

# CLARINET: A RISC-V Based Framework for Posit Arithmetic Empiricism

RIYA JAIN, Indian Institute of Technology, Bombay, India

NIRAJ SHARMA, Indian Institute of Technology, Bombay, India

FARHAD MERCHANT, RWTH Aachen University, Germany

SACHIN PATKAR, Indian Institute of Technology, Bombay, India

RAINER LEUPERS, RWTH Aachen University, Germany

Many engineering and scientific applications require high precision arithmetic. IEEE 754-2008 compliant (floating-point) arithmetic is the de facto standard for performing these computations. Recently, *posit arithmetic* has been proposed as a drop-in replacement for floating-point arithmetic. The posit data representation and arithmetic offer several absolute advantages over the floating-point format and arithmetic including higher dynamic range, better accuracy, and superior performance-area trade-offs.

In this paper, we present a consolidated general-purpose processor-based framework to support posit arithmetic empiricism. The end-users of the framework have the liberty to seamlessly experiment with their applications using posit and floating-point arithmetic since the framework is designed for the two number systems to coexist. The framework consists of *Melodica* and *Clarinet*. *Melodica* is a posit arithmetic core that implements parametric fused-multiply-accumulate and, more importantly, supports the *quire* data type. *Clarinet* is a *Melodica*-enabled processor based on the RISC-V ISA. To the best of our knowledge, this is the first-ever integration of *quire* to a RISC-V core. To show the effectiveness of the *Clarinet* platform, we perform an extensive application study and benchmarking on some of the common linear algebra and computer vision kernels. We perform ASIC synthesis of *Clarinet* and *Melodica* on a 90 nm-CMOS Faraday process. Finally, based on our analysis and synthesis results, we define a quality metric for the 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.

CCS Concepts: • **Hardware** → **Arithmetic and datapath circuits**; *Application specific instruction set processors*.

Additional Key Words and Phrases: posit arithmetic, RISC-V, open-source hardware, custom instructions

## ACM Reference Format:

Riya Jain, Niraj Sharma, Farhad Merchant, Sachin Patkar, and Rainer Leupers. 2020. CLARINET: A RISC-V Based Framework for Posit Arithmetic Empiricism. *ACM Trans. Arch. Code Optim.* 1, 1 (June 2020), 18 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Authors' addresses: Riya Jain, [riyajain78@gmail.com](mailto:riyajain78@gmail.com), Indian Institute of Technology, Bombay, P.O. Box 1212, Bombay, Maharashtra, India, 400076; Niraj Sharma, Indian Institute of Technology, Bombay, 4th Floor, Department of Electrical Engineering, Bombay, India, [nirajns@iitb.ac.in](mailto:nirajns@iitb.ac.in); Farhad Merchant, RWTH Aachen University, Aachen, Germany; Sachin Patkar, Indian Institute of Technology, Bombay, 4th Floor, Department of Electrical Engineering, Bombay, Maharashtra, India; Rainer Leupers, RWTH Aachen University, Kopernikusstra e 16, Aachen, NRW, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

  2020 Association for Computing Machinery.

XXXX-XXXX/2020/6-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

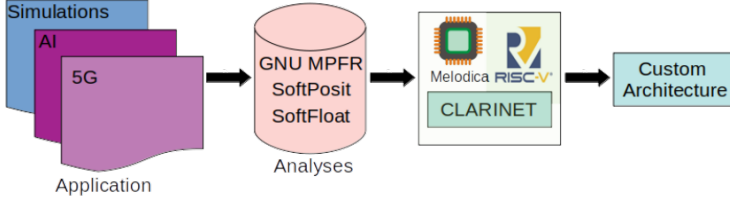


Fig. 1. Clarinet in the platform design cycle

## 1 INTRODUCTION

High precision arithmetic is necessary for several scientific and engineering applications. Some of the examples are Monte Carlo simulations, Kalman Filter, circuit simulations, and others [13][18]. These applications require high precision arithmetic hardware for efficient execution [5]. Floating-point arithmetic hardware has been challenging to design due to the intricacies involved in staying within the limits of the desired area and power budget [16][21]. Specifically, designing a portable floating-point unit (FPU) has been a complex task due to the precise requirements posed by the IEEE 754-2008 (floating-point) standard. Recently, researchers and computer architects have either compromised on the compliance to the standard or devised their formats to overcome the design challenges [9][3]. *Posit* is one such data representation proposed by John L. Gustafson in the year 2017 which aims to overcome shortcomings of the floating-point format [9].

The posit arithmetic and data representation have several absolute advantages over floating-point arithmetic and format. The advantages are simpler hardware, smaller area and energy footprints, and higher dynamic range and numerical accuracy. In general,  $n$ -bit posit has a better dynamic range compared to the  $n$ -bit floating-point [8]. In the past, researchers have shown that the  $n$ -bit floating-point arithmetic can be replaced by  $m$ -bit posit arithmetic units where  $m < n$ . It has also been shown empirically that the replacement does not cause loss of accuracy, yet improves the area and energy footprints [4]. The posit representation is a *superset* of the floating-point format and can serve as a drop-in replacement for floating-point arithmetic.

