A variable precision hardware acceleration for scientific computing

Andrea Bocco

N°d'ordre NNT : 2020LYSEI065

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON
Opérée au sein de :
CEA Grenoble

Ecole Doctorale InfoMaths EDA N17° 512
(Informatique Mathématique)

Spécialité de doctorat : Informatique

Soutenue publiquement le 29/07/2020, par :

Andrea Bocco

A variable precision hardware acceleration for scientific computing

---

Devant le jury composé de :

Frédéric Pétrot		**Président et Rapporteur**
Professeur des Universités,	TIMA, Grenoble, France	
Marc Dumas		**Rapporteur**
Professeur des Universités,	École Normale Supérieure de Lyon, France	
Nathalie Revol		**Examinatrice**
Docteure,	École Normale Supérieure de Lyon, France	
Fabrizio Ferrandi		**Examinateur**
Professeur associé,	Politecnico di Milano, Italie	
Florent de Dinechin		**Directeur de thèse**
Professeur des Universités,	INSA Lyon, France	
Yves Durand		**Co-directeur de thèse**
Docteur,	CEA Grenoble, France	

" Arithmetics is being able to count up to twenty without taking off your shoes. "

Mickey Mouse

INSA-LYON

Abstract

InfoMaths
CEA Grenoble

Doctor of Philosophy

A variable precision hardware acceleration for scientific computing

by Andrea BOCCO

Most of the Floating-Point (FP) hardware units support the formats and the operations specified in the IEEE 754 standard. These formats have fixed bit-length. They are defined on 16, 32, 64, and 128 bits. However, some applications, such as linear system solvers and computational geometry, benefit from different formats which can express FP numbers on different sizes and different tradeoffs among the exponent and the mantissa fields. The class of Variable Precision (VP) formats meets these requirements.

This research proposes a VP FP computing system based on three computation layers. The *external* layer supports legacy IEEE formats for input and output variables. The *internal* layer uses variable-length internal registers for inner loop multiply-add. Finally, an *intermediate* layer supports loads and stores of intermediate results to cache memory without losing precision, with a dynamically adjustable VP format. The VP unit exploits the UNUM type I FP format and proposes solutions to address some of its pitfalls, such as the variable latency of the internal operation and the variable memory footprint of the intermediate variables. Unlike IEEE 754, in UNUM type I the size of a number is stored within its representation.

The unit implements a fully pipelined architecture, and it supports up to 512 bits of precision, internally and in memory, for both interval and scalar computing. The user can configure the storage format and the internal computing precision at 8-bit and 64-bit granularity This system is integrated as a RISC-V coprocessor. The system has been prototyped on an FPGA (Field-Programmable Gate Array) platform and also synthesized for a 28nm FDSOI process technology. The respective working frequencies of FPGA and ASIC implementations are 50MHz and 600MHz. Synthesis results show that the estimated chip area is $1.5mm^2$, and the estimated power consumption is 95mW.

The experiments emulated in an FPGA environment show that the latency and the computation accuracy of this system scale linearly with the memory format length set by the user. In cases where legacy IEEE-754 formats do not converge, this architecture can achieve up to 130 decimal digits of precision, increasing the chances of obtaining output data with an accuracy similar to that of the input data. This high accuracy opens the possibility to use direct methods, which are more sensitive to computational error, instead of iterative methods, which always converge. However, their latency is ten times higher than the direct ones. Compared to low precision FP formats, in iterative methods, the usage of high precision VP formats helps to drastically reduce the number of iterations required by the iterative algorithm to converge, reducing the application latency of up to 50%. Compared with the MPFR software library, the proposed unit achieves speedups between 3.5x and 18x, with comparable accuracy.

Acknowledgements

To begin with, I would like to thank my tutor from CEA-Grenoble, Engr. *Yves Durand*, and my supervisor from INSA-Lyon, Prof. *Florent de Dinechin*, for the invaluable technical knowledge that they transmitted to me and the motivational support throughout my thesis work. I also want to remember all colleagues in the LSTA lab in CEA-Grenoble and the wonderful time spent at the workplace with them. In particular, I want to thank *Vincent Mengue* and *Yvan Miro* for helping me with the synthesis of my designs, to *Cesar Fuguet Tortolero*, *Erich Guthmuller*, and *Anthony Philippe* for their support in hardware architectures and FPGA design, to *Tiago Trevisan Jost* for his active collaboration to this project and for providing me software and compiler support for this work, and to *Simone Bacles-Min* and *Julie Dumas* to have played an active role in teaching me the French language.

Many thanks to *Roman Gauchi*, *Valentin Egloff*, *Maxime Montoya*, and *François Dolique* to help me to come out of several technical problems during my work and with whom I had an excellent collaboration. I want to thank *Maxcence Bouvier*, *Sota Sawaguchi*, *Adrian Evans*, and *Pascal Vivet* for helping me review this work and the published papers relating to it. A special thanks go to *Fernando Barrera Cervantes*, *Gabriele Giachin*, *Marco Arborio*, *Giulio Milici*, and *Davide Pala*, who turned out to be special friends in this adventure away from home. On the same length, I would like to thank all my friends and colleagues at the LSTA lab and from all CEA for supporting me through these three years and for all the good times spent as colleagues or Ph.D. students.

I will be forever indebted to my mother *Franca*, my brother *Stefano*, all of my family for their endless patience, encouragement, and love throughout all my life, especially starting from the beginning of my studies in Turin.

I will never forget my father, *Claudio*, who has always followed me with love and admiration since I was born, and now he follows me from another place, my heart.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
BEZ	**B**ranches **E**qual **Z**ero
BMF	**B**ounded **M**emory **F**ormat
CG	**C**onjugate **G**radient
DFT	**D**ensity **F**unctional **T**heory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DUE	**D**efault **U**num **E**nvironment
FDM	**F**inite **D**ifference **M**ethod
FEM	**F**inite **E**lement **M**ethod
FF	**F**lip-**F**lop
FIFO	**F**irst **I**n **F**irst **O**ut
FP	**F**loating **P**oint
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FPU	**F**loating **P**oint **U**nit
FSM	**F**inite **S**tate **M**achine
GB	**G**uard **B**its
GE	**G**auss **E**limination
GPR	**G**eneral-**P**urpose **R**egister
GPRF	**G**eneral-**P**urpose **R**egister **F**ile
GPU	**G**raphics **P**rocessing **U**nits
HLS	**H**igh-**L**evel **S**ynthesis
HW	**H**ard**W**are
IA	**I**nterval **A**rithmetic
IB	**I**nteger **B**its
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
ISA	**I**nstruction **S**et **A**rchitecture
IVP	**I**nitial **V**alue **P**roblem
JA	**JA**cobi
KB	**K**ilo **B**ytes
LSU	**L**oad and **S**tore **U**nit
LUT	**L**ook-**U**p **T**able
MBB	**M**aximum **B**yte **B**udget
MMU	**M**emory **M**anagement **U**nit
ODE	**O**rdinary **D**ifferential **E**quations
PDE	**P**artial **D**ifferential **E**quations
RF	**R**egister **F**ile
RND	**R**ou**ND**ing
RTN	**R**ound **T**o **N**earest (half away from zero)
RTNE	**R**ounded **T**o the **N**earest **E**ven
RAM	**R**andom **A**ccess **M**emory
RDD	**R**oun**D** **D**own
RDU	**R**oun**D** **U**p

RTI	**R**ound **T**o **I**nterval
SDRAM	**S**ynchronous **D**ynamic **R**andom-**A**ccess **M**emory
SM	**S**pectral **M**ethod
SPD	**S**ymmetric and **P**ositive-**D**efinite
SR	**S**tatus **R**egister
SUE	**S**econdary **U**num **E**nvironment
SW	**S**oft**W**are
TLB	**T**ranslation **L**ookaside **B**uffer
UE	**U**num **E**nvironment
ULP	**U**nit in the **L**ast **P**lace
VP	**V**ariable **P**recision
WGP	**W**orking **G**-layer **P**recision

Chapter 1

Introduction

Scientific notation is widely used by mathematicians, physicists, and engineers. Equation 1.1 depicts a possible format to express numbers in scientific notation.

$$x = s \cdot m \cdot \beta^e \tag{1.1}$$

A number x, presented in radix β, is made of a sign s, a significand m (also called the mantissa), and an exponent e. The common name used to identify this format is "*Floating-Point*" (FP). In FP, the decimal point divides the integer and the fractional part of the mantissa. Its position in the real axis changes (floats) according to the value of the exponent value. This variability of the decimal point in the real axes justifies the word "floating".

The FP format was standardized in 1985 (and reviewed in 2008) by the Institute of Electrical and Electronics Engineers (IEEE) committee in the IEEE 754 standard [1]. The introduction of this standard stopped the compatibility issues introduced by companies while developing custom FP units (FPUs). The IEEE 754 standard defines arithmetic and interchange formats (Figure 1.1), rounding rules, operations on arithmetic formats, and rules to handle arithmetic exceptions.

FP applications have widely different requirements in terms of arithmetic precision. Consequently, choosing the appropriate data format among the ones offered by the IEEE 754 standard can be difficult.

A wide range of applications shows optimal performance for FP representations that cannot be encoded through one of the standard formats. For instance, many applications in neural networks and signal processing require fewer than 16 bits of representation, and using one of the IEEE-754 formats can be inefficient. Others are sensitive to the accumulation of rounding, cancellation, and absorption, computational errors. Such accumulations can quickly lead to

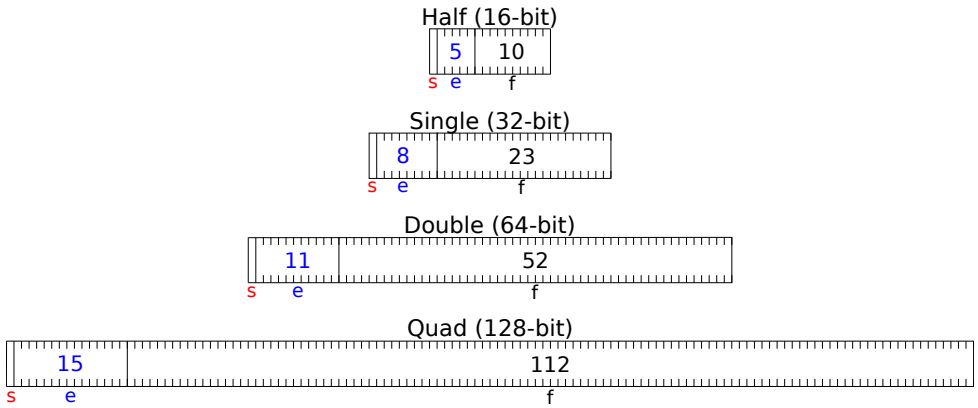


Figure 1.1: IEEE 754 standard formats types: definition of floating-point formats on 16, 32, 64, and 128 bits

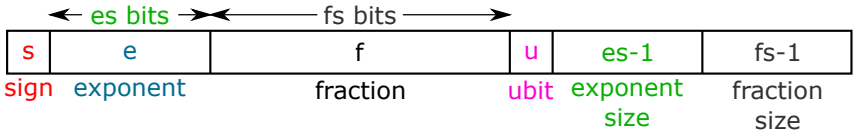


Figure 1.2: The Universal NUMber (UNUM) floating-point format

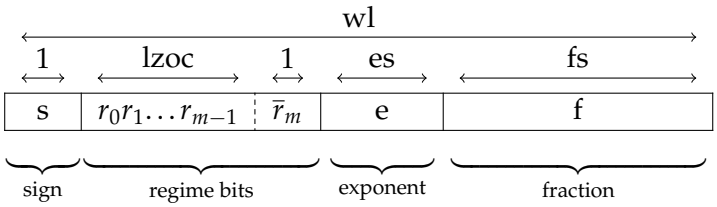


Figure 1.3: The posit floating-point format

entirely inaccurate results. For example, this may happen for linear solvers and experimental-math applications which may only show stability for formats above 128 bits, beyond the capacity of mainstream IEEE-754 hardware support.

This work aims to improve the stability of numerical computations on one side and augment their efficiency in memory occupancy and energy cost on the other side while preserving their stability. The class of *Variable-Precision* (VP) computing groups the techniques used to compensate for the problems mentioned above (overkill, cancellation, rounding, numerical stability, and memory footprint). To reduce the effects of these problems, the precision of variables in algorithms must vary. For this purpose, this work uses VP computing as the primary tool to tune variables precision. Tuning the precision of variables is also appropriate for energetic efficiency because it decreases the data memory footprint, and reduces memory pressure.

