can be used to execute basic posit arithmetic operations that involves reading arrays of input posits from memory, performing the desired computations using the posits and writing the posit output back to memory.

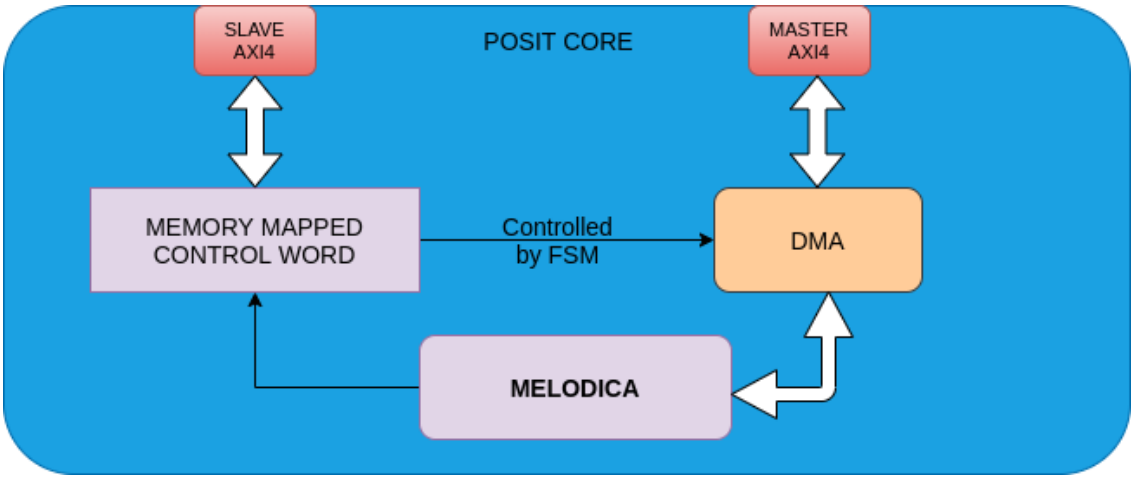


Figure 4.3: Posit Core

The Posit Core is a Memory Mapped I/O that has been implemented as shown in Fig 4.3, its working involves the following steps:

AXI4

The Posit Core has AXI4 slave as one of the interface with the Flute Core. The AXI4 Slave receives the instruction details as a set. The list includes opcode of the instruction to implement, source addresses of the arrays of posits on which the instruction has to be executed, the length of the input arrays and finally the destination where the posit result is to be stored.

Memory Mapped Control Word

The Control word is Memory Mapped Register that stores the instruction till it is fully executed. As the slave gets write requests for the instruction information it simultaneously

writes them in the Control Word. The Control word also stores the status signals that gives pointers indicating till where the instruction has been executed. When all the information has been received and stored in the Control word, the source addresses are sent to the DMA.

DMA

DMA (Direct Memory access) is used to input/output (I/O) posits as it can send or receive data directly to or from any peripheral device or even the main memory, bypassing the CPU to speed up memory operations. The DMA receives the instruction information from the MMR.
The DMA uses the the source addresses and the length of arrays to create memory accesses. The DMA controller will consolidate the several individual memory requests into occasional burst requests. The benefit of using burst mode is reduction in memory contention due to a reduction in the number of memory arbitration. So here a burst request is sent as a read request from the AXI4 Master to the fabric interconnect after all the information is packed. Having sent the address, the DMA waits for the data to arrive from the memory as a write response to the AXI4 master. The data arrives one by one in form of packets depending on the posit length and the AXI4 bus width. When the data is received it is then unpacked and interpreted to send it to the Posit ALU for computation.
The status field of the Control word is modified depending on which stage the DMA has reached. This status register can be read by the Master that is programming the Posit Core to improve the users understanding of the execution of the instruction.

Melodica

The Melodica is used to perform the Fused Operations on the posit vectors. The ALU using the product of the inputs accumulates the value in the quire. A series of fused operation computations keep occurring till the data for that instruction from the memory is exhausted. The final quire data can either be directly sent to the DMA or the value can be converted to its posit equivalent and then sent to the DMA. The DMA uses the destination address and stores the value in the system memory. After the write request has been sent using the AXI4 master, one waits for the write response telling if the value has been correctly written. This raises an interrupt flag that is sent stating the instruction has been completed and correctly update in the memory. The Melodica also modifies the status field of the Control word.

4.2.3 Memory Controller

The Memory controller acts as a mediator between interconnect fabric and low-level memory interface. It contains the logic necessary to read and write to DRAM, and to "refresh"

the DRAM based on the type of memory attached , its size, organization etc. It can be connected to different kinds of devices at the same time, including SDRAM, SRAM, ROM, and memory-mapped I/O.

The Memory controller supports fabric's burst reads and writes but the burst address requests have to be made into made into burst memory transactions so a deburster is required. Deburster is a AXI4-slave-to-AXI4-slave conversion module whose parameter interface is an AXI4-slave that carries no burst transactions while the output interface is an AXI4-slave that carries burst transactions. On the back side of the Memory Controller is a raw memory interface, a simple, wide, R/W interface which is connected to real memory in hardware (BRAM, DRAM, ...).

4.3 System Integration and Working

The combined instruction to be executed by the Posit Core is sent to it by the CPU. A set of assembly instructions corresponding to the operation are loaded in the CPU. The assembly instructions consists of stores that write the desired opcode, source addresses, vector length and destination address into the address locations of the memory mapped control word. On receiving the instruction information, the Posit core performs a set of tasks as described earlier, finally sending burst memory requests. These memory requests are received by the slave of the AXI4 deburster and sent to the memory controller that performs the memory transactions to give the input posit values that are sent back to the posit core. The posit core performs the desired operations. An interrupt flag is raised and sent to the CPU when the Posit Core has successfully completed its task. The CPU can also send read request for reading the status field to the Posit Core. The accelerator pipelines are clean and it was used to successfully evaluate the dot product of two arrays.