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 [17]. However, no such framework exists for hardware exploration. There is a dire need for an easily reconfigurable hardware platform for early-stage design space exploration of posit arithmetic for various applications. With ever-increasing popularity and a conducive open-source ecosystem, we believe that RISC-V [7] is an excellent vehicle to have a quintessential framework supporting posit arithmetic empiricism. We chose the BSV high-level HDL [1] as the implementation language to enable rapid design space exploration through an easy reconfiguration of the hardware platform. The position of the proposed framework called *Clarinet* in the platform design cycle is delineated in Fig. 1 along with the posit arithmetic core, *Melodica*. The major contributions of this paper are:

- We present *Clarinet*; a floating-point arithmetic enabled CPU-based framework for posit arithmetic empiricism. Clarinet is based on the RISC-V ISA (with custom instructions for posit arithmetic), and is derived from the open-source *Flute* core developed by *Bluespec Inc* [2]. The Clarinet framework also features a customized RISC-V gcc tool-chain to support the new instructions.
- We present *Melodica*, a reconfigurable posit arithmetic core that supports fused-multiply-accumulate (FMA) with quire functionality, and type-converters between floating-point, posit and quire data representations.

- Through Clarinet, we also present a new usage model where posits and floating-point can coexist as independent types cleanly, allowing applications to be ported more easily to posits when they offer an advantage.
- Finally, we investigate applications in the domain of linear algebra and computer vision to show the effectiveness of Clarinet as an experimental platform. For *five* different applications, we demonstrate that Clarinet supports trade-off-analyses between performance, power, area, and accuracy for applications of interest. We also outline the ease-of-use aspect of Clarinet.

### Why add-on?

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.

As of now, we support a limited number of operations and quire to carry-out experimental studies for our applications. We plan to extend Melodica with more functionality. 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. The organization of the paper is as follows. In Section 2, we discuss posit, quire and float formats, the Flute core, and some of the recent implementations of posit arithmetic. Clarinet is described in Section 3, and Melodica in Section 4. Application analyses and benchmarking are presented in Section 5. Experimental setup and results are discussed in Section 6. We conclude our work in Section 7.

## 2 BACKGROUND AND RELATED WORK

### 2.1 Background

**2.1.1 Posits.** A posit number is defined by two parameters: the width of the posit number,  $N$ , and the maximum width of the exponent field,  $es$ . One of the important advantages of the posit number format is that we can vary  $es$  to trade-off between greater dynamic range (larger  $es$ ) and greater precision (smaller  $es$ ).

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 1, where  $r$  is the number of leading bits in the regime.
- 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.

For a number represented in the posit format, its value is as per the equation 2.

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

$$value = (-1)^s * (2^{2^{es}})^k * 2^{exp} * 1.f \quad (2)$$

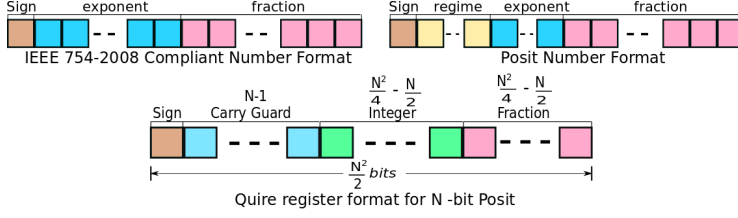


Fig. 2. IEEE 754-2008 floating-point, posit, and quire register format for N-bit posits

Posits do not have a representation for NaNs, or separate representations for  $\pm 0$  and  $\pm \infty$ . Posits recognize only two special cases – zero and not-a-real (NaR), and support one rounding mode Round-to-Nearest-Even (RNE). Posit number system shows better accuracy around 1 than floating-point of the same size [6]. Table 1 summarizes the different bit representations with posits, using 8-bit posits as an example. Posit and floating-point formats are depicted in Fig. 2.

**2.1.2 Quire.** The *quire* is a fixed-point register that serves the purpose of accumulation like a Kulisch accumulator [14]. The quire for a given posit-width is sized to represent the smallest posit squared, and the largest posit squared without any overflow. When the quire is used as an accumulator for a series of steps, it allows computation without intermediate rounding. The size of an N-bit quire is determined by  $\frac{N^2}{2}$  where N is the posit number width (Fig. 2).

#### Numerical Examples:

Numerical examples of pi and dot product are shown in Fig. 3. For pi calculation, 32-bit quire (q32) converges better compared to 32-bit posit (p32) or 32-bit floating-point (f32). In all our experiments, we have used 64-bit floating-point as a reference. For iteration 11 there is a dramatic increase in normalized error (compared to 64-bit floating-point) for p32 and f32, but only a marginal increase in error for q32.

In dot product, we use randomly generated vectors in the range of i)  $2^0$  to  $2^{-10}$ , and ii)  $2^0$  to  $2^{-1000}$ . We choose these ranges since the numbers are representable in all the formats. q32 outperforms p32, f32 and q24. Absolute error in f32 is observed to be 9.9534475E-08, in p32 it is 1.0127508E-08, in q24 it is 8.14790212E-07, and in q32 it is 2.676927E-09 in dot product of 10000 element vectors with the input data range of  $2^0$  to  $2^{-1000}$ . The loss of precision in q24 is close to one digit only while the reduction in the total bit-width is 8 bits compared to f32. The elaborated experiments on the dot product are depicted in section 5.2. These preliminary experiments on pi calculation and dot product outline the superiority of using quire over 32-bit floating-point arithmetic.