VP computing rethinks conventional FP arithmetic in order to adjust the computation precision to the application requirements. VP lets the user explore the tradeoff between result precision, computation latency, and data memory footprint. This tradeoff can be explored by tuning the precision of the data in memory, or by tuning the precision of the computation, in the VP FPU, or both. The term VP computing covers different paradigms such as *Fixed point*, *Mixed precision*, *Extended precision*, *Arbitrary precision*, and *Infinite precision*, which we detail later. Most of them require changing the FP memory format, the hardware used, or the software level. These difficulties give space to explore VP FP formats such as UNUM [2] or Posit [3].

The *UNUM* type I (Universal NUMber, introduced by John L. Gustafson in his book "THE END of ERROR - Unum Computing" [2], Figure 1.2), is a VP FP format which provides variable-length exponent and mantissa fields. For a given FP size, it allows trading exponent bits for mantissa bits when the exponent is near zero. Exponent and mantissa lengths are encoded within the same number. This format supports Interval Arithmetic (IA), which guarantees that the output interval contains the exact answer. It is meant to improve the accuracy and convergence of iterative algorithms.

The Posit format [3] is a tapered precision format (like UNUM type I), but it has fixed bit-length. Differently from UNUM, posit encodes the exponent in magnitude-offset form, and the exponent field does not have a limit in its size.

This work adopts the UNUM type I format to support all the VP techniques since, with it, it is possible to exploit the benefits of having exponent and fraction fields of variable size. The native IA support in UNUM is useful to keep track of the computational error among FP operations. So far, IA was presented positively, and suddenly, it has several drawbacks. In iterative applications, intervals are enlarged among iterations. This behavior, also known as *interval explosion*, can quickly lead to large meaningless intervals. To avoid it, instead to compute a given algorithm directly in IA, IA can be used to bound the computational error at

the end of a given algorithm.

Most modern FP computing systems offer two levels of precision: the one provided by the memory format and the one provided internally by the FPU. There is also a trend to have accumulation registers in FPUs that provide more bits than the memory format [4]. These two levels of precision reduce the effects of the computational errors but only for the so-called fused operations, which are sequences of high precision operations done on the internal registers of the FPU. The main limiting factors in modern hardware architectures are memory size and bandwidth [5], [6]. To control them, the precision of data stored in the main memory becomes that of the memory format. Data arrays should be kept as compact as possible to optimize memory bandwidth. These two limiting factors currently restrict applications to 64-bit memory formats.

However, an iterative linear solver may require to keep vectors with high-precision data among iterations. The precision required for this is greater than the input-output format. However, it is smaller than the internal accumulation registers of the FPU, designed to hide the accumulation of rounding errors.

This thesis aims to improve the balance between numerical stability and data footprint in memory (in particular, the close memory, i.e., the processor cache). This work proposes an innovative computing environment that supports a third FP format that can improve results precision in FP computation. This third format can provide dynamic precision tailored to the algorithm's immediate needs while being able to fit into memory (ideally into the cache, close to the processor). We adopted the UNUM type I format for this purpose.

The impact of varying precision is not of the same nature in all applicative scenarios. For this reason, this work distinguishes cases where the benefit may be analyzed and quantified in terms of an error on the result, through numerical analysis.

This work aims to use VP within iterative kernels for numerical applications, to improve the result accuracy at minimal memory usage cost. Furthermore, it illustrates a VP FP arithmetic coprocessor architecture based on RISC-V (using a RISC-V-RocketChip generator [7]), that:

- Supports legacy IEEE formats for input and output variables.
- Uses variable-length internal registers (up to 512 mantissa bits) for inner loop multiply-add.
- Supports loads and stores of intermediate results to cache memory with a dynamically adjustable precision (up to 512 bits of mantissa).

The system is integrated on FPGA and demonstrated on representative examples.

The work was done in the LISAN laboratory, specialized in multi-core architectures and low power digital design, in CEA-LETI at Grenoble (France), in collaboration with INSA-Lyon (France). The remainder of this document is organized as follows:

Chapter 2 introduces to the reader the variable-precision topic. Firstly, it motivates why Variable-Precision (VP) is needed (and why specific applications may need formats different from the conventional IEEE 754 ones), secondly it describes the benefits of using VP, the application domains, and the impact for applications on high-and low-precision.

Chapter 3 presents the state of the art VP Floating-Point (FP) formats and hardware architectures. This chapter explains what a FP number is and presents the state of the art VP FP formats. It introduces the UNUM and the posit formats, showing how they work and highlighting their differences from the IEEE 754 standard. After this, it presents state of the art in VP FP hardware architectures.

Chapter 4 describes the core of this work: the architecture to support VP computing. It illustrates the motivations of supporting the UNUM type I FP format in main memory, proposing solutions to address some of its pitfalls such as the variable latency of the internal operation and the variable memory footprint of the intermediate variables. After that, it presents

the designed hardware architecture passing through the chosen register file organization, the working model of the unit, the programming model, and how to build compiler support for it. The last section illustrates how it is possible to modify the BLAS libraries to embed this unit in existing computation environments.

Chapter 5 details the microarchitecture of the VP FP computing unit. It presents with a top-down approach the design techniques used to implement the VP coprocessor. It concludes by showing the ASIC synthesis results, and by listing some possible optimizations for the microarchitecture.

Chapter 6 shows how the proposed VP architecture is integrated into an FPGA emulation environment. After that, it illustrates the results obtained out of the VP system from some FP benchmarks for high precision.

Chapter 7 concludes this thesis by highlighting what was learned and suggesting further developments for the work done.

Chapter 2

Motivations for the variable precision

The goal of this study is to propose a Variable-Precision (VP) Floating-Point (FP) computing architecture to improve the balance between numerical stability and data footprint in memory. VP is a class of techniques that exploit a FP representation for real numbers. The dynamic range and the precision may be adjusted dynamically according to the computation needs. This chapter explores the VP concept, introduces the benefits possible to achieve with VP, and shows the applications that can be improved through VP.

VP can be used to minimize the calculation error of an algorithm to an acceptable level. It is possible to reformulate this problem in finding, for a given algorithm, the smallest data format for a set of variables that meets the given error requirement on another set of variables. This class of problems drives the strategy for a priori or a posteriori precision tuning of the variables for a target algorithm, which justifies the introduction of a low-granularity VP FP format in computational systems. Tuning the precision of variables requires evaluating the tradeoff between memory occupation and computational error, which changes significantly among application domains.

This work arbitrarily divides VP applications into two sub-categories addressing *"high precision"* (e.g., 64 bits or more) and *"reduced precision"* (e.g., 64 bits or fewer) problems. Nowadays, there is no standard for what concerns the threshold between high and low precision applications. Nevertheless, these two application domains solve different problems by adopting different solutions.

In this chapter, Section 2.1 shows the benefits of using VP in FP computing. Sections 2.1.1 and 2.1.2 focus on the existing definitions of VP in literature and, Section 2.1.3, focuses on the numerical problems on standard FP computation. Then, Section 2.2 explains why the VP can reduce and track applications' computational error.

This work focuses on scientific computing applications. Thus, Section 2.3 shows representative high-precision application domains with different computational physics and computational chemistry characteristics. These applications are based non-linear equations (e.g., Newton), but their solutions are based on linear algebra algorithms and require an augmented precision. The side effect of augmenting variables precision is that it increases the size of their representation in memory, and therefore it has to be gradual and controlled. This chapter concludes with Section 2.4, which depicts the problems that may arise when a computing system uses VP.

2.1 Benefits of variable precision in Floating-Point computing

Changing the precision of variables in an algorithm can improve stability, optimize memory occupation, and reduce the energy cost of numerical calculations. The impact of varying precision depends on the applications. For example, increasing the precision of variables is meaningful when algorithm stability is concerned. On the contrary, decreasing variables precision, and by

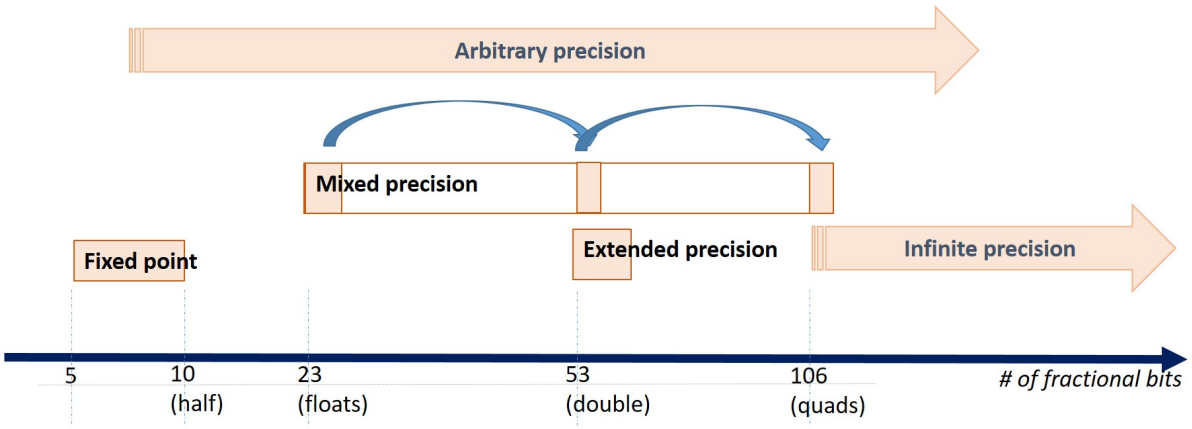


Figure 2.1: Different shades of precision: the precision dynamic of a variable-precision format changes among the used encoding in memory

doing so, reducing their memory footprint, is appropriate to improve the energy efficiency of the computational system.

2.1.1 Precision versus accuracy

Before digging into the VP details, this section clarifies the difference between precision, computational error, and accuracy. *Precision* is the resolution of representation; it is often expressed as a given number of bits or digits. *Computational error* is the distance between an evaluated quantity and its actual value. *Accuracy* is a measure of all the errors among all possible executions. For instance, it can be expressed in bits or digits by computing the log_2 or log_{10} of the worst-case computational error.

$$x = 2^{27} = 1.\underbrace{2893719365}_{p=10} \pm 0.\underbrace{00001}_{a=5}$$

For example, in the equation above, the precision p of the FP number x is ten decimal digits, while its accuracy a, the number of significant digits is five decimal digits (due to ± 0.00001).

In the literature, u or *ulp* [8] (unit in the last place) notes the numbers' precision, and ϵ notes the relative accuracy of a number, the relative error bound. It may be expressed in ulps as it is a convenient way to see how the precision matches the accuracy [8]. The main challenge is to determine how to tune the precision in order to improve accuracy.

2.1.2 Different notions of variable precision

Variable Precision (VP) computing rethinks conventional Floating-Point (FP) arithmetic to adjust the computation precision to the application requirements, exploring the tradeoffs between result precision, computation latency, and data memory footprint. In the literature, several definitions related to VP are used in several contexts. For example, in scientific computing, several libraries support *multiple-precision* [9], which conventionally means "other precisions than the types provided by IEEE 754 standard".

The wording of multiple-precision is ambiguous and covers different techniques. According to the literature, VP computing is mainly divided into five computing techniques: *fixed-point*, *mixed-precision*, *extended-precision*, *arbitrary-precision*, and *infinite-precision*. The reader may find variants as these definitions are frequently interchanged or mixed, for instance, in *trans-precision* computing [10].

These techniques (depicted in Figure 2.1) are not straightforward to implement since they require deep expertise in FP arithmetic and profound modifications in the software tool-chain. Since there is no hardware support for most of these techniques, they are implemented through multiple-precision software libraries (e.g., MPFR [9]) whose use is not straightforward. They use dedicated data types to encode values, and explicit function calls to initialize the declared objects and call the desired (atomic) operations.

Fixed-point computing

Fixed-point computing can be seen as a VP technique when the programmer uses variables with different configurations of integer and fractional bits. In the fixed-point encoding, the frontier between integer and fractional part is static. Thus, fixed-point computing limits the dynamic range and decimal precision. However, it is based on integer operations, which are simpler and faster than the FP ones.

Mixed-precision

Mixed-precision [11] is a VP method which uses the FP types provided by the IEEE 754 standard to adjust the computation precision. It allows controlling, with a lower precision, the round-off errors (for a subset of operations) within a calculation [12]. The most notable mixed-precision implementation is the XBLAS [11] library. It exploits the maximum precision natively available on the host platform or uses a double-double type to represent a value.