Chapter 5

Clarinet

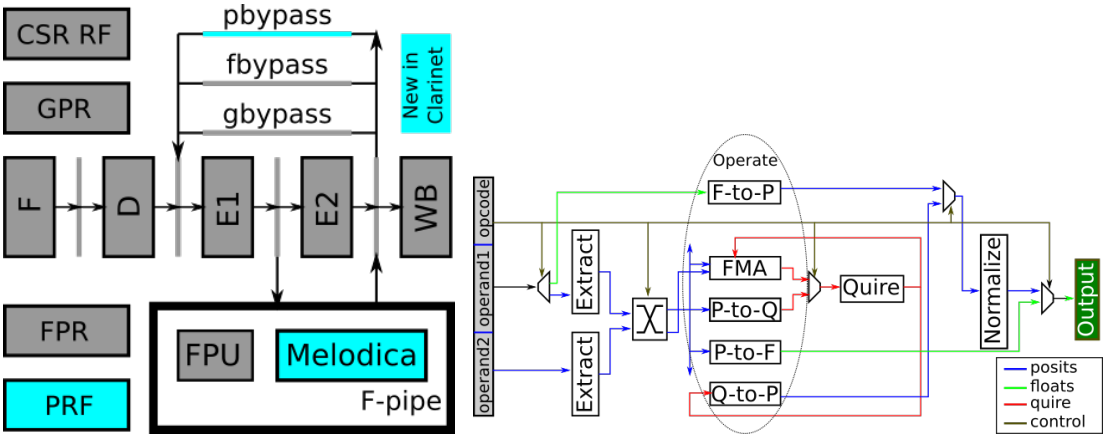
Clarinet, a RISC-V CPU that is enhanced with special instructions for posit arithmetic and a dedicated posit register file (PRF). We prefer Melodica as an add-on feature in the Flute rather than a replacement for floating-point arithmetic hardware. With Clarinet, We are trying to enable researchers to study the advantages and disadvantages of posit arithmetic. A posit arithmetic empiricism; reasoning based on empirical data for posit arithmetic is needed to quantify the benefits. Furthermore, we see an opportunity for the floats and posits to coexists in a single platform to trade among power, performance, area, and accuracy.

Since Melodica is an add on to the Floating point arithmetic unit, as of now, we support a limited number of fused and type convert operations and quire to carry-out experimental studies for our applications. We decided to only add the above mentioned operations to bring out the usability of quire to floating point numbers. We plan to extend Clarinet with more functionality since the Melodica core is extensible to support operations demanded by the applications. To the best of our knowledge, this is the *first-ever* quire enabled RISC-V CPU.

5.1 Clarinet organization

Clarinet's organization is illustrated in Fig. 5.1a. The starting point for Clarinet was a Flute CPU core as shown in Section 4.1, configured with the RV32IMAFC variant of the RISC-V ISA [17].

Clarinet integrates Melodica as a functional execution unit parallel to the existing floating point unit. A new module hierarchy, (`F-pipe`), encapsulates both the existing floating point core, and the new Melodica core. A thin layer of logic in `F-pipe` directs the five new instructions to Melodica, while all other floating point instructions continue to be serviced by the `FPU`. `F-pipe` also routes responses from Melodica back to the Clarinet pipeline. Except for instructions that update the quire, all other instructions result in outputs from Melodica destined for the `FPR`, `PRF` and `CSR RF`.



(a) Clarinet pipeline (b) Melodica in Clarinet

Figure 5.1: Clarinet

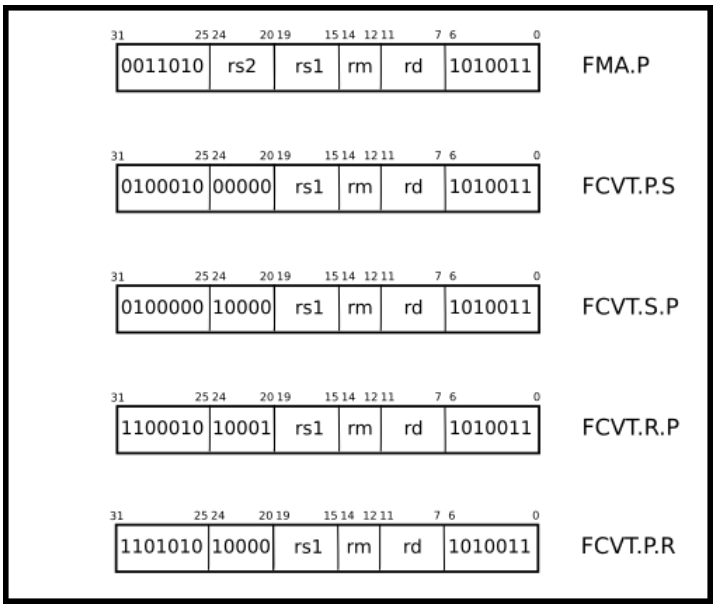


Figure 5.2: New Clarinet instructions

5.2 Custom Instructions

In order to use the integrated Melodica execution unit we added five new instructions to the existing instruction set implemented in Flute. As shown in their bit representations in Fig. 5.2 all the instructions belong to the `R`-format type of the RISC-V ISA. All five instructions use the `FP-OP` value as defined in [17], for their seven-bit opcodes. In order to handle posit types, a new binary encoding `10` was introduced for the `fmt` field. In `R`-format instructions, these bits occupy the LSB of the `funct7` instruction field. Also new `Rs2` binary encoding was introduced for the posit (`10000`) and quire (`10001`) types.

- **FMA.P:** Multiplies two posit operands present in the PRF at `Rs1` and `Rs2`, and accumulates the result into the quire. Do not update `FCSR.FFLAGS`.

- **FCVT.S.P:** Converts the posit value in PRF at `Rs1` to a floating-point value which is written to the FPR at `Rd`. This instruction may update `FCSR.FFLAGS`.
- **FCVT.P.S:** Converts the floating-ppint value in the FPR at `Rs1` to a posit value which is written to the PRF at `Rd`. This instruction may update `FCSR.FFLAGS`.
- **FCVT.R.P:** Converts the posit value in the PRF at `Rs1` to a quire value which is written to the quire. Do not update `FCSR.FFLAGS`.
- **FCVT.P.R:** Converts the value in the quire to a posit value which is written to the PRF at `Rd`. This instruction may update `FCSR.FFLAGS`.

The decision to add new instructions instead of reusing existing opcodes belonging to the `F` subset of the RISC-V ISA was driven by two requirements – integrating quire functionality (which does not exist in floating-point), and type-converter instructions that would allow posits and floating-point to coexist in an application as independent types.

The new type-converter instructions allow existing programs to be run on Clarinet without the need to modify their original data segments as has been demonstrated in section 6.2.1. From our experiments we realised that applications could see significant reductions in normalized error through the introduction of quire-based accumulation even when most of the computation remained in floating-point. When an application can benefit from the use of posits (be it greater dynamic range or accuracy), the type-converter instructions allow the user to convert a part of the computation to posits and accumulate into the quire register. In order to do so, they would first need to convert their intermediate floating-point data to posits using the type-converter instructions, before executing the `FMA.P` instruction that accumulates into the quire. Eventually, the results are converted back to the floating-point format before writing out to memory.