**2.1.3 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 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

Table 1. Posit Bit Patterns (8-bit Posits)

| Bit pattern | Value             |
|-------------|-------------------|
| 0000_0000   | 0                 |
| 1000_0000   | $\pm \infty$      |
| All others  | as per equation 2 |

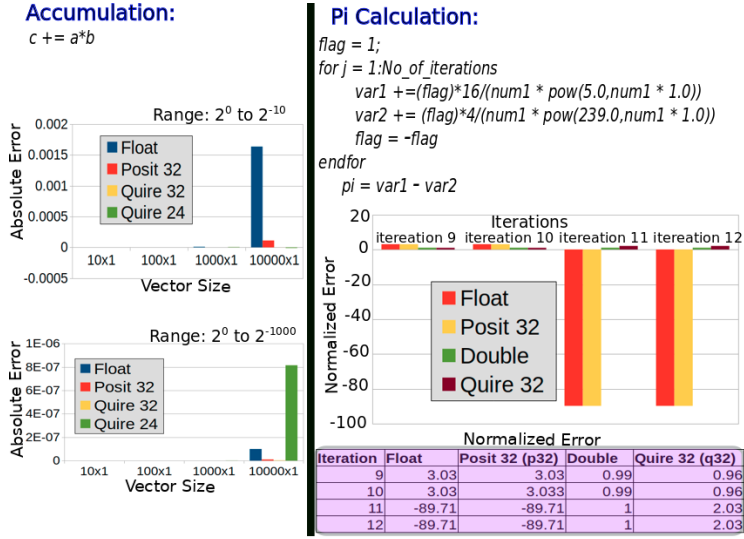


Fig. 3. Examples of accumulation and  $\pi$  calculation with posit, quire, and floating-point representation

of the RISC-V ISA [7]. 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.

## 2.2 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 [11] and [10]. In [10], the authors have covered the design of a parametric adder/subtractor, while in [11], 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 [12]. 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 [4]. A major drawback of the generator presented in [4] is that it is a non-pipelined design resulting in low operating frequency for large bit-width adders and multipliers.

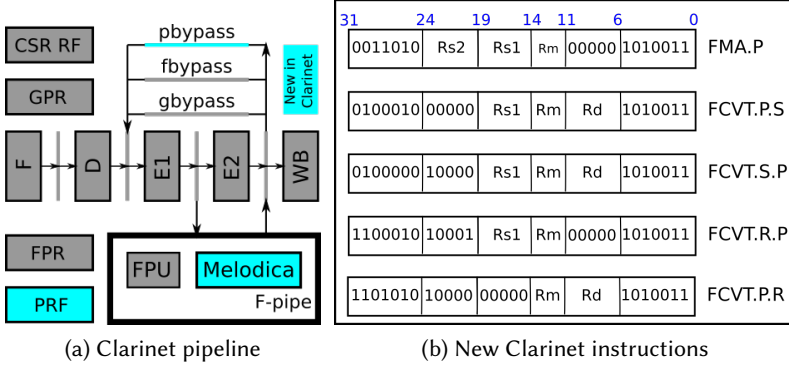


Fig. 4. Clarinet

*Cheetah* presented in [15] discusses the training of deep neural network (DNN) using posits. We believe that the architecture presented in [15] 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 [22][19]. More recently, [20] 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.

### 3 CLARINET

The system comprises two main components – *Melodica*, a parameterizable Posit Numeric Unit that implements quire described in Section 4, and *Clarinet*, a RISC-V CPU that is enhanced with special instructions for posit arithmetic and a dedicated posit register file (PRF).

#### 3.1 Clarinet organization

Clarinet’s organization is illustrated in Fig. 4a. The starting point for Clarinet was a Flute CPU core, configured with the RV32IMAFc variant of the RISC-V ISA [7].

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.

### 3.2 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. 4b 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 [7], 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-point 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 5.1. From our experiments as illustrated in Fig. 3, 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.

### 3.3 Integrating the quire

As indicated in Fig. 2 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.

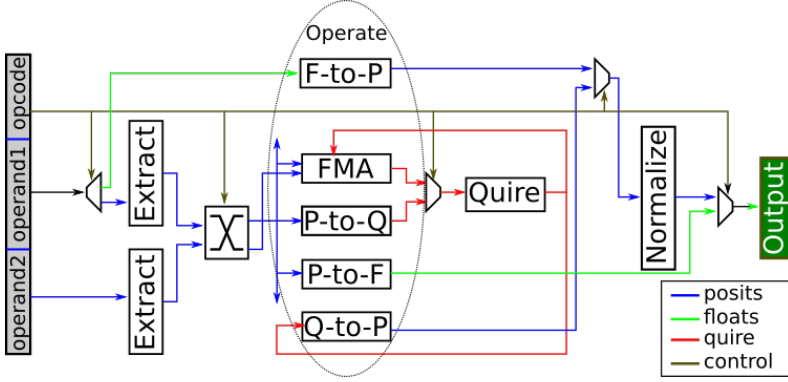


Fig. 5. Melodica

### 3.4 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. 4a and handles only posit values.

## 4 MELODICA

Melodica is a 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 ( $FW$ ). Melodica supports any size float input but for Clarinet float-width is set to 32. 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) into the quire, 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. 5. 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.

### 4.1 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 *r* and the *exp* field as given by equation 3, where maximum posit scale width is *psw* and maximum posit fraction width is *pfw*.