Extended-precision

This work defines *extended-precision* as the methods to extend the accuracy of operations using standard IEEE 754 FP types. For example, the "compensated sum" uses intermediate operations to minimize rounding errors and to avoid saturation. It virtually increases the size of the fractional part of a few bits. Another alternative is to use FP expansions [13].

Arbitrary- (or infinite-) precision

Arbitrary-precision, sometimes called *infinite-precision*, uses custom number formats with custom representations or custom sizes for exponent and fraction, that requires custom software or hardware support. Unlike mixed and extended-precision, arbitrary-precision allows gradual sizes for the fractional part, up to a specific limit, which depends on the implementation (for instance, the hardware size or the memory capacity). Infinite-precision has the same characteristics of arbitrary-precision, but it is not constrained to implementation [14].

The state of the art on arbitrary-precision integers is GMP [15]. GMP represents numbers using full words, and it implements basic operations with sophisticated and optimized algorithms in assembly. The state of the art on arbitrary-precision FP is MPFR [9]. It is based on GMP. The bit-length of MPFR numbers is adjustable with a bit granularity, and it is limited only by the available memory. Furthermore, it provides the correct rounding of operations. ARB [16] is a serious contender of MPFR in terms of execution time (but not in the accuracy of each operation).

Other exotic alternatives

In previous paragraphs, precision is only a matter of exponent and fractional sizes. However, there are other numerical representations for arithmetic-intensive applications. For example, signed digit and on-line arithmetic [17] are alternative representations of integers. Others are the Residue Number System (RNS) arithmetic [18], the Logarithmic Number System (LNS)

arithmetic [19]–[21]. These representations require a complex conversion for translating back to binary digit IEEE representation, which is another barrier for their use in a computing environment. Besides, the complexity of operations in RNS and LNS does not scale linearly with precision. Therefore, this study does not consider them part of its scope.

2.1.3 Numerical problems in floating-point computing

Most numerical techniques are built upon continuous mathematics without considering the discrete nature of conventional Floating-Point (FP) formats. However, the implementation of those algorithms uses finite representation once delivered to a conventional FP unit. It only approximates the continuous behavior of algorithms, without any guarantee that the approximation is suitable. Such an implementation can raise issues of *numerical accuracy* (defined in Section 2.1.1) or *numerical stability*. The numerical stability is the property that an algorithm implementation does not amplify errors on its inputs. If the numerical accuracy and stability are not appropriately evaluated, iterative algorithms may not converge to the correct value. However, they are not always straightforward to bound.

Augmenting the precision of variables can solve these issues. The main challenge is not only to improve the accuracy by adjusting the precision but also to control the algorithm's stability. Moler [22] and Higham [23] studied the bound of the computational error in iterative solvers. They were able to distinguish different sources of round-off errors in iterative algorithms. Working with a smaller u usually leads to a better result accuracy in most applications, but there are cases for which the programmer must be aware that it does not [23] (Chapter 2). The value of u, which bounds the relative error to the required value, can be computed with *perturbation theory* [23] (Chapter 3), divided into two main methods. The first one is *forward error analysis*, which considers the input data and examines how each operation magnifies errors. The second one is *backward error analysis*, which considers the output data and asks the question, what is the interval of inputs that could lead to this output. So it does not consider the output error.

Conventionally, algorithms are instrumented after a static analysis of the round-off error, done with the perturbation theory [23]. Few algorithms accept a closed-form to express the round-off error. The ones who accept it are quite simple (e.g., square root, division, and linear matrix operations). For several algorithms, even if these bounds are available, calculating them is more expensive than the algorithm itself (for example, when it involves an explicit value of the condition number) or is too pessimistic.

For instance, the solution deployed in linear algebra for iterative solvers is to periodically evaluate the residual, that is supposed to convey some information about the distance from the exact solution, after each computation phase. If the norm of the residual is not satisfactory (i.e., above a threshold fixed a priori), a new computation macro phase is started. These steps are repeated until the target error bound is met, or the number of iteration overpassed. Finally, another alternative is to profile a priori the application, which is only possible when the data set presents reproducible characteristics. However, it is not the case for general-purpose libraries as the ones used in scientific computing.

Using interval arithmetic to bound the rounding error

Interval Arithmetic (IA) is an alternative to error analysis. With its use, rounding and measurement errors can be bound. In IA, each represented value is an interval. For instance, instead of rounding a variable to 2.0, IA provides an interval saying that the result is between 1.97 and 2.03. The main objective of IA is to get the *inclusion property* in results. In IA, the inclusion property guarantees that, for every input interval, the result of a specific function always contains the one of the same function computed with infinite precision. It is possible to implement this

property by guaranteeing that every arithmetic operator (e.g., +, -, $\times$, and $\div$) respects it.

$$\forall x \in I_x, \forall y \in I_y, x + y \in I_{x+y}$$

The inclusion property is the reason why exact computation can be realized with IA, guaranteeing reliable results. The most common use of IA is to keep track of rounding and computational errors directly during the calculation.

Unfortunately, the quick compounding of errors hinders the use of IA, especially in iterative or convolutional applications, which leads to the well-known problem of *interval explosion*. Since IA enlarges the computed results to guarantee the inclusion property, some variables (e.g., those used as accumulators) may quickly get their interval enlarged at each iteration. These accumulations lead to having results in intervals so large that they get meaningless, e.g., $(-\infty; +\infty)$. In other words, the round-to-nearest (round half away from zero) policy, used in conventional FP computing, compensates somehow the computational error among operations. On the contrary, IA enlarges the output interval systematically to guarantee the inclusion property.

Variable Precision (VP) can be used in IA to reduce the interval explosion effect by increasing the precision of intervals endpoints. VP cannot prevent the interval explosion effect. By augmenting the data precision, VP can help IA to go further in the algorithm, for instance, in iterative computational geometry applications, without having meaningless results. Under these circumstances, if the target algorithm is simple enough, the programmer does not need to rethink the code. It still obtains an interval of confidence in output that respects the inclusion property. The state of the art for VP IA computing is the MPFI software library [24].

IA can also be used to fix control code constructs in FP algorithms. A typical example of ambiguity in the control code is a condition that checks the equality of a variable with the value zero with the '==' operator.

```
if ( op(var1,var2)==0.0 ) dotrue(); else dofalse();
```

An error in computing the condition may significantly impact the precision of the final result. There are some cases when this condition is meaningless since `op(var1,var2)` may be affected by the cancellation effect. Thus, its actual value is not zero. In this case, increasing the precision of the result may not be enough. The solution proposed by this work is to adopt IA as a tool to avoid this kind of problem.

With IA, the previous problem can be formulated more solidly by checking whether the result of `op(var1,var2)` contains the value `0.0`, rather than checking if it is precisely equal to zero. With this formulation, it is possible to instrument the code to ensure that 0 is a possible value of `op(var1,var2)`. Unlike for scalars, IA does not have simple three-way comparisons. Other cases, such as "do two intervals contain both the value x?" have to be considered. This study does not address this problem. For more details about interval comparisons, please refer to the IEEE 1788-2015 standard for interval arithmetic [25].

Another classic problem of IA is that $x - x \neq 0$. In IA, this operation outputs an interval containing zero. Operations among intervals containing zero magnify the interval explosion effect.

IA can be useful in several applications. As a general rule, the applications which benefit from using IA, exploiting the inclusion property to gain reliability on the result, are the ones that do not have a long chain of operations (e.g., non-iterative applications). Among other applications, this work identified two main categories of such applications. The first relates to *control algorithms*, e.g., the control code of a robot which has to guarantee the integrity of the robot and the people surrounding it. This category requires large intervals that do not require high precision to check the domain of validity of internal variables. The second category is related to *computational error* computation. This category requires small intervals with high precision to evaluate the computational error of a given algorithm.

2.2 Variable precision: improving and tracking applications computational error

There are several applications where the numerical stability is a primary concern and where the memory format used to store intermediate variables drives the computational error. These applications partially motivate the choice of having multiple Floating-Point (FP) formats in the IEEE 754 standard. However, these formats have two main limitations justifying the need for a Variable-Precision (VP) FP memory formats:

1. They have bit-sizes which are limited to powers of two.
2. They cannot have bit-sizes beyond 64 bits (or 128 for the new standard version).

For the first point (1), this choice keeps variables aligned in memory on power-of-two boundaries. This alignment simplifies the data organization in memory, the hardware, and compiler design. Despite, increasing to the "next" IEEE 754 FP format (e.g., from 32-bit `float` to 64-bit `double`) doubles the size of the variables, which increases the cache miss rate and degrades the performance of the executed program. A smaller increase of the fractional part would be sufficient in many applications and would lead to a smaller memory footprint of FP variables.

With a VP format, the programmer can *tune* the memory footprint of FP variables with finer granularity while having better control over their precision and on the bound of their computational error. In this way, FP variables have a minimal memory footprint concerning the IEEE standard. They do not need to be doubled, and they can still be stored on contiguous addresses in memory. Minimizing the memory footprint of variables maximizes the cache hit rate and, as a consequence, the system performance. The gain in performance is due to the reduced memory footprint of variables, including the more compact stack region required by the compiler to move data among functions.

For the second point (2), every time a computation needs a higher (or different) precision than the ones provided by the hardware, software libraries (e.g., MPFR [9]) must be used. These libraries encode the FP data on complex data structures, and library function calls perform operations between them. Concerning conventional floats, the usage of these libraries makes the code writing more complex, and the vast memory footprint required by FP data, encoded as structures, makes the program execution slow. These two points motivate the research on VP FP formats different from the IEEE 754 standard.

2.2.1 Variable precision to bound the computational error

The capability to bound the computational error in an expression is the essential requirement for the forward and backward error analysis. These error analysis techniques define the precision of the variables in FP algorithms. They were used by Higham [23] to bound the computational error of iterative solvers.

When focusing on iterative solvers, computing the residual with higher precisions allows us to decrease the algorithm's computational error [23], [26]. There are several techniques to do that. The first technique is to increase the computing precision inside the computing unit (like Kulisch [4]). Unfortunately, it is not sufficient in the cases where the number of variables that require high precision is greater than the number of extended-precision accumulators available in the computing unit.

The second technique is to store extended-precision accumulators of the computing unit in memory. Some VP formats, which are compatible with the memory hierarchy organization, have to be selected. This work focuses on these formats.

2.3 Variable precision for high-precision scientific applications

Nowadays, modeling calculation is often performed on 32-bit (or more commonly on 64-bit) systems while employing the IEEE 754 Floating-Point (FP) standard. The reason behind this is mainly the availability of stable hardware and software easy to use and affordable.

However, according to Morrison [27], when complex non-linear dynamic systems have to be described or solved, "no level of numerical accuracy is always enough". The meaning of this expression is twofold. First, the IEEE 754 formats' precision may not be enough for some application domains, which brings room for improvements in both algorithms and VP FP representation formats. Second, there is a real demand and need from the scientific community for bigger memories, larger data-paths, or larger FP formats.

Thus, this section presents a non-exhaustive list of application domains that are considered meaningful for Variable Precision computing (Section 2.3.1). This discussion ends by showing to the reader that most of the high-precision scientific-applications concerns to linear systems solving (Section 2.3.2).

2.3.1 High precision scientific application domains

This work defines "high precision scientific applications" as the applications that require higher precision than the IEEE 754 64-bits FP format. A list of these kind of applications is made by Bailey in [28]. We choose the 64-bit boundary because it is the most common among modern high-end computer architectures suitable for scientific computing. This section presents a non-exhaustive list of applications that require hardware support for high, even variable, precision FP computing. It mainly focuses on the computation techniques used in *computational physic* and *computational chemistry* since they are excellent representatives of scientific computing.

Computational physic

Computational physics [29] covers topics such as climate model, weather forecast, and astrophysics, as well as industrial topics such as electromagnetics, fluid mechanics (computational fluid dynamics), and protein structure prediction. In most cases, the models are translated into systems of linear equations, or systems of Partial Differential Equations (PDEs), or systems of Ordinary Differential Equations (ODEs). Scientists extensively use numerical methods to solve these systems. Those methods approximate the functions of interest by polynomials, which are translated into systems of linear equations, computers can handle that. From [30], it is possible to generate three classes of numerical techniques:

- The *Finite-Difference* Methods (FDM [31]) are numerical methods for solving differential equations. FDMs are discretization techniques that solve differential equations by approximating them with finite difference equations. FDMs convert a linear (or non-linear) ODE system, or PDE system, into systems of linear (or non-linear) equations which can then be solved by linear algebra techniques.