5.3 Integrating the quire

The recommended size of the quire can grow very rapidly with increasing posit-width. This implies that treating the quire register similar to an entry in one of the register files would be quite expensive as far as hardware resources are concerned. For instance, using 32-bit posits would mean making a 512-bit quire value available on the forwarding paths and from the register files. Further, providing a path from quire to memory (via modified load and store instructions) would require extensive modifications of the memory pipeline.

Clarinet takes a novel approach to integrating the quire. The quire can be updated directly using the new instructions (`FCVT.R.P` and `FMA.P`). However, in order to save hardware resources, there are no instructions to directly access the quire or read and write the quire to memory. To read the quire's value it has to be first converted to a posit type using the instruction `FCVT.P.R` which would bring the converted value into the `PRF`. These decisions allow us to contain the cost of integrating the quire to just the actual storage for the quire register.

5.4 The posit register file

A key advantage of posits (especially with quire) is that it may be profitable to implement non-standard widths for posit numbers while still retaining most of the precision advantages of operating with posits and quire. To this end, we introduced a new PRF into Clarinet – one that is sized to the value of the posit variables being handled by the Melodica core. While it would have been possible to reuse the floating point register file for posit operations, this would not have permitted the flexibility of benefiting from the use of narrower posit-widths for applications that allowed lower bit-widths. The registers in the PRF may only be accessed by instructions which directly take posits as inputs or result in a posit output. A new register file implies the creation of a new bypass path to forward in-flight posit operands from the output of the `F-pipe` to the input to `E1`. This new path is marked as `pbypass` in Fig. 5.1a and handles only posit values.

Chapter 6

Results

6.1 Verification of Melodica

The current parameterizable implementation is done using BlueSpec, converted to Verilog using verilator and simulates the results in Vivado HLS.

6.1.1 Extract-Basic Operations-Normalize Pipeline

Successfully been able to run the Extract-Add-Normalization pipeline for exhaustive tests for 8, 16, 32 bit posit on De0-Nano board. The two posit say X, Y are given as input to the extraction block. The output posit is seen at the end of the normalisation block. In the Extract-Add-Normalize pipeline, I perform the operation X + Y - Y . Also, Verified Extract-ADD/SUB/MUL/DIV/MAC-Normalize pipeline with exhaustive tests for 8, 16 bit posit and extensive tests for 32 and 64 bit posits using SoftPosit[16].

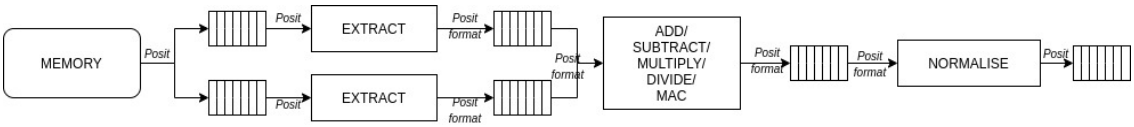


Figure 6.1: Extract-ADD/SUB/MUL/DIV/MAC-Normalize Pipeline

Add/Subtract Pipeline

From the posit input into the pipeline before the FIFO to the final output at the end of the FIFO is 7 cycles. The Extract block in section 3.1 is has 1 stage, Normalisation block in section 3.2 also has 1 stages and the addition/subtraction block in section 3.3.1 has 3 stages. At the beginning and end of pipeline there are FIFOs. So the system has 5 stages.

Multiply Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 7 cycles. The Extract block in section 3.1 is has 1 stage, Normalisation block in section 3.2

also has 1 stage and the multiplication block in section 3.3.2 has 3 stages. At the beginning and end of pipeline there are FIFOs. So the system has 5 stages.

Divide Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 4 + stages required for division cycles. The Extract block in section 3.1 is has 1 stage, Normalisation block in section 3.2 also has 1 stage and the division block in section 3.3.3 takes input dependent stages. At the beginning and end of pipeline there are FIFOs. So the system has 2 + stages required for division cycles stages.

MAC Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 9 cycles. The Extract block in section 3.1 is has 1 stage, Normalisation block in section 3.2 also has 1 stage and the MAC block in section 3.3.4 has 3 stages for add, 2 stages for multiply. At the beginning and end of pipeline there are FIFOs. So the system has 7 stages.

6.1.2 Extract-Fused Operations Pipeline

Successfully been able to run the Extract-FDP pipeline for a bunch of vector arrays for 8,16,32 and 64 bits using the Flute Core and synthesized on Vivado. Also, extensively verified the F to P-Normalize pipeline for 8, 16, 32 and 64 bit posits using SoftPosit.

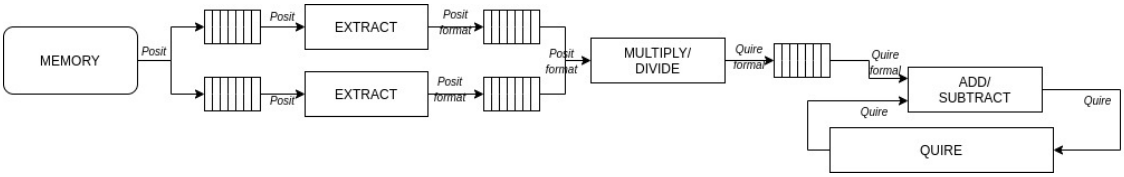


Figure 6.2: Extract-FMA/FMS/FDA/FDS Pipeline

FMA/FMS Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 7 cycles. The Extract block in section 3.1 is has 1 stage and the FMA/FMS block in section 3.4.1 has 4 stages (2 for add, 2 for mul). At the beginning and end of pipeline there are FIFOs. So the system has 5 stages.

FDA/FDS Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 5 + input dependent cycles. The Extract block in section 3.1 is has 1 stage and the FDA/FDS block in section 3.4.1 has 2 stages (for adder) + input dependent divider stages. At the

beginning and end of pipeline there are FIFOs. So the system has 3 + input dependent stages.

6.1.3 Type Converter Pipelines

F to P-Normalize pipeline

Extensively verified the F to P-Normalize pipeline for 8, 16, 32 and 64 bit posits using SoftPosit.

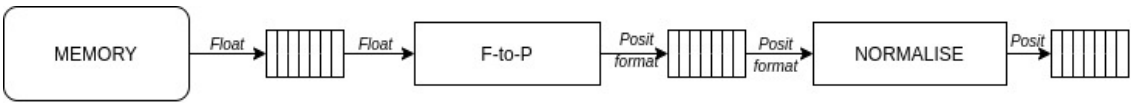


Figure 6.3: F to P-Normalize Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 5 cycles. Normalisation block in section 3.2 also has 1 stage and the f-to-p block in section 3.5.1 has 2 stages. At the beginning and end of pipeline there are FIFOs. So the system has 3 stages.