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

Extraction operates on the *N*-bit input posit word and generates four outputs as illustrated in Fig. 6a. In Fig. 6 and Fig. 7 detection is defined using the *det* block. The steps involved are: i) check for special cases like 0 and  $\pm\infty$  to determine zero-infinity flag (*zif*). If the sign of the posit number is negative, perform a two's complement of the remaining *N*-1 bits, ii) compute *k* using equation 1. The *r* 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, iii) determine the value of the exponent. The *exp* may have up to *es* bits. We 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 *exp* field continues until the end of the posit, iv) calculate *scale* using equation 3. Calculate the *f* field by extracting remaining bits (if any) after the *exp*.

## 4.2 Normalize

The normalization illustrated in Fig. 6b, 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: i) computation of *k* and the *exp* bits from the *scale* value based on the equation 3, ii) construction and concatenation of the *r* and *exp* fields where regime bits are calculated based on run-length of 0s and 1s, iii) shift the *f* field by shifting *r* and *exp* field. The concatenated value is rounded to the nearest even depending on the *f* bits truncated in the previous stage and the truncation flag (*tf*), iv) 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.

## 4.3 Operate

This stage in Melodica performs computations on the input operands. The particular operation performed is based on the opcode dispatched to Melodica. The five operations that are supported by Melodica are divided into two categories – type converters and compute.

*Compute Operation – Fused-Multiply-Accumulate (FMA)*. The FMA, illustrated in Fig. 7a computes the product of two input posit numbers and adds the result to the quire. Using quire as an accumulator helps preserve the overflow or underflow bits without the need to round the results.

The FMA is performed as follows: i) the hidden bit information is added to the input *f* fields depending on the posit value. Also check for corner cases – NaN, 0, and  $\pm\infty$ , ii) the *f* and sign fields are multiplied using integer multipliers, and the scales are added to create the scale of the output, iii) 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 two's complement is performed, iv) 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.

*Type-converter – Float-to-Posit (F-to-P)*. The F-to-P block converts a float input to posit format as depicted in Fig. 7b. The conversion may result in a loss of precision when using narrower posit

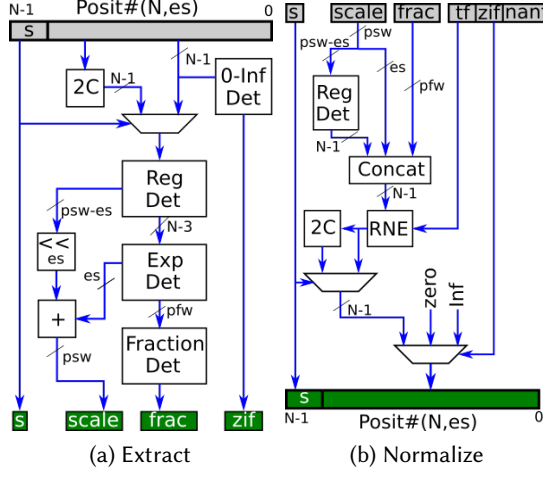


Fig. 6. Extract and Normalize components in Melodica

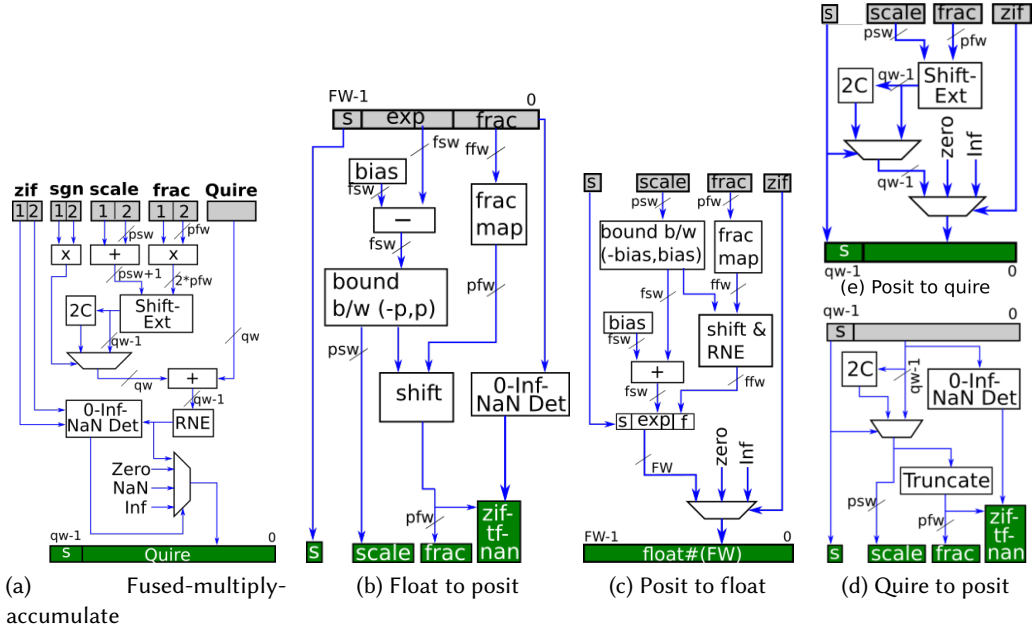


Fig. 7. Melodica Operations

types. For a number represented in the posit format, its value is as per equation 4, where  $fsw$  is the exponent width and  $ffw$  is the fraction width of float.