- The *Finite-Element* Method (FEM) is a numerical method for solving problems of partial differential equations. The FEM formulation of the problem results in a system of algebraic equations by *approximating* the unknown function over the domain. To solve the problem, it subdivides a complex system into smaller parts called finite elements. The equations that model these finite elements are then assembled into larger systems of equations that model the initial problem.

- *Spectral* Methods (SM), like FDMs and FEMs, can be used to solve problems like ODEs and PDEs. Spectral methods [32] are a class of techniques used to numerically solve some

differential equations, potentially involving the fast Fourier transform. SMs and FEMs are similar. The main difference between them is that SMs use basis functions that are are global smooth functions (very commonly Fourier series, or orthogonal polynomials for non-periodic problems), while FEMs use basis functions that are usually polynomials of fixed degrees with null value outside of the neighborhood grid points. The advantage of spectral methods is that if the problem parameters are analytic functions, their convergence is exponential in the number of basis vectors. In the presence of discontinuities, their behaviour may become less satisfactory, c.f. the Gibbs phenomenon in Fourier analysis [33].

The approximation techniques in FDMs, FEMs, and SMs suffer from different error sources: they generate an intrinsic error, depending on the choice of basis functions and their discretization. All three methods go through a linear solving phase, which amplifies this functional error. Thus, VP FP computing can find a place in the linear solving phase by reducing this error.

Examples of VP applications are all the physics problems that translate the first ODEs set into a set of coupled first-order differential equations. An algebraic function vector, called "stepper", integrates these equations.

The integration method may vary according to the problem. The most common beings *Runge-Kutta*, *Richardson extrapolation*, or *multistep* [34], chapter 17. In all cases, even for stable systems, the round-off error is accumulated, but cannot be compensated as in iterative methods. This accumulation of errors may be related to the precision used in local steps [35]. Thus, the use of VP in the stepper allows us to trade the global computational error against an increase in computation complexity for a given number of iterations.

Computational chemistry

Computational chemistry is a branch of chemistry, based on computer simulations, widely used in the design of new drugs and materials. It uses theoretical chemistry methods, incorporated into efficient computer programs, to compute both static and dynamic configurations of molecules and solids. Taking quantum effects into account, it renders the problem more complicated than classical mechanics [36].

Besides methods based on simplifications of the fundamental atomic interaction laws, more modern solid-state applications rely on Density Functional Theory (DFT) to reach good accuracy with a comparatively low computational cost. Within the vast spectrum of numerical techniques involved in DFT algorithms, some explicitly require high precision, e.g., the computation of the accumulation of electron density [37]. The complexity and the large size of the problems require advanced numerical methods:

- Dense or sparse calculation of eigenvalues is a widespread problem, which appears in the computation of wave functions from a Hamiltonian.
- Spectral element methods solve plane wave formulations [30].
- Finite differences are also used, and they are generally preferred to finite elements for regular grids.
- Computing minimum potential energy involves non-linear, unconstrained optimization problems. Therefore, optimization techniques are also considered, such as steepest descent or conjugate gradient.

Since both computational chemistry and physics require solving large linear systems with high precision, Section 2.3.2 provides a general overview of the techniques and methods that can be used to solve these systems.

2.3.2 Solving large linear systems

Numerical methods' literature well covers linear algebra because of its very central role in many scientific applications. The most representative kernel is the linear solver, which computes the solution x of a linear system $Ax = b$. In physics and chemical modeling, the typical problem sizes range from 10^3 to 10^6, which has two direct implications:

- Matrix storage becomes a crucial issue, even though the matrices considered are sparse. Hence, methods working on the full matrix size should be avoided.
- Computing is memory bound. Memory bandwidth optimization is at least as crucial as arithmetic latency in the system's overall performance. System design takes both criteria into account by using composite performance models (e.g., roofline model [38]).

Classical solvers are based on direct methods that compute the solution in one pass. The LAPACK package [39] collects most of these methods and has been used in scientific applications. Direct methods are still used for linear systems of "small" size (e.g., in signal processing for embedded applications). For linear systems whose size is above 10^3 elements, iterative techniques are preferred since they require less computation and memory resources[1]. Variable precision is particularly suitable for such techniques: it reduces error computation while increasing only the memory footprint of the solution vector.

Difference between direct and iterative algorithms

Direct methods generally offer stable and predictable results for "small" matrix sizes (e.g., with up to m=10^3 elements). The memory cost, $\mathcal{O}(m^2)$, and processing costs, $\mathcal{O}(m^3)$, of direct methods, limit their scalability.

Algorithm 1 General scheme of an iterative solver that computes the solution x of Ax=b

1: $n = 0$
2: *compute* x_0
3: **while** $(n < max)$ **do**
4: $\quad r_n = Ax_n - b$
5: $\quad$ *solve* $A \times d_n = r_n$ ▷ It is a simpler solver, it returns only an approximation of d_n
6: $\quad x_{n+1} = x_n + d_n$
7: $\quad n++$

Algorithm 1 shows the generic pattern for *iterative methods*. The "solve" step (line 5) may return an approximation of d_n, used to compensate (for each iteration) the computational error on x_n. Iterative solvers differ from the approximation technique implemented in this step.

The drawback of iterative methods is that they do not always work out of the box. Different problems do require different iterative solver settings, depending on the nature of the equations and the data to solve. Moreover, iterations introduce their own round-off error term, which can contribute to the algorithm's non-convergence. For this reason, r_n should be computed with higher precision than one of the output results. Wilkinson and Moler cover this point.

Wilkinson-Moler bounds

According to Wilkinson [41] and Moler [22], it is possible to bound the computational error of iterative solvers while distinguishing different sources of round-off errors. They proposed to compute the intermediate values of d_n with another precision (ϵ_2) concerning one of the input

[1] However, for sparse matrices, MUMPS is a counter example [40].

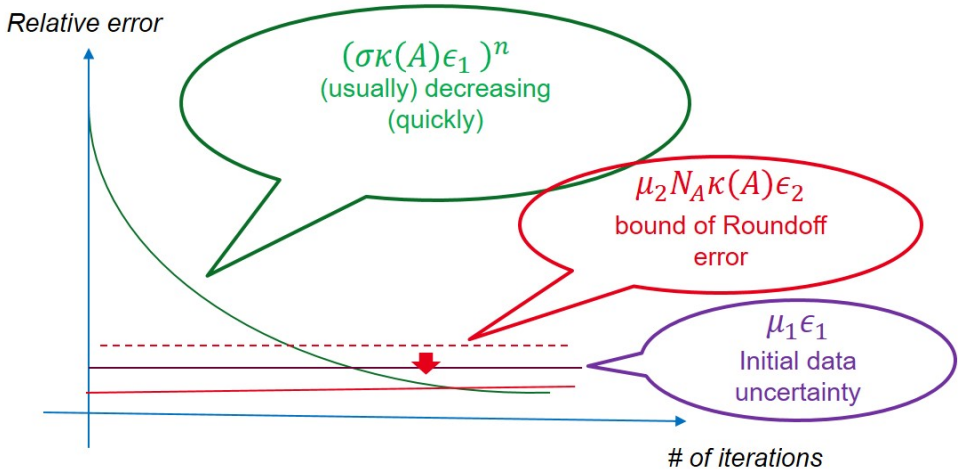


Figure 2.2: Decomposition of relative error for a general iterative solver

values (ϵ_1). Moler's equation (Equation 2.1) bounds the computational error of an iterative solver between its exact theoretical result x and the calculated result x_n, at the iteration n, which includes the round-off error.

$$\frac{\|x_n - x\|}{\|x\|} \leqslant (\kappa(A)\epsilon_1)^n + \mu_1\epsilon_1 + \mu_2 N_A \kappa(A)\epsilon_2 \tag{2.1}$$

In this bound, ϵ_1 is the initial precision of the input data, ϵ_2 is the computation precision inside the FPU, $\kappa(A)$ is the condition number of the input matrix A, n is the number of iterations of the iterative algorithm. The coefficients μ_1 and μ_2 depend on the input matrix, while N_A depends on the ϵ_2 value (for more details, please refer to [22]). Out of this complex notation, Figure 2.2 depicts how the *terms* of the relative error bound (Equation 2.1) behave along with the number of the algorithm iterations:

- The first term $(\kappa(A)\epsilon_1)^n$ is in $\mathcal{O}(\epsilon_1)^n$, becomes negligible when the number of iterations of the algorithm increases ($n \to \infty$).
- The second term $\mu_1\epsilon_1$ is in $\mathcal{O}(\epsilon_1)$, bounds the maximum precision achievable for the output result. There is no sense to have an output precision better than the starting one (ϵ_1). Thus, this term somehow limits the maximum number of iterations n to achieve the maximum output precision.
- The third term $\mu_2 N_A \kappa(A)\epsilon_2$ is in $\mathcal{O}(\epsilon_2)$, bounds the round-off error introduced during the computation. By tuning ϵ_2 with a "big enough" precision (smaller ϵ_2), the round-off error, magnified by the condition number $\kappa(A)$ of the input matrix A, can be compensated. *Variable Precision* is useful for this tuning.

2.4 Problems in using variable-precision computing

This section focuses on the problems that may arise when using Variable Precision (VP). First of all, in an algorithm, the set of variables for which the augmentation of their precision may improve the algorithm output results changes among applications. For example, considering ODE (Ordinary Differential Equations) solvers, the precision adjustment may concern only a subset of the problem variables, or only the intermediate values (as in linear algebra), or all of them (in chaotic problems). Since this work considers a dynamic adjustment of precision,

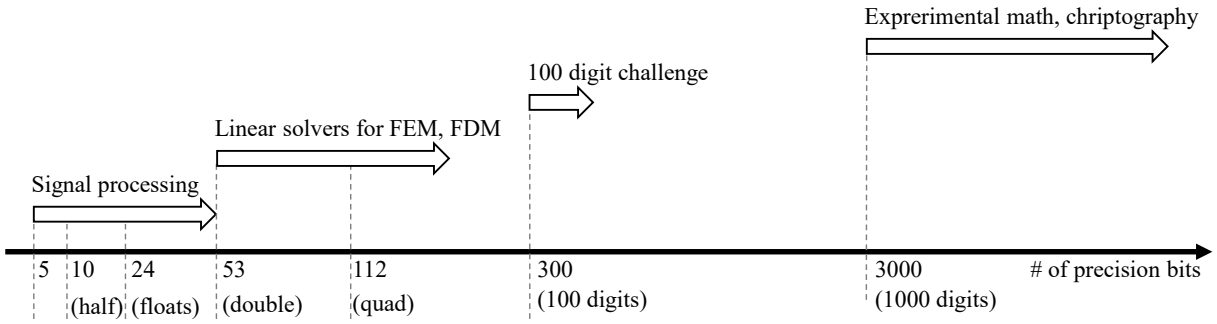


Figure 2.3: Required fractional fields sizes of floating-point variables involved for (a non-exhaustive list of) different applications

which means that the precision of variables changes during the process runtime, it must also define when the change occurs, and the decision criteria.

The question of dynamic criteria for changing precision is a complex mathematical issue, and the solutions depend highly on the applicative context. One may use a priori estimations of the condition number, or dynamic error evaluation when the condition number value is available and not too computationally demanding, or even interval arithmetic to qualify the result bounds. In the (extreme) case of chaotic systems, a usual way of judging the validity of the numerical result is to compare the results obtained by different computation methods applied to the same problem.

VP is meaningful for complex contexts, where developers and users demand a highly reusable programming model and are very reluctant to alter their legacy code. Section 4.8 addresses this other subject. Sections 2.4.1 and 2.4.2 describe the impact of importing VP for high and low precision applications.

2.4.1 Impact of high precision in computing systems

High precision scientific computation involves Floating-Point (FP) numbers with potentially a large mantissa bit-widths. This section discusses the problems that may arise during computations among high-precision values in real computing systems. To start this discussion, the first part of this section illustrates which are the orders of magnitude of the precisions involved for some application domains. The second and last part of this section discusses the problems that may arise in high-precision computing due to memory constraints.

Significand sizes for variables involved in scientific computing

Most representations of real numbers are radix-based. The following equation depicts a possible representation of a number x.

$$x = m \times \beta^e$$

The symbol m notes the significand (or mantissa) of x, β the radix, and e the exponent. Representing the number requires to store both significand fraction and exponent in memory. Hereafter, this study adopts the base 2 ($\beta = 2$) to represent numbers in memory. Thus, the e and m fields are represented in binary format.