Extract-P to F Pipeline

Extensively verified the Extract-P to F pipeline for 8, 16, 32 and 64 bit posits using SoftPosit.

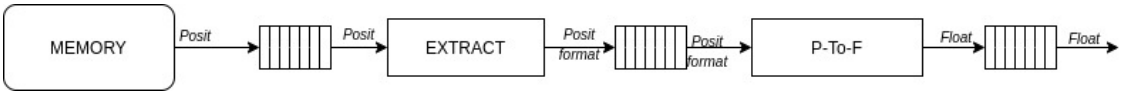


Figure 6.4: Extract-P to F Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 5 cycles. Extraction block in section 3.1 also has 1 stage and the p-to-f block in section 3.5.2 has 2 stages. At the beginning and end of pipeline there are FIFOs. So the system has 3 stages.

Q to P-Normalize Pipeline

Extensively verified the Q to P-Normalize pipeline for 8, 16, 32 and 64 bit posits using SoftPosit.

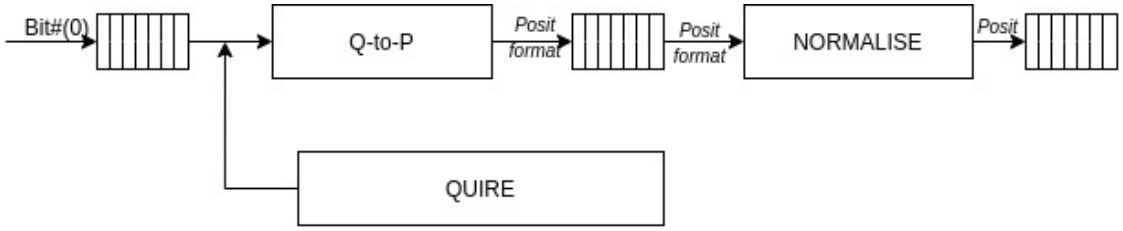


Figure 6.5: Q to P-Normalize Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 6 cycles. Normalisation block in section 3.2 also has 1 stage and the q-to-p block in section 3.5.3 has 3 stages. At the beginning and end of pipeline there are FIFOs. So the system has 4 stages.

Extract-P to Q Pipeline

Extensively verified the Extract-P to Q pipeline for 8, 16, 32 and 64 bit posits using SoftPosit.

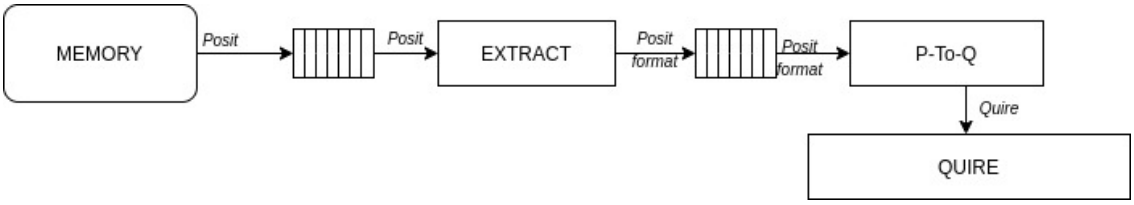


Figure 6.6: Extract-P to Q Pipeline

From the posit input into the pipeline to the final output at the end of the FIFO is 5 cycles. Extraction block in section 3.1 also has 1 stages and the p-to-q block in section 3.5.4 has 2 stages. At the beginning and end of pipeline there are FIFOs. So the system has 3 stages.

6.1.4 Comparison of results

Resource usage for different bit size inputs for the different operations that can be performed on Melodica as shown in 6.1.4:

Operation	**LUT**	**Register**	**DSP**	**No of Stages**	**Delay (ns)**	F_{max} **(MHz)**
Addition						
N= 8, es= 0	465	226	0	5	6.15	162.6
N= 16, es= 1	1235	392	0	5	8.542	117.07
Multiplication						
N= 8, es= 0	384	209	0	5	5.13	194.93
N= 16, es= 1	757	349	1	5	7.044	141.96
Division						
N= 8, es= 0	393	252	0	2+input-dependent	4	250
N= 16, es= 1	911	415	0	2+input-dependent	6.95	143.89
MAC						
N= 8, es= 0	2101	541	0	7	5.2	192.31
N= 16, es= 1	3091	853	1	7	6.971	143.46
FMA						
N= 8, es= 0	675	379	0	5	3.38	295.86
N= 16, es= 1	2213	1216	1	5	5.25	190.48
FDA						
N= 8, es= 0	733	444	0	3+input-dependent	4.94	202.43
N= 16, es= 1	2419	1359	1	3+input-dependent	5.21	191.94
F-To-P						
N= 8, es= 0	161	98	0	3	3.12	320.5
N= 16, es= 1	412	152	0	3	7.442	134.38
P-To-F						
N= 8, es= 0	136	106	0	3	1.96	510.2
N= 16, es= 1	392	380	0	3	2.5	400
Q-To-P						
N= 8, es= 0	304	162	0	4	3.25	307.69
N= 16, es= 1	1259	410	0	4	4.01	249.37
P-To-Q						
N= 8, es= 0	214	134	0	3	1.97	507.6
N= 16, es= 1	599	438	0	3	2.8	357.14

6.2 Case Study

I look into application kernels that are rich in floating-point arithmetic operations. I cover case studies on optical flow estimation using Lucas-Kanade method. I use this information to tweak parameters in Clarinet to arrive at a customized Clarinet instance.

6.2.1 Using Clarinet - A Simple Example

No	Instruction	Disassembly	RF/Quire Updates	Comments
1	00052007	flw ft0, 0(a0)	`FPR[0](ft0) <- 0x40200000`	Load 2.50 to FPR ft0 from memory
2	00452087	flw ft1, 4(a0)	`FPR[1](ft1) <- 0x40800000`	Load 4.00 to FPR ft1 from memory
3	44000053	fcvt.p.s p0, ft0	`PRF[0](p0) <- 0x5400`	Execute F-to-P on ft0. Result in PRF p0
4	440080d3	fcvt.p.s p1, ft1	`PRF[1](p1) <- 0x6000`	Execute F-to-P on ft1. Result in PRF p1
5	c5110053	fcvt.r.p p2	`Quire <- 0x0`	Execute P-to-Q on p2. Result in quire
6	34100053	fma.p p0, p1	`Quire<-0x00..0a00000000000000`	Accumulate (p0*p1) into quire
7	34100053	fma.p p0, p1	`Quire<-0x00..1400000000000000`	Accumulate (p0*p1) into quire
8	d5000153	fcvt.p.r p2	`PRF[2] <- 0x7100`	Execute Q-to-P on Quire. Result in PRF p2
9	41010153	fcvt.s.p ft2, p2	`FPR[2]<- 0x41a00000`	Execute P-to-F on p2. Result in FPR ft2