The fixed length  $f$  field is interpreted directly from the input operand and mapped to the field in posit format using equation 2 and 4. 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. The output of the F-to-P needs normalization before write-back to the PRF.

$$value = (-1)^{sign} * 2^{exp-bias} * 1.f \quad (4)$$

*Type-converter – Posit-to-Float (P-to-F).* The P-to-F block converts a posit input to float format as illustrated in Fig. 7c. The P-to-F block receives its input after the Extract stage, and its output is directly sent to the FPR in Clarinet. Depending on the configuration parameters for Melodica this operation may result in a change in widths between source and target types.

The main operation involved in the conversion is to bound the scale between (-bias, bias) for the target float format, and truncation of the f field if the field-width for the target format is narrower than the source format using equation 2 and 4. In addition the converter also checks for special cases for the target format (zero, NaN, and  $\pm\infty$ ).

*Type-converter – Quire-to-Posit (Q-to-P).* The Q-to-P block converts a value in the quire format to posit format so that it can be written to the PRF after normalization. After adjusting for a negative value, the scale and f are extracted from the quire as illustrated in Fig. 7d. The truncation flag (tf) which is generated from truncating the fraction value, is sent to the Normalize block to control rounding to RNE.

*Type-converter – 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. 7e.

## 5 CASE STUDIES

We cover case studies on some of the linear algebra kernels and optical flow in computer vision. We look into application kernels that are rich in floating-point arithmetic operations. For matrix operations, we develop a subset of basic linear algebra subprograms (BLAS) and linear algebra package (LAPACK) using SoftPosit for the analyses [17]. Based on the investigation, we arrive at a suitable arithmetic size for each of the kernels in BLAS and LAPACK, and optical flow estimation using Lucas-Kanade method. We use this information to tweak parameters in Clarinet to arrive at a customized Clarinet instance.

### 5.1 Using Clarinet - A Simple Example

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 2 is of a simple case where a user loads two, 32-bit floating-point

Table 2. An ASM example on Clarinet-Melodica

| Sl. 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.ft2, p2   | FPR[2] <- 0x41a00000           | Execute P-to-F on p2. Result in FPR ft2   |

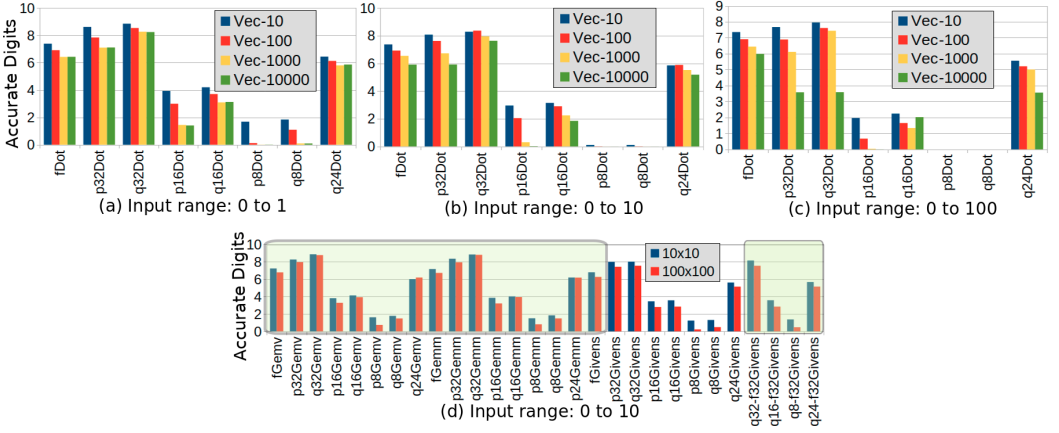


Fig. 8. Accurate number of digits with different data types in BLAS and LAPACK routines

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.

## 5.2 BLAS and LAPACK

The BLAS and LAPACK are encountered in a wide range of engineering and scientific applications. In BLAS, we consider dot product (xDot), matrix-vector (xGenv), matrix-matrix operations (xGemm), and in LAPACK, we consider Givens rotation (xGivens) where x denotes the data type used for the implementations. For all the matrix operations, we implement nine different versions using different data types for comparison and use 64-bit floating-point implementation as a reference. We randomly generate numbers using rand() function. Since, Clarinet supports quire and FMA, we emphasize more on quire based implementations with using SoftPosit for our analyses. To calculate the error in xDot, we average the relative error over 100K runs. To calculate error in xGenv and xGemm, we use  $\frac{\|\hat{x}-x\|_2}{\|x\|_2}$  and  $\frac{\|\hat{A}-A\|_2}{\|A\|_2}$ , respectively where x and A are the operations computed in 64-bit floating-point and  $\hat{x}$  and  $\hat{A}$  are the operations computed using SoftPosit.

The accurate digits in the different implementations are shown in Fig. 8. In dot product, the 32-bit quire (q32Dot) results in 8.8 accurate digits for small ( $<10$ ) input vector sizes in range of 0 to 1 (Fig. 8a). For large vectors ( $<10000$ ) in the same range, the number of accurate digits drop to 8.2 which is a drop of 6.8%. In the same input range, we observe a drop of 12.3% in fDot, 17.4% in p32Dot, and 9.3% in q24Dot. For the input vector range of 0 to 10 and the sizes of 10 to 10000, we observe a similar trend (Fig. 8b). Varying the range of input vectors impacts the accuracy heavily, specifically for large vectors. We observe a drop in the number of accurate digits by 55.6% in q32Dot, 53.94% in p32Dot, and 36.3% in q24Dot while in fDot it is 18.63% (Fig. 8c). The drop in accuracy is due to the fact that the posit and quire are more accurate for the values around 1.0 while as the input range shifts from 1.0 the accuracy deteriorates.