Figure 2.3 depicts the number of fractional bits involved in the calculations depending on the target applications. Depending on the application, the bit size of the significand field of the FP numbers involved, its precision, may range from a few tenths of bits to several thousand. For example, some integrals in physics modeling typically require hundreds of bits above the

usual precision [42], [43]. Recently, a competition proposed ten relevant problems that have to be solved with 100-digit precision [44].

There are even more extreme cases in quantum physics where very high precision is required. For example, authors of [45] solve Schrödinger type equation eigenvalue problems numerically with very high precision (from thousands to a million decimals), to derive the wave-functions precisely enough to model the behavior of their system. However, this type of calculation is obtained with Taylor expansions and makes extensive use of specific large numbers of multiplication methods. Therefore, this case may be out of the scope of this study.

Memory constraints for variables involved in scientific computing

During high precision scientific computation, the programmer has to face physical hardware systems. Modern systems have 64 bits of internal data-path bit-width. Thus, the programmer must structure his code in such a way it can exploit the system underneath.

The problems that may arise, in modern computing systems, during high-precision scientific computing, are basically due to the main memory architecture. These problems are:

- The programmer must take care of all the techniques to write code to avoid stack overflow or page overflow. These issues may arise when passing massive parameters in the stack among functions, or when allocating massive data structures in memory.
- All high-precision variables need dynamic memory allocation on the heap. System-calls perform this allocation. However, they complicate the writing of the code and slows down the overall execution.
- State of the art code for high-precision scientific computing applications leverages software libraries [39], [46], and this brings two main problems. Taking MPFR [9] as an example, the first problem is that the system memory (including the cache) is polluted in a way that is difficult to predict with intermediate software variables instantiated by MPFR routines. The second problem is that, for programming reasons, the exponent and the mantissa of a given value are not contiguous in the main memory. This problem arises when using large memory structures (e.g., a matrix of high precision FP values): the number of cache misses due to the bouncing of the code to fetch exponent and mantissa values from memory may increase. Complex problems slow down system performance.
- Finally, to operate on variables that are not aligned on power-of-two boundaries, or on variables which have to be elaborated through software libraries, dedicated hardware support is required.

2.4.2 Impact of low precision in computing systems

Low (or reduced) precision computation involves short Floating-Point (FP) numbers. In this work, low-precision FP numbers are all the FP numbers with a bit-width smaller or equal than the system data-path one (e.g., 64 bits). In low-precision Variable-Precision (VP) computing, the main challenge is to deal with encoding variables on a length, which is not a power of two.

Reducing the bit-width of variables can bring three main challenges:

- As for high-precision variables, dealing with variables not aligned on power-of-two boundaries.
- Detecting *saturation* when the exponent range of variables is too small.
- Detecting the *non-convergence* of algorithms when the precision of the variables is too low.

The first point, as for high-precision variables, can be overpassed by providing dedicated hardware support. This hardware support can mask the potential additional latency needed to handle misaligned variables in memory.

The second point concerns the *saturation* problem. It may arise when, without modifying the algorithm, the length of variables, in particular their exponent dynamics, is reduced. By reducing the exponent dynamics, some operations inside the algorithm may saturate to infinity. This saturation is easy to detect through elemental software debugging techniques. The programmer could use saturation to evaluate the minimum exponent dynamics that each variable can have according to a specific input data set.

The last point concerns the *non-convergence* problem. It may appear when, without restructuring the algorithm, the precision of variables is reduced. Reducing the precision of variables, some intermediate variables of the algorithm may diverge (or converge) to wrong values. This effect is due to the accumulation of errors magnified by the reduction of the precision. The non-convergence is not always easy to detect since some iterative applications converge to a wrong solution (a canonical example is the "three-body problem"). The programmer can evaluate the minimum required precision that each variable can have according to a specific data set by using the non-convergence effect.

Chapter 3

State of the Art: what is known for variable precision

Figure 3.1 depicts the Floating-Point (FP) formats available in state of the art. Modern applications use the IEEE 754 FP formats ② defined in the IEEE 754 standard [1]. They are represented on predefined bit-lengths. According to Chapter 2, because of the discrete nature of fixed-size FP formats, FP applications are affected by rounding, cancellation, and absorption, computational errors. The accumulation of these errors can lead quickly to entirely inaccurate results.

As stated in Chapter 2, Variable-Precision (VP) computing can improve the accuracy of the final result of a problem to be solved by varying the precision of variables in memory. VP computing has been explored through several programming languages, software dedicated libraries, and hardware models.

This chapter depicts the state of the art of VP computing. Section 3.1 focuses on the VP formats available in state of the art. It firstly provides a global picture of the existing VP formats, and then it focuses on some of them in Sections 3.1.3, 3.1.4, and 3.1.1. These VP formats are compared with each other in Section 3.2.

To conclude this chapter, Section 3.3 covers the software and hardware VP implementations in state of the art. Section 3.3.1 introduces the existing VP software libraries. Then, Section 3.3.2 shows architecture examples of existing FP units. After that, it presents the two main VP FP hardware architectures: the Kulisch one, in Section 3.3.3, and the Schulte one, in Section 3.3.4.

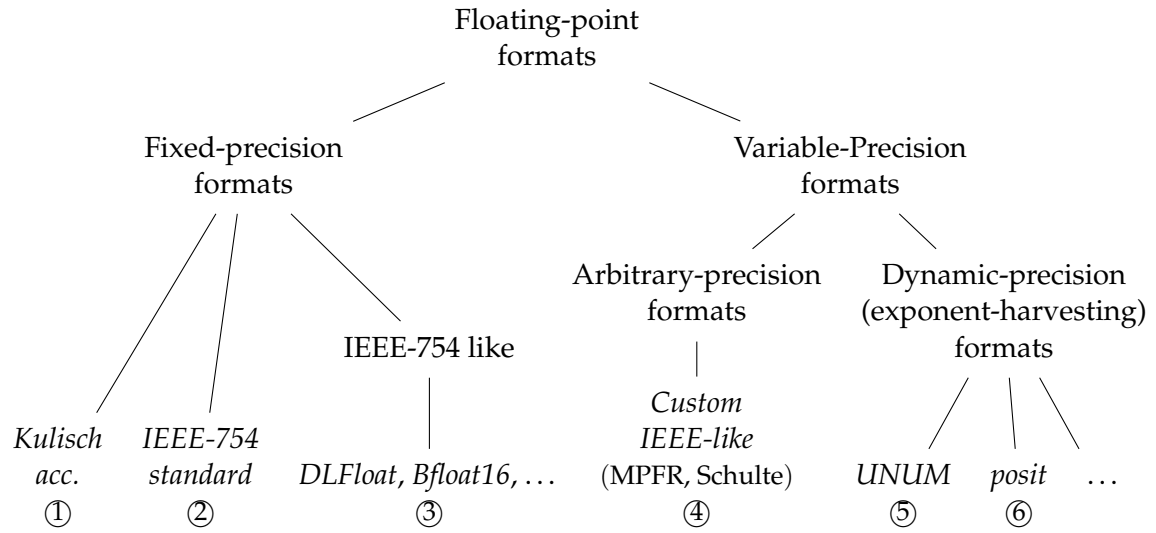


Figure 3.1: Subdivision of the existing floating-point formats in state of the art

3.1 Variable-precision FP formats and representations

This section focuses on the existing Floating-Point (FP) formats available in state of the art. Figure 3.1 divides into categories some of these formats. FP formats can be divided into two major families: *Fixed-precision* formats and *Variable-precision* formats. The first one contains all FP format with fixed size and fixed length for the exponent and fraction FP fields. The second one contains all the FP formats with variable size and variable length for the exponent and fraction fields.

In the fixed-precision family, FP numbers can be represented only with some allowed FP configurations (with predefined sizes for the exponent and fraction fields). This family is divided into three categories: the formats which implement FP operations employing fixed point accumulators ① (e.g., Kulisch [4]), the formats which follow the IEEE-754 standard [1] ②, and the formats which mime the IEEE-754 standard with different 8-bit length configurations for exponent and mantissa ③. Examples of the latter are the DLFloat [47] and Bfloat16 from Google [48]. With fixed-precision formats, the programmer has minimal flexibility in tuning the precision, and memory footprint, of the target application variables: it is possible only to change the utilized format. For most applications this is not a limit, and programmers can benefit of state of the art FP hardware acceleration. However, in the fixed-precision family, advanced programmers cannot customize the bit-length of FP numbers or the bit-length of the FP fields.

Thus, the scientific community investigated the second FP formats family: the Variable-Precision (VP) ones. The VP family hosts FP formats, suitable for VP computing, where the user can tune (according to some rules) the bit-length of the full number, or the bit-lengths of the exponent and fraction fields. According to Chapter 2, VP computing allows us to obtain better computation precision, and therefore, better result accuracy. VP formats are divided into two subcategories: *Arbitrary-precision* and *Dynamic-precision* formats.

Arbitrary-precision formats are all the three-fields (IEEE-754 like) VP FP formats, with customizable bit-length (at compile time) for the exponent and fraction fields ④. Within this family, the programmer can bound the mantissa and exponent bit-length of FP variables, depending on their maximum required precision and their maximum value in magnitude (like in MPFR [9] and Schulte [49]). Examples of VP FP formats belonging to this category are the ones defined in the Marto [50] HLS library.

Dynamic-precision VP-formats, ⑤ and ⑥ on figure, follows the idea of "trying to compact the FP exponent field, for exponent values near zero, to gain precision for the mantissa field". These formats leverage different types of exponent encodings to minimize its memory footprint near the value zero.

Section 3.1.1 covers in detail the custom IEEE-like. Section 3.1.2 illustrates the exponent-harvesting used in the UNUM and the posit VP FP formats. These two formats are covered in Sections 3.1.3 and 3.1.4, respectively. This work adopts the UNUM format as memory format since, at the beginning of this work (2016), only the UNUM format existed as VP FP format.

3.1.1 The custom IEEE-like formats

This section covers the arbitrary precision Floating-Point (FP) formats that mime the IEEE 754 standard and follow some of the rules defined in it. This category of formats contains all the FP formats based on a three-field encoding and fixed in size. With these formats, the programmer can tune the lengths of the exponent and fraction fields according to some rules dictated by the hardware constraints.

Figure 3.2 shows the custom IEEE-like format. It is similar to an IEEE-754 format, but unlike it, the user can tune the exponent size (es) and the fraction size (fs). The rules to express unique

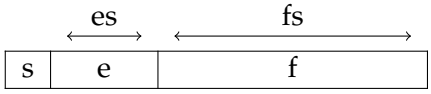


Figure 3.2: Custom IEEE 754-like floating-point format

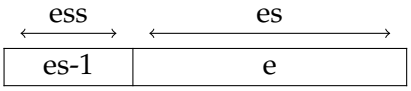
values (such as infinite or not-a-number), and the ones for rounding, can be different (or not) from the ones defined in the IEEE-754 standard.

Equation 3.1 shows a possible absolute value that may be encoded in a custom IEEE-like format (Figure 3.2). In this equation, some unique encodings for the exponent and fraction fields define the Not-a-Number (NaN) and infinity (∞) values. NaN is used to detect when operations output a non-existing numbers ($\infty \cdot 0 = NaN$). Infinity can be used to detect overflow in FP operations ($BigNumber_1 + BigNumber_2 = +\infty$, $\frac{BigNumber}{SmallNumber} = +\infty$).

$$x = \begin{cases} NaN & \text{if } e = e_{max} \\ \pm\infty & \text{if } e = e_{max} - 1 \ \& \ f = f_{max} \\ (-1)^s \cdot 2^{-(2^{es-1}-1)} \cdot (0 + \frac{f}{2^{fs}}) & \text{if } e = 0 \\ (-1)^s \cdot 2^{e-(2^{es-1}-1)} \cdot (1 + \frac{f}{2^{fs}}) & \text{if } e > 0 \end{cases} \tag{3.1}$$

3.1.2 Exponent-harvesting techniques

In conventional FP formats, the exponent field is encoded on a fixed amount of bits. With "*exponent-harvesting*" encodings, this work denotes all the exponent encodings which try to harvest some exponent bits for exponent values around zero. These bits can extend the mantissa field and increase the precision of the FP number. This work identified two main types of exponent-harvesting encodings.