Table 6.1: An ASM example on Clarinet-Melodica

Clarinet-Melodica introduces a new usage model for the posit programmer by focusing on quire functionality. While Clarinet-Melodica does not offer support for operations like posit addition, subtraction, and multiplication through dedicated instructions, it does so via the `FMA.P` instruction. The example presented in Table 6.1 is of a simple case where a user loads two, 32-bit floating-point numbers from memory and does a series of operations on them using the quire. In this example, Clarinet-Melodica is configured to use 16-bit posits. In particular, instruction number 6 illustrates how a user could use the `FMA.P` instruction to multiply two posit operands (by initializing the quire to zero). Furthermore, this form of multiplication does not suffer rounding error as the result accumulates into the quire. Similarly substituting the first or second operand in `FMA.P` as $\pm$ 1.0 would allow the user to add posits to or subtract posits from the quire.

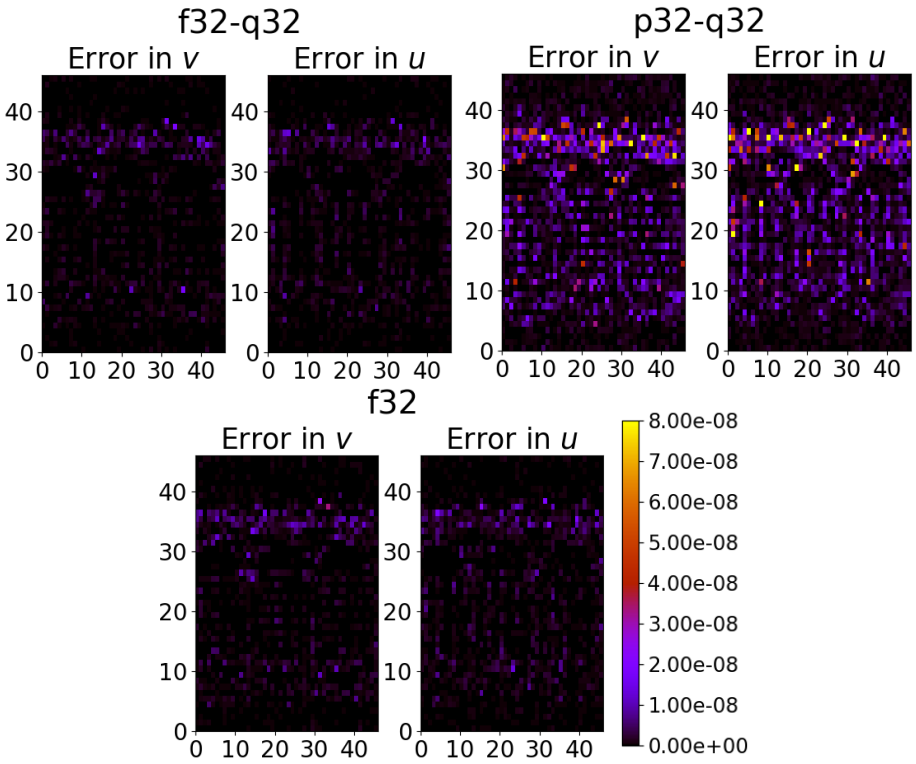
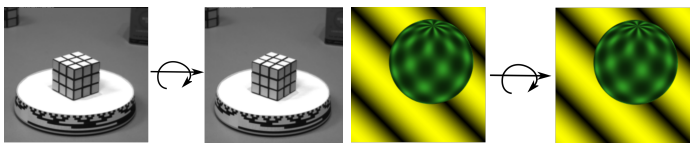


Figure 6.8: Error heat-maps of Rubik's cube for pixels between 0-255

6.2.2 Lucas-Kanade Optical Flow

Lucas-Kanade is a differential method of tracking features given a sequence of frames. Given `I` as brightness-per-pixel at `(x,y)`, the local optical flow (velocity) vector $(\vec{u},\vec{v})$ is given by equation 6.1.

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \tag{6.1}$$



(a) Rubik's cube (b) Sphere

Figure 6.7: Dataset for Lucas-Kanade

The Lucas-Kanade method is used to calculate the optical flow for consecutive frames of rotating objects which are given in Fig. 6.7. I compare the different posit and single-precision floating-point configuration combinations with 64-bit floating-point values using SoftPosits, and generate heat maps of the absolute error for both `u` and `v`. The three configurations that are being compared are: i) 32-bit single-precision floating-point arithmetic (`f32`), ii) 32-bit single-precision float arithmetic combined with `N`-bit quire