A similar trend is observed in xGenv, xGemm, and xGivens routines for increasing matrix sizes and varying ranges (Fig. 8d). A key observation here is that in p32Givens and q32Givens routines where

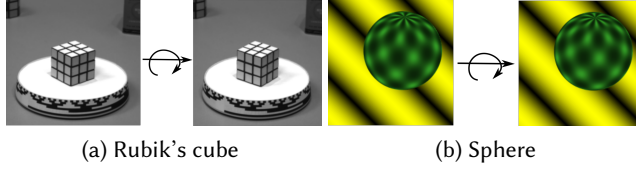


Fig. 9. Dataset for Lucas-Kanade

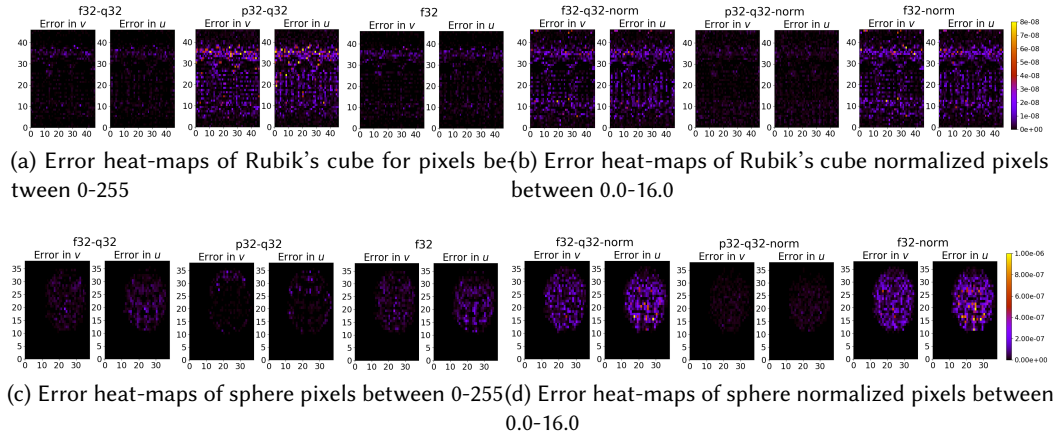


Fig. 10. Error heat-maps for Rubik's cube and sphere

we observe the number of accurate digits are significantly higher (8.2 and 8.8 respectively) compared to fGivens (6.79). The shaded region in Fig. 8 represents the routines that can be executed on the current version of Clarinet due to absence of posit addition, multiplication, division hardware. For implementation of routines in software we have used floating-point in conjunction with quire. For example, q32-f32Givens is implementation of Givens rotation using combination of 32-bit quire and 32-bit floating-point arithmetic. The implementation yields similar accuracy as q32Givens since the majority of the operations are dominated by quire. In BLAS routines, the 100% of the arithmetic operations can be implemented using only quire. Based on the accurate digits and supported arithmetic in Clarinet, we can arrive at the quality of Clarinet which is further discussed in Section 6.2.

### 5.3 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 5.

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \quad (5)$$

The Lucas-Kanade method is used to calculate the optical flow for consecutive frames of rotating objects which are given in Fig. 9. We 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:

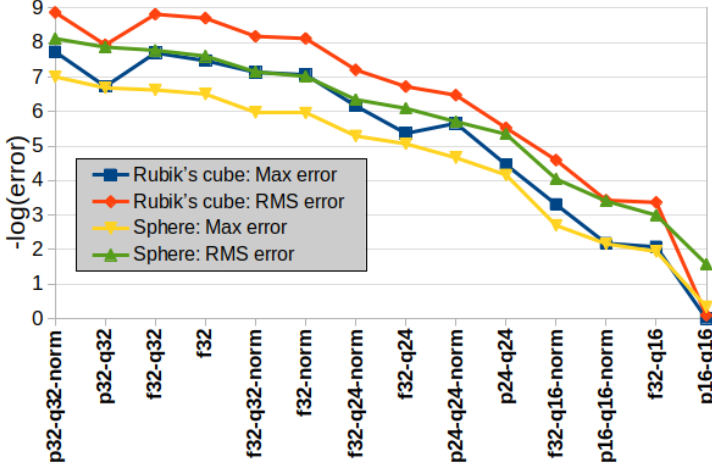


Fig. 11. Maximum error and RMS error with respect to 64-bit floating-point

i) 32-bit single-precision floating-point arithmetic (f32), ii) 32-bit single-precision float arithmetic combined with N-bit quire 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 we have normalized (norm) grey-scale pixel values (0 to 255) to (0.0 to 16.0).

From the heat-maps in Fig. 10a 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. 11. The y-axis value for RMS error in Fig. 11 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 3 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.

Table 3. RMS error in optical flow

| 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 |

## 6 EXPERIMENTAL RESULTS

### 6.1 Implementation Setup

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 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 3.2.
- **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 4, 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 4 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, we found it more instructive to highlight the cell switching power which amplifies the effect each configuration has on dynamic power dissipation.