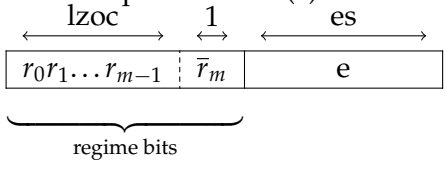


The first one is the exponent encoding used in by the *UNUM* [2] format ④. Here, the exponent is encoded in two distinguished fields: the first to encode the exponent size (es), and the second to encode the exponent (e). The es-1 field size (es size, ess) is constant, and the e field size depends on the value encoded in the es-1 field. Negative exponent values (exp) are encoded with a bias which changes according to the exponent size.

$$exp = \begin{cases} e - (2^{es-1} - 1), & \text{if } e \neq 0 \text{ (normal), where } 0 \leqslant e \leqslant (2^{es-1} - 1) \\ 1 - (2^{es-1} - 1), & \text{if } e = 0 \text{ (subnormal)} \end{cases}$$

With this representation, the value with minimal footprint is the value one with ess+1 bits (es-1=0 and e=1). The maximum representable exponent value depends on ess. The larger is ess, and the higher is the maximum representable exponent value. The es-1 field adds overhead in the exponent representation of ess bits, compared to a two's complement representation, or a biased one (like IEEE 754). This bit-overhead involves a loss on the mantissa representation of the same amount of bits for exponent values large in magnitude. This overhead is logarithmic on the maximum supported exponent value of the format.

The second type of exponent encoding is adopted in the *posit* [3] format ⑤. Here the exponent encoding is not biased and it is encoded on two separated fields: the regime field (r), which hosts the regime bits, and the exponent field (e).



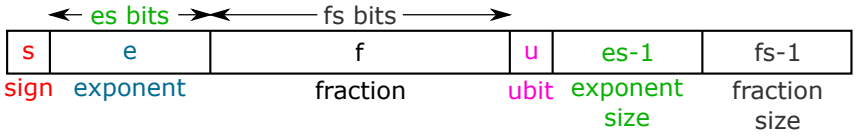


Figure 1.2: The UNUM floating-point format (repeated from page 2)

The exponent field size (es) is constant, while the size of the regime field changes among the encoded exponent values. The regime field is made of two parts. The most significant one ($r_0 r_1 \dots r_{m-1}$) is a variable-length sequence of either all zeros or all ones. The second part is a single bit ($\bar{r}_m$) used as a termination bit to delimit the first part. This bit is opposite to the ones unused in the first part. The length of the first part (which can be computed through a leading-zero-one-counter, lzoc) is used to encode the magnitude and the sign of the exponent (exp).

$$exp = \begin{cases} +((lzoc - 1) \cdot 2^{es}) + e, & \text{if } r_0 = 1 \text{ (positive values)} \\ -(lzoc \cdot 2^{es}) + e, & \text{if } r_0 = 0 \text{ (negative values)} \end{cases}$$

In this encoding, the exponent is not biased anymore. The *lzoc* and *e* encode the exponent value. The exponent values are divided into slots containing 2^{es} exponent values each. The *lzoc* value points the slot (or the exponent magnitude), and the *e* value points the exponent value within the slot (it works as an offset). If the most significant part of the regime field is zero, the exponent value is positive; else, it is zero.

With this representation, the exponent values with minimal bit-length are the $2 \cdot 2^{es}$ values nearby the exponent value zero. They are represented on es+2 bits (lzoc=1). The maximum representable value for the exponent depends on the available amount of bits to encode the FP number. If *n* bits are available, the maximum exponent number representable with this encoding is $((n-2) \cdot 2^{es}) + (2^{es} - 1)$. This exponent encoding can guarantee a more compact representation nearby zero compared to UNUM. The exponent footprint grows linearly with the exponent value.

3.1.3 Exponent-harvesting formats: the UNUM format

This section presents the UNUM format introduced by J.Gustafson in [2]. The UNUM Floating-Point (FP) format (Figure 1.2) is *self-descriptive* about the exponent size and the fraction size. It extends the three-field IEEE FP format adding three additional fields (the so-called *utag*):

- The *es-1* and *fs-1* fields. They express the bit size (minus one) of the exponent and fraction, respectively.
- The *ubit* (*u*ncertainty *bit*) field. It supports interval arithmetic in the format. It encodes the exactness (ubit=0) or not (ubit=1) of the FP number.

The UNUM format requires that the minimum length of each field is one bit. Please note that the *exponent size* and *fraction size* fields contain the absolute value of the bit size of the corresponding fields decremented by 1. Unlike the IEEE-754 standard, the sizes of the exponent and the fraction fields (*es* and *fs*) are dynamic. Their values can be customized based on the needs of the application and user. According to [2], those fields should be aligned on the right of the ubit. In this way, all the additional three fields can be stripped away easily if just the float part of the number is needed.

The size of the es-1 and fs-1 fields, *ess* (es size) and *fss* (fs size), define the maximum length of the UNUM fields. The couple *(ess,fss)* is named *UNUM Environment* (UE). In UNUM, the number of exponent bits ranges from 1 to 2^{ess}, while the number of fraction bits ranges from 1 to 2^{fss}. The minimum and the maximum footprint of a UNUM are $ess + fss + 4$ and $ess + fss + 2^{ess} + 2^{fss} + 2$, respectively.

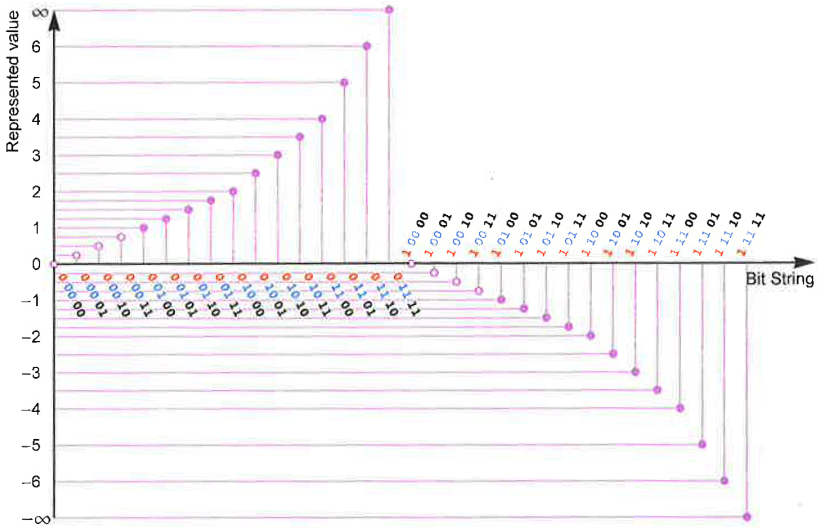


Figure 3.4: Exact values (ubit=0) of a UNUM number in a 5-bit encoding [2]

How to decode the UNUM format

The value encoded in the UNUM format, without considering the *ubit* field, is depicted in Equation 3.2.

$$x = (-1)^s 2^{e-(2^{es-1}-1)+(1-Boole[e>0])} \cdot \left(Boole[e > 0] + \frac{f}{2^{fs}} \right) \tag{3.2}$$

It expresses the value of a three-field FP number in function of the number of bit used to represent the exponent and fraction fields (this number is defined by the current UE). Equation 3.2 is not a continuous formula. It depends on the result of the $Boole[test]$ function (that returns 1 if the *test* is *true*, and 0 if the *test* is $false$). The fraction $\frac{f}{2^{fs}}$ is always between 0 included, and 1 not included ($f \leqslant 2^{fs}$-1). Depending on if $Boole[e > 0]$ function returns 1 or 0, the number is represented in *normal* or *subnormal* form respectively. Equation 3.3 rewrites the Equation 3.2. It shows the differences between normal ($e > 0$) and subnormal ($e = 0$) numbers representation.

$$x = \begin{cases} (-1)^s \cdot 2^{e-2^{es-1}+2} \cdot (\frac{f}{2^{fs}}) & \text{if } e = 0, \text{ subnormal} \\ (-1)^s \cdot 2^{e-2^{es-1}+1} \cdot (1 + \frac{f}{2^{fs}}) & \text{if } e > 0, \text{ normal} \end{cases} \tag{3.3}$$

How to encode infinity UNUM intrinsically supports both normal and subnormal numbers and embeds them into its format. In FP arithmetic, special FP values like $\pm\infty$ and NaN are essential. This section shows how UNUM encodes the $\pm\infty$ values are encoded in UNUM.

The UNUM format dedicates two values to represent *infinity* (using the sign to determine $+\infty$ or $-\infty$). The encodings chosen to represent these numbers are the two largest numbers (in modulo) that the UNUM format can represent: the ones with the es-1, fs-1, e, and f fields with all the bits set at 1. They are mapped as positive and negative infinity ($\pm\infty$), with the *ubit* set to 0. Equation 3.4 shows this concept with a mathematical representation.

$$x = (-1)^s \cdot \begin{cases} 2^{e-2^{es-1}+2} \cdot (\frac{f}{2^{fs}}) & \text{if } e = 0 \\ \infty & \text{if } e = e_{max} \ \& \ f = f_{max} \\ 2^{e-2^{es-1}+1} \cdot (1 + \frac{f}{2^{fs}}) & \text{otherwise} \end{cases} \tag{3.4}$$

Figure 3.4 plots the possible values of Equation 3.4 in a five-bit float example: one bit for the *sign*, two bits for the *exponent*, and two bits for the *mantissa*. Here, the range of values that a

UNUM number can reach (vertical axis) is more extensive than the one defined in an IEEE 754 number (with the same exponent and fraction field sizes). This is because in the UNUM format all the encodings used to map NaN values in the IEEE-754 standard ($NaN|e = e_{max}, \forall s, \forall f \neq 0$) are used to represent FP values. According to [2], this allows us to avoid choosing a larger UNUM environment, which enlarges the bit-lengths of the exponent and fraction fields of a UNUM number when it is not needed. This argument is valid for numbers defined on few bits (e.g., 5), but it is negligible for larger numbers (e.g., on 32 bits). Before presenting the encoding for not-a-number values, the usage of the *ubit* is presented.

The usage of the "ubit" field This section presents the usage of the *ubit* (*u*ncertainty *bit*) flag in the UNUM format. If it is 0, the UNUM number is an "exact value". It is possible to obtain an exact value if the previous operation did not need to round the fraction field. If it is 1, The UNUM number is a "not-exact value". It is possible to obtain a not-exact value if the previous operation did round the fraction field. The not-exact flag signals that, in a number, there are some non-zero bits after the end of the fraction, but there is no space to store them in the current UNUM format setting.

The ubit field, if set to 1, indicates a one-ULP worth of error or uncertainty of its value [2]. A *ULP* (Unit of the Least Precision) is the difference between exact values represented by bit string that differ by one Unit in the Last Place, or the last bit of the fraction. The ubit encodes this difference, which embeds interval arithmetic within the UNUM format.

In a chain of rounded FP operations, the error can exceed one ULP. A UNUM operator, to correctly bound the computational error in UNUM, enlarges the ULP by reducing fraction size (i.e., it decrements the fs-1 field). The increasing of the ULP width captures the interval explosion effect in standard UNUM computations, as in conventional interval arithmetic (Section 2.1.3).

Almost infinity, NaN and almost zero encodings FP values like $\pm\infty$ and NaN are essential to support FP arithmetic. Out of all the possible intervals encodable in a UNUM, two concepts deserve particular mention: *almost infinity* and *beyond the infinity*.

Almost infinity Setting the ubit of a UNUM transforms the FP number from a scalar value to an interval. One of them is the possibility to represent the concept of "almost infinity", meaning "a finite number that is too big to express". This number is pretty different from the mathematical definition of "infinity": *infinity* is the conceptual expression of such a "numberless" number (infinity is the quality or state of endlessness or having no limits in terms of time, space, or other quantity). This value corresponds to the UNUM number with the higher finite value that a UNUM format (with fixed es and fs) can express and with the ubit set. It is one ULP less than the representation of infinity. In the case of "almost infinity" the expressed interval is $(maxreal, +\infty)$ or $(-\infty, -maxreal)$, depending on how the sign bit is set. Notations $+\infty)$ and $(-\infty$ express "almost infinity" at the right and left bound of an interval, respectively. The alternative notation $\pm\infty \ldots$ can be used instead [2].