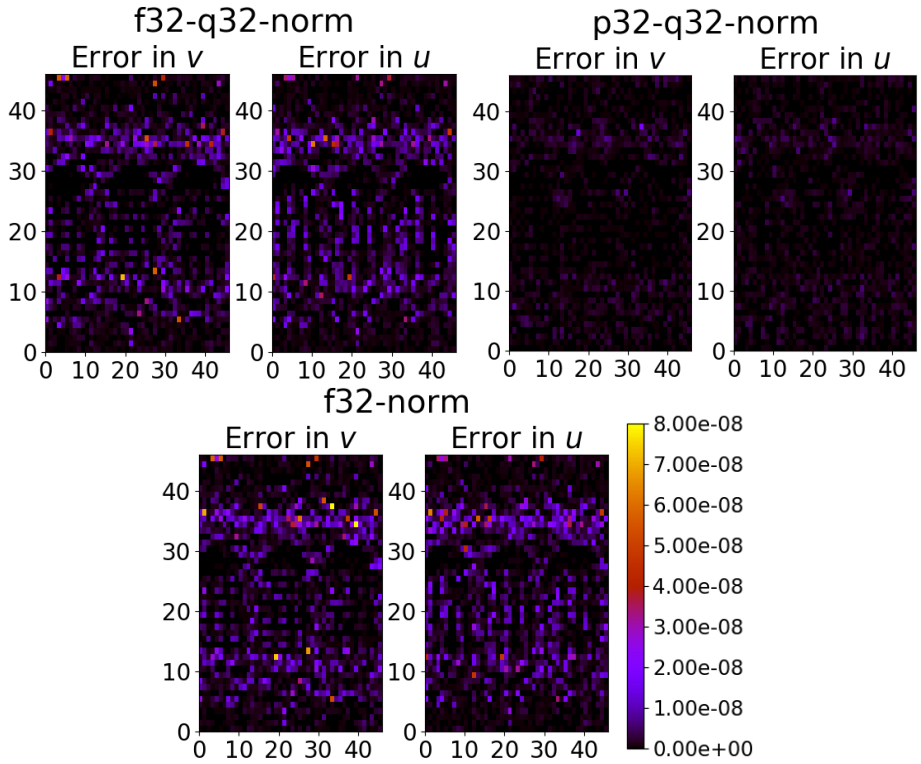


Figure 6.9: Error heat-maps of Rubik's cube normalized pixels between 0.0-16.0

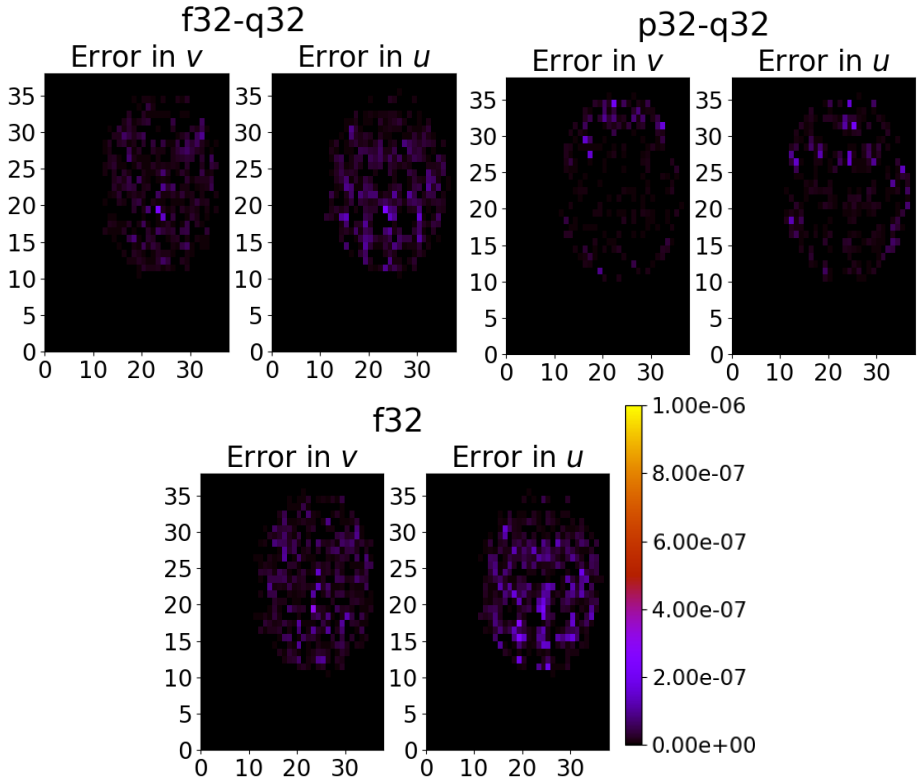


Figure 6.10: Error heat-maps of sphere pixels between 0-255

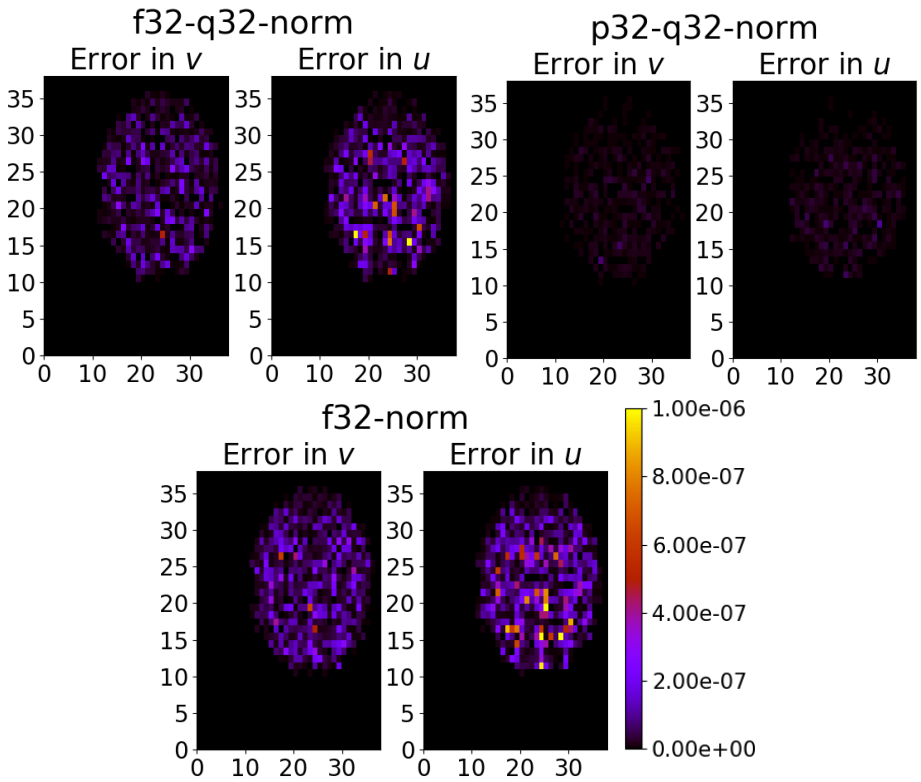


Figure 6.11: Error heat-maps of sphere normalized pixels between 0.0-16.0

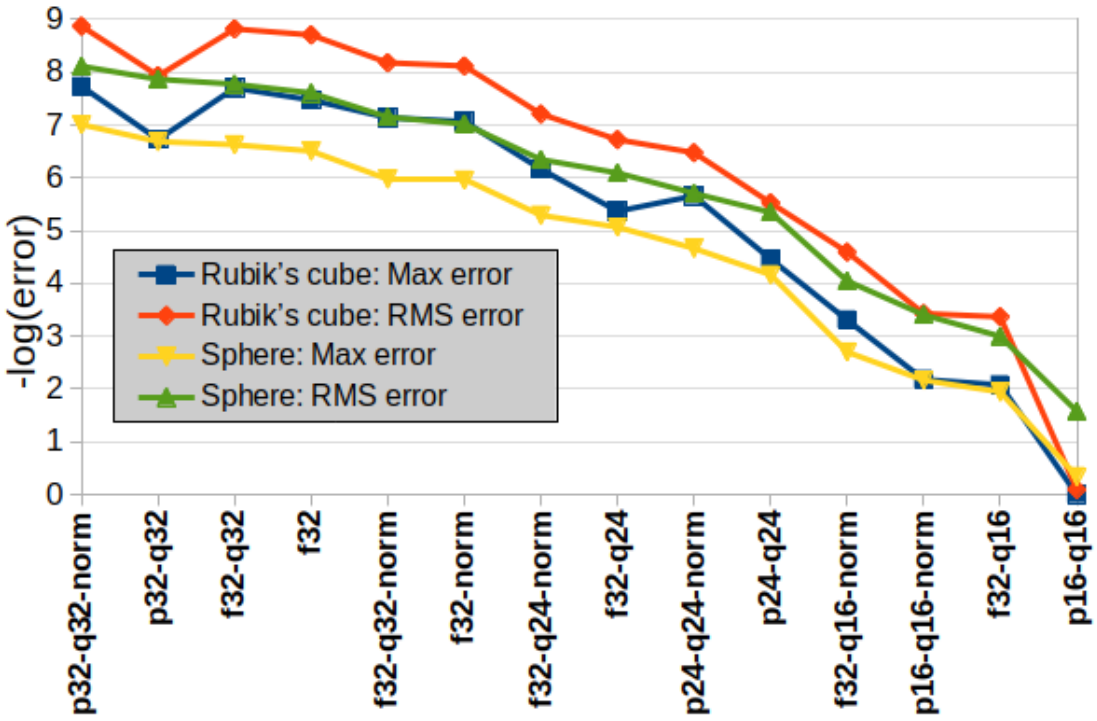


Figure 6.12: Maximum error and RMS error with respect to 64-bit floating-point

arithmetic (f32-qN), iii) N-bit posit arithmetic and N-bit quire arithmetic (pN-qN). Furthermore, owing to the better accuracy of posits around 1.0 I have normalized (norm) grey-scale pixel values (0 to 255) to (0.0 to 16.0).

From the heat-maps in Fig. 6.8 the effects of normalization and q32 on error become obvious. When working with normalized data, the configuration p32-q32 clearly outperforms all other configurations. For data which is not normalized, the performance of p32-q32 depends on whether the data naturally falls around 1.0. However, even in the non normalized case, f32-q32 performs consistently better than f32. The general trend in maximum and RMS error for Rubik's cube and sphere object frames for different configurations are shown in Fig. 6.12. The y-axis value for RMS error in Fig. 6.12 gives the number accurate digits for the configurations compared to 64-bit floating-point. Allowing one decimal places of tolerance to error, p24-q24-norm configuration can give accurate results close to f32. With a penalty of 2 more decimal place p16-q16-norm can be a feasible alternative. When optical flow is computed for posit configurations for values not around 1.0 the accuracy falls.

As summarised in Table 6.2 p32-q32-norm configuration results in an order improvement in accuracy as compared to f32 for the sphere dataset. The f32-q32 configuration for gray-scale pixel values (0-255) improves the accuracy by 23% and 32% for Rubik's cube and sphere dataset respectively.

6.3 Experimental results

Different configurations of Clarinet-Melodica were synthesized using Synopsys Design Compiler. All designs were synthesized with a clock frequency of 200 MHz, on a Faraday 90 nm-CMOS Faraday process. No special memory cells were used to synthesize the register files or branch target buffers.

Melodica is not a complete posit implementation. It delivers accumulator functionality using Quire, and is meant to be used alongside a 32-bit floating-point implementation. The baseline for comparisons is a 32-bit RISC-V Clarinet processor with support for 32-bit floating-point arithmetic, and no Melodica. This is the minimum functionality required in a RISC-V CPU to integrate Melodica.

For the purpose of comparison, the following five implementations were evaluated:

- **Clarinet-Base**: This is the baseline implementation, which features a 32-entry, 32-bit wide floating-point register file (FPR) and bypass logic, and an FPU which is

Configurations	Rubik's cube	Sphere
p32-q32-norm	1.3415103400712e-09	7.683256359993e-09
f32-q32	1.5047392902563e-09	1.6771673416184e-08
f32	1.9613096794474e-09	2.4623573707996e-08

Table 6.2: RMS error in optical flow

capable of single-precision arithmetic. Clarinet-Base does not integrate a Melodica core, but does support the new posit-related custom instructions as described in Section 5.2.

- **Clarinet-Double**: Support for 64-bit floating-point arithmetic is added to the Clarinet-Base implementation. The FPR is doubled to be 64-bit wide and the bypass paths for floating-point values are suitably widened. The FPU arithmetic unit is now capable of processing 64-bit floating-point operands.

- **Clarinet-P-16-1**: Melodica configured with `N`=16 and `es`=1, is integrated into the Clarinet-Base configuration. In this configuration, Melodica features a 128-bit Quire. The PRF has 32, 16-bit registers, and bypass logic.

- **Clarinet-P-24-2**: Melodica configured with `N`=24 and `es`=2, is integrated into the Clarinet-Base configuration. In this configuration, Melodica features a 288-bit Quire. The PRF and bypass logic are widened to 24-bit.

- **Clarinet-P-32-2**: Melodica configured with `N`=32 and `es`=2, is integrated with the Clarinet-Base configuration. In this configuration, Melodica features a 512-bit Quire. The PRF and bypass logic are widened to 32-bit.

As indicated in Table 6.3, adding support for 64-bit floating-point leads to a nearly 72% increase in area over `Clarinet-Base`. In comparison adding Melodica configured with `N`=16, `es`=1 adds approximately 9% area. Interestingly, the area overhead moving to wider values of `N` (24 and 32) is marginal (around 3%). The reason for this is Clarinet's organization where moving from Clarinet-Base to Clarinet-P-16-1, introduces the new PRF and bypass logic for posit types apart from the Melodica pipes themselves. On the other hand, moving to wider posit-widths introduces no new structures, but simply widens existing ones.

Table 6.3 also notes the cell switching power. Cell switching power does not include the power dissipated due to net switching. Net switching, in particular the clock network, dominates overall power dissipation. Between 70% and 80% of the power is dissipated in the clock tree alone and is largely unchanged in the different configurations. For this reason, I found it more instructive to highlight the cell switching power which amplifies the effect each configuration has on dynamic power dissipation.