### 6.2 Quality of Clarinet

Quality of Clarinet (QoC) is defined using equation 6.

$$QoC = \frac{A_{base} - \frac{A_{iuc}}{K}}{A_{base}} \times 100 \quad (6)$$

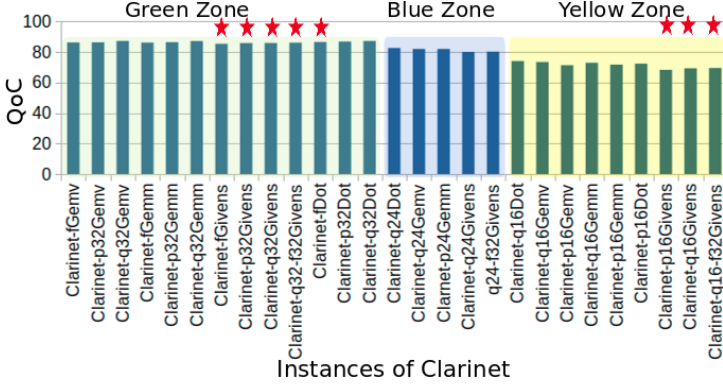


Fig. 12. Quality of Clarinet instances for a subset of BLAS and LAPACK

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 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. We segregate the QoC 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  $0.75 \leq \text{QoC} < 0.85$  and in yellow zone the QoC is less than 0.75. The QoC for different BLAS and LAPACK routines is shown in Fig. 12. The routines marked with the red star can not be executed on the current implementation of Clarinet due to presence of posit square-root and division in the routines. The QoC can not be defined as a constant for any instance, but it is a function of accuracy of the executing software application. The QoC for Rubik's cube and sphere frames is depicted in Fig. 13. The routines that involve addition and multiplication of posits are implemented using FMA operation of Clarinet as described in Section 5.1. 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, we notice an order improvement in accuracy. As the accuracy varies the QoC varies and depending on the accuracy requirements of the application, a suitable instance of Clarinet can be considered.

### 6.3 Disclaimer and Limitations

**Disclaimer:** We notice that the implementation of qX2\_fdp\_add/sub in SoftPosit uses 32-bit posit storage underneath while performing 24-bit posit accumulations that results in a 512-bit quire register. In hardware, we provide support for the 288-bit quire register for the accumulation of

Table 4. Comparison of total cells and area for different Clarinet configurations

| 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                     |



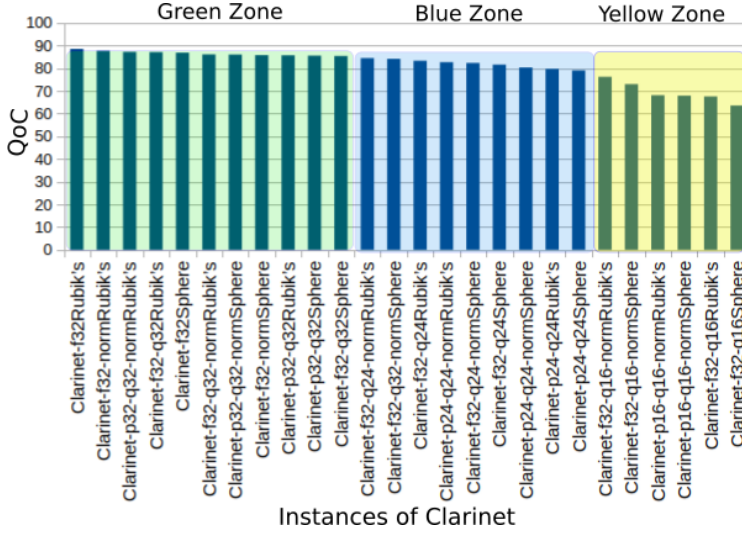


Fig. 13. Quality of Clarinet instances for Lucas Kanade

24-bit posits. Based on our interactions with the developers of the SoftPosit library, we identify that it is fair to compare the accuracy of 24-bit quire (accumulation of 24-bit posits) in software and hardware.

**Limitation 1:** In its present form, Clarinet can execute applications that contain multiply, add, and multiply-accumulate operations on posits while division and square-root are not supported. The absence of the operations limits the application analyses and execution unless square-root and division are implemented in software.

**Limitation 2:** While Melodica has seen extensive unit-level verification, further system-level tests are in progress on Clarinet.

## 7 CONCLUSION

We presented Clarinet – an open-source hardware-software framework that allows posit and floating-point arithmetic to coexist in experiments. A posit arithmetic core called Melodica was presented, and the design components of the core were described. Melodica is the first-ever posit arithmetic core supporting quire that is integrated into a RISC-V CPU. We delved into case studies on basic matrix operations and the Lucas-Kanade optical flow method. The analyses of the kernels and application helped us to quantify the quality of different Clarinet instances. The advantages of different arithmetic formats were identified based on the accuracy of the numerical results. Finally, we presented synthesis results for Clarinet and outlined some of the limitations of the current implementation. We enable the researchers with a consolidated framework that can be used for experimental studies on posit arithmetic. We demonstrated high-quality, medium-quality and low-quality implementations on Clarinet by segregating the implementations in green, blue and yellow zones. The implementations that fall in green zones are high quality while the implementations that fall in yellow zone are of poor quality. Our quality metric incorporated the area footprint of the platform. In the future, we plan to extend Melodica to support more operations, and also to explore its use as a posit-enabled accelerator.