Another valuable property is that (with the ubit) it is possible to contain the overflow effect saturating the result of a UNUM operation to the *almost infinity* value. Unlike the IEEE-754 standard, all the next operations never receive a number equal to "exact infinity", that will propagate an error infinitely large in the next operations of a FP algorithm. Note that, the value "exact infinity" is used to encode the exact infinity when generated by mathematical constraints (like $\frac{number}{0}$ or $number \cdot \infty$).

Beyond the infinity: Not a Numbers In the IEEE-754 standard, it is possible to encode several Not-a-Number (NaN) values by varying the fractional bits' encoding. Unfortunately,

UNUM special value	Mathematical notation
± exact infinity	$\pm\infty \downarrow$
± almost infinity	$\pm\infty \ldots$, or $+\infty$), or $(-\infty$
quiet Not-a-Number	qNaN
signaling Not-a-Number	sNaN

Table 3.1: Mathematical notations for UNUM special values

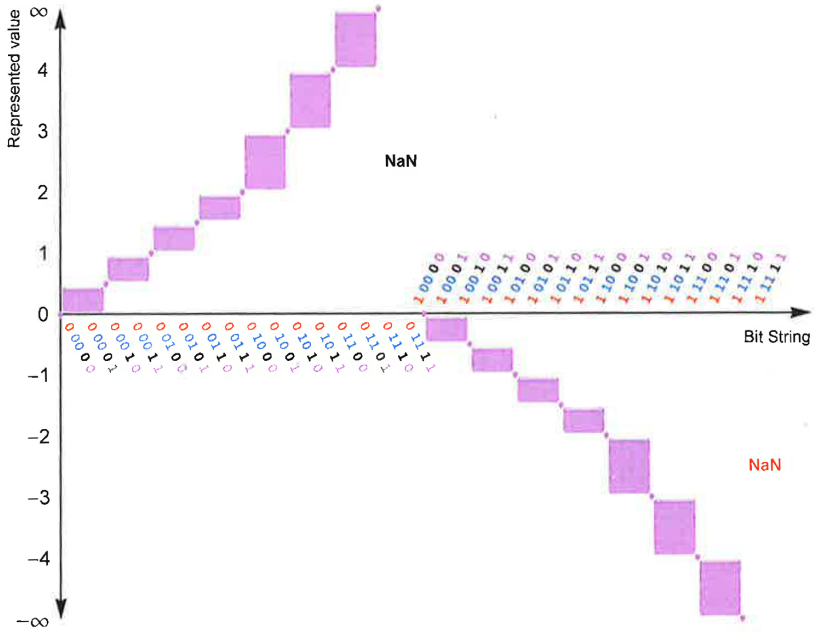


Figure 3.5: All the possible values of a UNUM number in a 5-bit encoding [2]

the common programming languages do not provide the support to exploit the information of what is the type of NaN obtained at the end of an arithmetic operation [2]. Due to this lack of information, the UNUM formats dedicates only two encodings to represent NaN values. The two available encodings for NaN values are *quiet* NaN, $qNaN$, and *signaling* NaN, $sNaN$.

In UNUM, the $qNaN$ value allows the computation to continue. However, any operation with a $qNaN$ as input provides a $qNaN$ as an output. The $sNaN$ value halts the computation and alerts the computer user. Unfortunately, [2] does not specify in which conditions sNaN appears.

According to the ubit definition, setting the ubit of a UNUM generates an interval between the exact number expressed in UNUM and the next exact one that differs by one ULP. Setting the ubit to the infinity encoding represents an interval between "infinity" and *"beyond the infinity"*. Since these values (positive and negative) are meaningless under a mathematical point of view, "beyond positive infinity" is used to encode $qNaN$, and "beyond negative infinity" is used to encode $sNaN$. Table 3.1 summarizes the concepts mentioned above with mathematical notations.

UNUM dynamics on the real axes Figure 3.5 shows, in a five-bit float example (one bit for the *sign* field, two bits for the *exponent* field, one bit for the *mantissa* field, and one bit for the ubit), how a FP number with the ubit behaves in the real axes. 'Dots' (●) and 'rectangles' (▪) represent exact and inexact numbers. Rectangles have rounded borders to remind that the ubit set defines an *open* interval between two exact numbers.

The ubound and the gbound formats

UNUM supports Interval Arithmetic (IA). However, by using only UNUM numbers, it is impossible to represent all the possible real intervals: e.g., a UNUM can not represent an interval containing zero. To support these intervals, UNUM supports the *ubound*.

The ubound The *ubound* is a pair of UNUMs representing a mathematical interval in the real axis. Exact UNUMs represent closed endpoints, and inexact UNUMs represent open endpoints. Some ubounds can be mapped to a single UNUM.

The ubound data structure extends IA to close, open, and half-open intervals while maintaining a well-defined behavior for all FP operators. The rounding rule upon ubound operations respects the inclusion property of IA (Section 2.1.3). If a qNaN (or sNaN) occurs in one ubound endpoint during computation, both ubound endpoints will be qNaN (or sNaN).

The gbound and the computation layers Gustafson distributes the FP computation on two domains: the *ulayer* and the *glayer* [2]. The *ulayer* (UNUM layer) belongs to the main memory and hosts UNUMs and ubounds. In this layer, it is impossible to implement operations between UNUMs and ubounds because their fields are not aligned. Moreover, extracting these fields without shift operations is unrealistic.

The *glayer* (general layer, or scratchpad layer) is the domain that contains the *scratchpad* for arithmetics. The FPU performs math operations among scratchpad entries. The scratchpad format equivalent of a ubound is the *gbound* (general bound), which the data structure used in the glayer for temporary calculations. It allows storing intervals with higher precision than in the UNUM environment. It embeds inside the values of the left and right endpoints of the interval.

Advantages and disadvantages of using the UNUM format

According to [2], UNUM has two main categories of advantages: its format and computing environment.

Format-related pros-cons The *utag* in the UNUM format adds an intrinsic bit overhead inside the number. Even if the *utag* adds this overhead, it can help to save memory space dynamically by reducing the overall length of the UNUM number. For example, to express the quad precision environment in UNUM, we need 4 bit for the exponent size (*es*), and 7 bit for the fraction size (*fs*) to cover the 15 exponent bits and 112 fraction bits that the IEEE 754 standard mandates. For that environment, $UE(ess = 4, fss = 7)$, the *utagsize* is equal to $1 + 4 + 7 = 12$. The minimum number of bit that we need is the one used to represent 0 is $utagsize + 3 = 15$. So this can bring a theoretical saving of up to $\frac{128}{15} \approx 8.5$ times of memory footprint for a single number. This saving up theoretically introduces the following advantages:

- Less memory footprint: the increase of the ratio *number of numbers* over the *number of bits*.
- Less power consumption: saved in the data bus between the memory and the computation unit.
- The running algorithm can be faster: in theory, fewer data words must be load-stored data from-to the system's main memory (RAM).

However, these advantages concern only a few FP values, those for which their exponent value is not too far from zero (near $2^0 = 1.0$). An IEEE-like exponent encoding is more convenient for all the other exponent values since it has a lower bit-length.

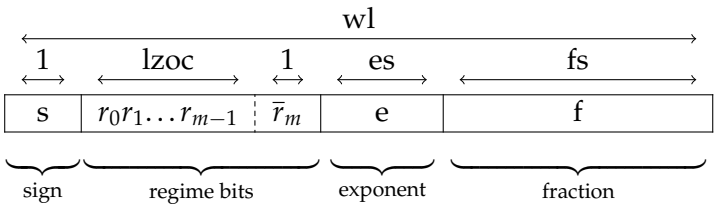


Figure 1.3: The posit floating-point format (repeated from page 2)

Computational-environment related pro-cons Interval arithmetic provides the inclusion property in the results of operations. It guarantees the safety of an algorithm. Please note that the programmer must take care of the "interval explosion", particularly in iterative applications, since it may mine the convergence of algorithms (Section 2.1.3).

3.1.4 Exponent-harvesting formats: the posit format

Figure 1.3 depicts the posit [3] Floating-Point (FP) format. Like UNUM, the goal of this format is to minimize the exponent encoding for small exponent values to increment the mantissa precision. Posit is different from UNUM in three parts: fixed *wl* size[1](Figure 1.3), no concept of subnormals, and no support for interval arithmetic.

Posit is made of four different fields: a sign *s*, an exponent *e*, a fraction *f*, and the *regime bits*. In the posit format, the exponent field *e* has fixed size, and this size, *es* (exponent size), is a parameter needed to decode the posit format (like ess and fss for UNUM, Section 3.1.3). The regime bits are made of two parts: a variable-length bit-string $r_0 r_1 \ldots r_{m-1}$, made of bits with the same value, and a special $\overline{r}_m$ bit used as a termination character of the previous bit string. Thus, it has negated value comparing to the value of the bits in the previous bit-string. The length of variable-length string, *lzoc*, obtained with a leading-zero-or-one-counter, is used to compact the exponent value encoding. The size of the fraction field *fs* is all the bits remaining by the other fields: wl-es-lzoc-2.

How to decode the posit format

Equation 3.5 shows the value encoded in the posit format.

$$x = \begin{cases} NaR \text{ (Not-a-Real)}, & \text{if } s = 1 \text{ and } r_0 \ldots \overline{r}_m{=}e{=}f{=}0 \\ 0, & \text{if } s = 0 \text{ and } r_0 \ldots \overline{r}_m{=}e{=}f{=}0 \\ (-1)^s \cdot 2^{+((lzoc-1)\cdot 2^{es})+e} \cdot (1 + \frac{f}{2^{fs}}), & \text{if } r_0 = 1 \text{ (positive exponent)} \\ (-1)^s \cdot 2^{-(lzoc\cdot 2^{es})+e} \cdot (1 + \frac{f}{2^{fs}}), & \text{if } r_0 = 0 \text{ (negative exponent)} \end{cases} \tag{3.5}$$

This equation is valid for all the possible *es* configurations of the posit number. Posit does not make a difference between positive infinity, negative infinity, signaling, and quiet not-a-number. One single encoding named Not-a-Real (NaR) represents all of them. Posit does not support subnormals. Thus zero is encoded as an exceptional value.

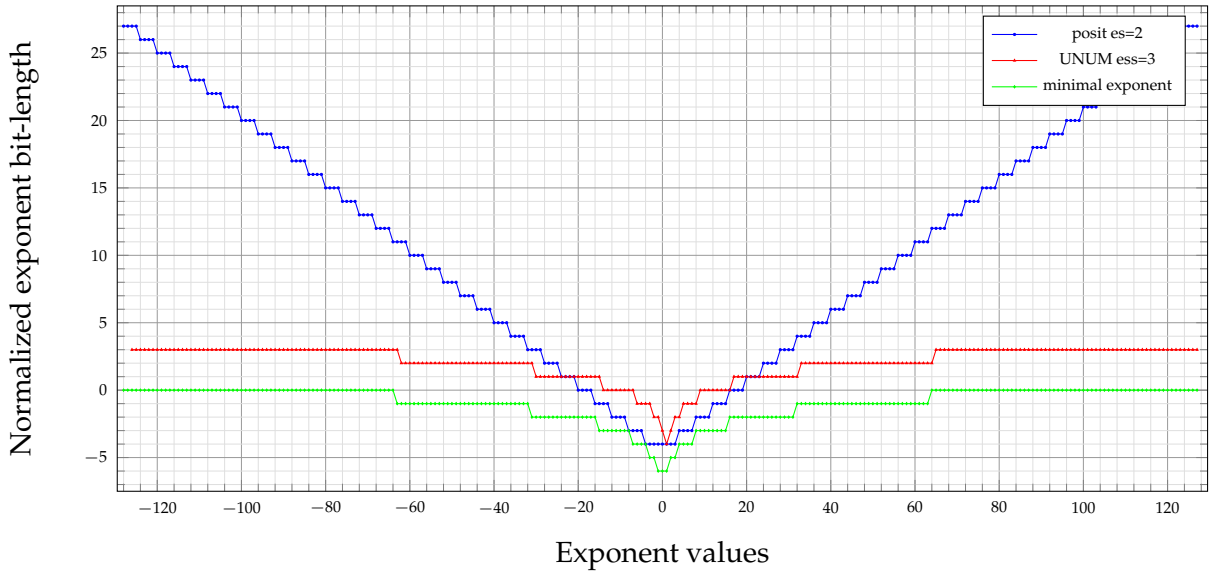


Figure 3.7: Exponent bit-length comparison in the posit and the UNUM formats, compared with a normalized two's complement 8-bit encoding. The green line corresponds to the minimum theoretical bit-length that an exponent may have

3.2 Comparison between the variable-precision formats

Figure 3.7 shows the comparison between the exponent encodings used for UNUM and posit formats, with a normalized two's complement 8-bit exponent encoding, representing the exponent encoding for a custom IEEE-like format. The horizontal axis shows all the possible exponent values which span between -128 and +127. The vertical axis represents the bit-overhead of the exponent encoding according to the 8-bit encoding baseline. All the points of the y-axis below zero are exponent encodings more compact than an 8-bit two's complement representation, while the points above zero are those requiring more bits.