6.3.1 Quality of Clarinet

Quality of Clarinet (QoC) is defined using equation 6.2.

$$QoC = \frac{A_{base} - \frac{A_{iuc}}{K}}{A_{base}} \times 100 \tag{6.2}$$

where A_{base} is the area of Clarinet-base, A_{iuc} is the are of the *instance under consideration*, and K is the number of accurate digits in the instance under consideration. The

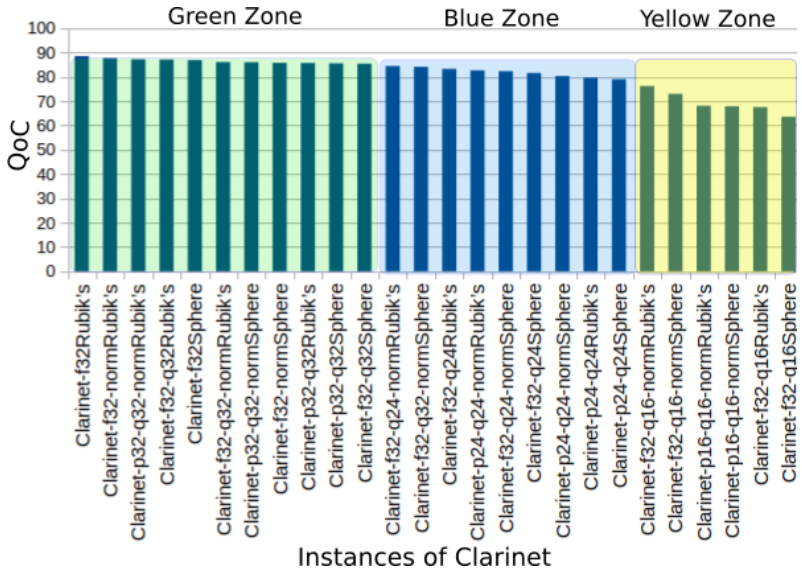


Figure 6.13: Quality of Clarinet instances for Lucas Kanade

QoC is a metric that incorporates platform configuration and application accuracy to measure the quality of Clarinet instances. A high-quality implementation would mean an implementation with a low area footprint supporting high precision computations. An implementation supporting high precision computations can be of low-quality if the implementation incurs a high area footprint. A similar equation can be formulated for the power footprint of the instances of Clarinet. The QoC is segregated in three zones : green zone, blue zone and yellow zone. The QoC in the green zone is superior to the rest of the zones and the quality of the platform is more than 0.85 (85%) while the QoC in blue zone is between 0.75$\leq$ QoC <0.85 and in yellow zone the QoC is less than 0.75. The routines that involve addition and multiplication of posits are implemented using FMA operation of Clarinet as described in Section 6.2.1. The QoC for `Clarinet-f32` is better than other Clarinet instances since it uses less area. But using `Clarinet-p32-q32-normRubik's` and `Clarinet-p32-q32-normSphere` implementations, which have QoC of 87.2% and 86% respectively, an order improvement in accuracy is noticed. As the accuracy varies the QoC varies and depending on the accuracy requirements of the application, a suitable instance of Clarinet can be considered.

Implementation	Melodica Gates	Total Gates	Total Area	Cell Switching Power (mW)
Clarinet-Base	0	59,543	416,359	20.63
Clarinet-Double	0	106,321	713,177	29.29
Clarinet-P-16-1	3,687	64,649	454,834	23.18
Clarinet-P-24-2	5,097	66,171	467,427	23.88
Clarinet-P-32-2	6,282	67,151	473,461	24.38

Table 6.3: Comparison of total cells and area for different Clarinet configurations

6.4 Future work

1. Ubit : The current posit system doesn't have the ubit feature that can be used to define intervals. Introducing ubit per tile will give better answers since it allows representation of closed, open, and half-open intervals allowing values to be represented even if they don't lie in the given exact posit number.

2. Parameterization of the Pipeline stages : To a new feature to the system, we can parameterize the number of stages on the pipeline so that we can have control over the clock frequency. This will give us variability.

3. Application : Implement the RISC-V system for a bunch of different applications with compiler support for both Clarinet and Melodica accelerator.

4. Design Improvement : Perform exhaustive testing or specifically check for corner cases and also do checking for es > 2 since soft posit doesn't have implementations above es > 2. Also, do a detailed study of the number of input cycles required to perform an integer division in Melodica.

5. While Melodica has seen extensive unit-level verification, further system-level tests are in progress on Clarinet.

Bibliography

[1] R. Jain, N. Sharma, F. Merchant, S. Patkar, and R. Leupers, "Clarinet: A risc-v based framework for posit arithmetic empiricism," 2020.

[2] C. Leong. (2018, Nov.) Softposit version 0.4.1rc. [Online]. Available: https://gitlab.com/cerlane/SoftPosit

[3] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: The good, the bad and the ugly," in *Proceedings of the CoNGA 2019*, ser. CoNGA'19. New York, NY, USA: ACM, 2019. [Online]. Available: https://doi.org/10.1145/3316279.3316285

[4] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl, "Make it real: Effective floating-point reasoning via exact arithmetic," in *DATE 2014*, March 2014, pp. 1–4.

[5] A. Volkova, M. Istoan, F. De Dinechin, and T. Hilaire, "Towards hardware iir filters computing just right: Direct form i case study," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 597–608, April 2019.

[6] Gustafson and Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, pp. 71–86, Jun. 2017. [Online]. Available: https://doi.org/10.14529/jsfi170206

[7] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks," in *2019 IEEE 26th ARITH*, June 2019, pp. 88–91.

[8] M. K. Jaiswal and H. K. . So, "Universal number posit arithmetic generator on fpga," in *DATE 2018*, March 2018, pp. 1159–1162.

[9] ——, "Architecture generator for type-3 unum posit adder/subtractor," in *ISCAS 2018*, May 2018, pp. 1–5.

[10] ——, "PACoGen: A hardware posit arithmetic core generator," *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019.

[11] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized posit arithmetic hardware generator," in *2018 IEEE 36th ICCD*, Oct 2018, pp. 334–341.

[12] H. F. Langroudi, Z. Carmichael, D. Pastuch, and D. Kudithipudi, "Cheetah: Mixed low-precision hardware & software co-design framework for dnns on the edge," 2019.

[13] H. Zhang, J. He, and S. Ko, "Efficient posit multiply-accumulate unit generator for deep learning applications," in *ISCAS 2019*, May 2019, pp. 1–5.

[14] J. Lu, S. Lu, Z. Wang, C. Fang, J. Lin, Z. Wang, and L. Du, "Training deep neural networks using posit number system," 2019.

[15] S. Tiwari, N. Gala, C. Rebeiro, and V. Kamakoti, "PERI: A Posit Enabled RISC-V Core," pp. 1–14, 2019. [Online]. Available: http://arxiv.org/abs/1908.01466

[16] L. van Dam, "Enabling high performance posit arithmetic applications using hardware acceleration," http://resolver.tudelft.nl/uuid:943f302f-7667-4d88-b225-3cd0cd7cf37c, 2018.

[17] Editors Andrew Waterman and Krste Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, 2019.