## REFERENCES

- [1] Bluespec Inc. 2020. BSV HL-HDL. <https://github.com/B-Lang-org/bsc>. (2020).
- [2] Bluespec Inc. 2020. FLUTE RISC-V Core. <https://github.com/bluespec/Flute>. (2020).
- [3] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell. 2019. Bfloat16 Processing for Neural Networks. In *2019 IEEE 26th ARITH*. 88–91. <https://doi.org/10.1109/ARITH.2019.00022>
- [4] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers. 2018. Parameterized Posit Arithmetic Hardware Generator. In *2018 IEEE 36th ICCD*. 334–341. <https://doi.org/10.1109/ICCD.2018.00057>
- [5] Gary Chun Tak Chow, Anson Hong Tak Tse, Qiwei Jin, Wayne Luk, Philip H.W. Leong, and David B. Thomas. 2012. A Mixed Precision Monte Carlo Methodology for Reconfigurable Accelerator Systems. In *Proceedings of the ACM/SIGDA ISFPGA (FPGA &Ž12)*. Association for Computing Machinery, New York, NY, USA, 57–66. <https://doi.org/10.1145/2145694.2145705>
- [6] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. 2019. Posits: The Good, the Bad and the Ugly. In *Proceedings of the CoNGA 2019 (CoNGA &Ž19)*. ACM, New York, NY, USA, Article Article 6, 10 pages. <https://doi.org/10.1145/3316279.3316285>
- [7] Editors Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation.
- [8] A. Guntoro, C. De La Parra, F. Merchant, F. De Dinechin, J. L. Gustafson, M. Langhammer, R. Leupers, and S. Nambiar. 2020. Next Generation Arithmetic for Edge Computing. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1357–1365.
- [9] Gustafson and Yonemoto. 2017. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomput. Front. Innov.: Int. J.* 4, 2 (June 2017), 71–86. <https://doi.org/10.14529/jsfi170206>
- [10] M. K. Jaiswal and H. K. . So. 2018. Architecture Generator for Type-3 Unum Posit Adder/Subtractor. In *ISCAS 2018*. 1–5. <https://doi.org/10.1109/ISCAS.2018.8351142>
- [11] M. K. Jaiswal and H. K. . So. 2018. Universal number posit arithmetic generator on FPGA. In *DATE 2018*. 1159–1162. <https://doi.org/10.23919/DATE.2018.8342187>
- [12] M. K. Jaiswal and H. K. . So. 2019. PACoGen: A Hardware Posit Arithmetic Core Generator. *IEEE Access* 7 (2019), 74586–74601. <https://doi.org/10.1109/ACCESS.2019.2920936>
- [13] N. Kapre and A. DeHon. 2012. SPICE<sup>2</sup>: Spatial Processors Interconnected for Concurrent Execution for Accelerating the SPICE Circuit Simulator Using an FPGA. *IEEE TCAD* 31, 1 (Jan 2012), 9–22. <https://doi.org/10.1109/TCAD.2011.2173199>
- [14] Ulrich W. Kulisch. 2002. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, Berlin, Heidelberg.
- [15] Hamed F. Langroudi, Zachariah Carmichael, David Pastuch, and Dhireesha Kudithipudi. 2019. Cheetah: Mixed Low-Precision Hardware & Software Co-Design Framework for DNNs on the Edge. (2019). arXiv:cs.LG/1908.02386
- [16] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl. 2014. Make it real: Effective floating-point reasoning via exact arithmetic. In *DATE 2014*. 1–4. <https://doi.org/10.7873/DATE.2014.130>
- [17] Cerlane Leong. 2018. SoftPosit Version 0.4.1rc. (22 Nov. 2018). <https://gitlab.com/cerlane/SoftPosit>
- [18] J. Liao, M. Jost, M. Schaffner, M. Magno, M. Korb, L. Benini, F. Tebbenjohanns, R. Reimann, V. Jain, M. Gross, A. Militaru, M. Frimmer, and L. Novotny. 2019. FPGA Implementation of a Kalman-Based Motion Estimator for Levitated Nanoparticles. *IEEE TIM* 68, 7 (July 2019), 2374–2386. <https://doi.org/10.1109/TIM.2018.2879146>
- [19] Jinming Lu, Siyuan Lu, Zhisheng Wang, Chao Fang, Jun Lin, Zhongfeng Wang, and Li Du. 2019. Training Deep Neural Networks Using Posit Number System. (2019). arXiv:cs.LG/1909.03831
- [20] Sugandha Tiwari, Neel Gala, Chester Rebeiro, and V. Kamakoti. 2019. PERI: A Posit Enabled RISC-V Core. (2019), 1–14. arXiv:1908.01466 <http://arxiv.org/abs/1908.01466>
- [21] A. Volkova, M. Istvan, F. De Dinechin, and T. Hilaire. 2019. Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study. *IEEE Trans. Comput.* 68, 4 (April 2019), 597–608. <https://doi.org/10.1109/TC.2018.2879432>
- [22] H. Zhang, J. He, and S. Ko. 2019. Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications. In *ISCAS 2019*. 1–5. <https://doi.org/10.1109/ISCAS.2019.8702349>