The green line depicts the minimal exponent encoding, in two's complement, for each possible exponent value. This line represents the minimum theoretical footprint that an exponent encoding can have. The other two lines show the exponent overhead for the UNUM and posit encoding, with ess=3 and es=2, respectively. Since both exponent encodings use additional bits to encode the exponent field length (or where it finishes), reaching the green line is impossible. This plot depicts the UNUM and the posit formats, with the configurations ess=3 and es=2, because they share the same minimum encoding footprint.

The posit format performs better than UNUM on the "window of values" where there is a bit-length advantage for representing exponent values comparing to an 8-bit two's complement exponent encoding. On the contrary, for the exponent values outside this window (especially for small es values in posit), the posit format has an exponent encoding for which its bit-length is drastically worse than the one used in the UNUM format. For instance, applications, such as accumulators, that compute on (also few) FP numbers with exponent values large in magnitude, can propagate significant cancellation errors in the overall computation. Therefore the posit proponents make a Kulisch-like accumulator.

Posit differs from UNUM in the encoding of the exponent field. Instead of encoding the exponent length and the exponent value, it uses the regime bits to encode the actual used

[1] Having fixed size does not prevent posit to enter the "Variable-Precision" format category since multiple posit instances with arbitrary size can be allocated in memory. In Posit, the mantissa's precision is variable, although this precision is fixed by the exponent value only.

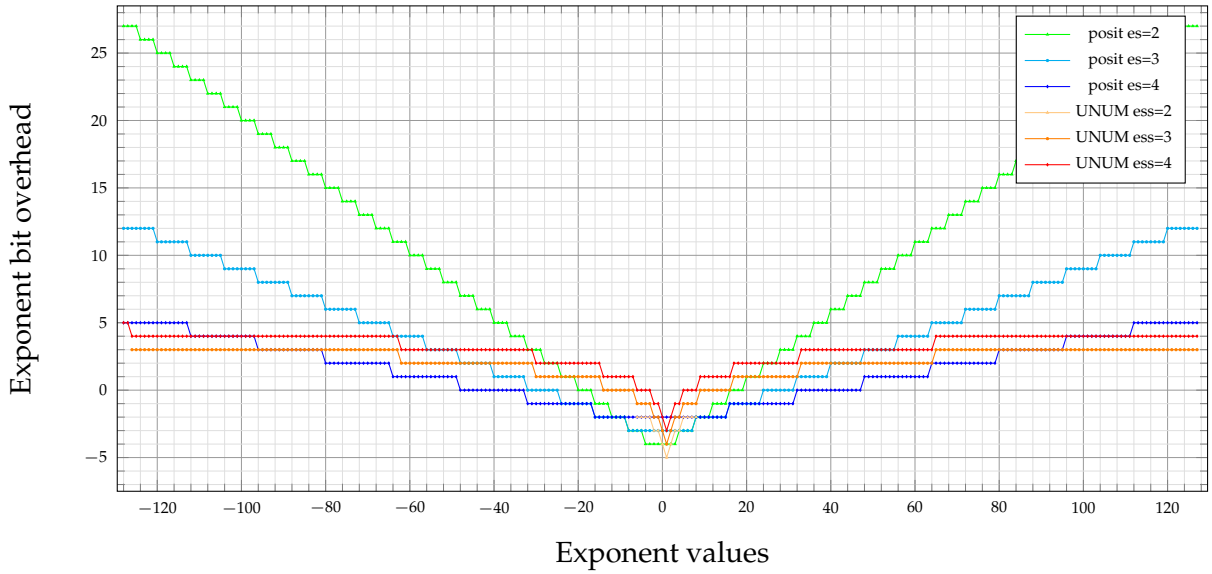


Figure 3.8: Comparison between UNUM and posit formats on different configurations

"window" of 2^{es} exponent values, and the exponent field to encode an offset among this window. This encoding allows us to avoid the redundant exponent representations of UNUM, minimizing the exponent field footprint for small exponent values.

As side effect, unlike UNUM, the posit exponent encoding grows linearly with the exponent value (UNUM grows logarithmically). The consequences that derive from this property are two. Firstly, posit suffers a substantial loss of precision for numbers having an exponent value large in magnitude. Secondly, posit has a reduced exponent dynamic compared to a conventional two's complement exponent encoding.

Figure 3.8 compares different UNUM and posit exponent encodings varying the ess and es global parameters, according to an 8-bit two's complement encoding. As it is possible to see, the posit exponent encoding grows linearly among the exponent values, while the UNUM one has a logarithmic behavior. The increase of the parameter es for posit, and that of the parameter ess for UNUM, increases the minimum exponent overhead that the encoding of the exponent can have. This bit-length increase is because both formats add bits to encode the exponent descriptor.

Unlike UNUM, posit does not have redundant exponent representation. Thanks to this, the posit format enlarges the "window" of values for which the exponent encoding is more compact than an 8-bit two's complement representation.

In conclusion, there is no exponent encoding that behaves better than the others. Each exponent encoding has a region of values for which it behaves better than another exponent encoding and some others for which it behaves worse than another exponent encoding. To better understand when one VP format performs better than another (and why), different benchmarks, with different data sets, must be run. As already said before, this work adopts only the UNUM format as memory format because, at the beginning of this work (2016), only the UNUM format existed as VP FP format.

3.3 Software and hardware implementations for variable-precision computing

At the state of the art, there are several multiple-precision software libraries (Chapter 2). For example, XBLAS [11] and MPFR [9] (based on GMP [15]) provide support for mixed-precision,

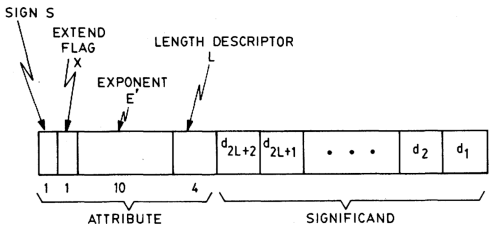


Figure 3.9: The decimal floating-point format used in the CADAC architecture [61]

and multiple- (and unbounded) precision computing, respectively. Section 3.3.1 provides further details about these libraries.

Nevertheless, these software solutions are computationally slow, and they are not suitable for high-speed applications. Such applications typically require high-accuracy in their results. Hence, the hardware implementation of Variable-Precision (VP) is crucial. In the literature, several VP architectures designs exist. They differ for the supported number format and the chosen to compute unit architecture.

The most common hardware architectures for Floating-Point (FP) computing are FP Units (FPU). Section 3.3.2 provides examples of this architecture. These architectures support multiple-precision FP computations by giving to the user support to use more than one FP format, among the ones defined in the IEEE 754 standard [1]. As stated in Chapter 2, a better result accuracy requires a subtle precision control in the FP formats.

To the best of the authors' knowledge, Kulisch made the first hardware proposal to significantly improve the result accuracy in FP applications [4], [51]. The idea of Kulisch is to guarantee an exact dot product by using a fixed-point accumulator large enough to contain the whole exponent dynamic of an FP number. Operations between FP numbers are performed by iterating on this long accumulator using micro-code instructions. This design paradigm has inspired several research groups (i.e., [52]–[60]). However, none of them (except posit[2][3]) implemented an architecture able to store data in memory with precision higher than conventional FP formats.

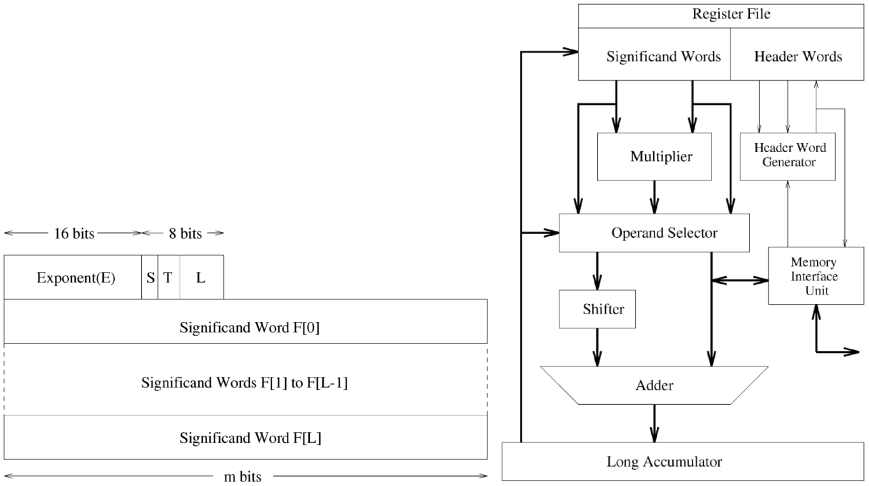
The main advantage of the Kulisch architecture is that it guarantees an error-less dot product operation thanks to its internal accumulator. As its main drawback, in applications where more than one accumulator is needed, it may be complex to store the internal accumulator's content to make room for the next accumulation. Moreover, the accumulator may have the size of several tens of thousands of bits.

In the same period as Kulisch, Cohen and Hull [61], [62] have presented an orthogonal approach for VP computing. Their idea is to have a variable-length (decimal-based) mantissa (Figure 3.9 shows an example). A descriptor encodes the length of the mantissa as the number of actually used digits. As for the Kulisch architecture, the hardware architecture is based on a microprogrammed machine, on a shared multiply-and-accumulate pipeline.

In 2000, Schulte [49] presented a similar approach to Cohen and Hull for VP computing. He presented a processor architecture that supports VP numbers internally and makes computation using interval arithmetic. The number format, Figure 3.10a, looks like conventional FP numbers where each interval endpoint is made of a sign, an exponent, and a mantissa.

Schulte presented a microprogrammed coprocessor, Figure 3.10b, based on an internal RAM as a register file, and a multiply-shift-add pipeline. Iterating on this pipeline, Schulte can solve any algorithm by approximating it as a polynomial. In the main memory, he stores only conventional 32 or 64 bits IEEE-754 FP numbers.

[2] The posit computing system requires an internal accumulator named "quire". The quire is similar to a Kulisch accumulator. It can be stored in memory as is. However, there are no specifications about how it can be stored in memory efficiently.



(a) Schulte floating-point format (b) Schulte micro-architecture

Figure 3.10: The floating-point format and architecture presented by Schulte [49]

In the next decade, other research groups proposed improvements to the Schulte's work. The authors of these works present optimizations on computation performances [63] or computation capability, for instance, by supporting trigonometric functions [64], [65]. All these architectures are microprogrammed, but none of them specify the data structure to store intermediate numbers in the main memory.

Another orthogonal direction is the one taken by the NVIDIA group, synthesized by Ozbilen [66]. Here, the concept of VP computing does not focus on augmenting the numbers' precision, but on downsizing for applications that do not require high-precision calculations. In [66], like in [67] and [68], each FP operation embeds the operands (in input) precisions and the result (in output) precision that an operator should have. This information brings the possibility to exploit the tradeoff between accuracy and execution time as well as the tradeoff between required power and result precision.

All identified architectures can be mapped into two leading hardware-architecture families: the ones of Kulisch and Schulte. These two families are described in detail in Sections 3.3.3 and 3.3.4, respectively. Before focusing on these details, Sections 3.3.1 and 3.3.2 provide an overview of some multiple-precision software libraries and existing FPU architectures.

3.3.1 Multiple-precision software libraries

Several applications use multiple-precision software libraries. For example, some Matlab functions use the LAPACK library [39]. Other examples are XBLAS [11], which extend LAPACK, and MPACK [69], based on multi-precision BLAS and LAPACK (MBLAS and MLAPACK). XBLAS routines are a multi-precision version of basic linear algebra routines (BLAS) [70]. Similarly, the (arbitrary precision) MPFR [9] library is based on mixed-precision, leveraging the GMP multiple-precision software library [15].

These libraries are used to emulate Floating-Point (FP) operations when the available hardware does not support the required mantissa precision. They store FP values in a dedicated FP format, where the exponent and the mantissa are two distinct fields. The lengths of their fields are customizable according to some rules. For example, MPFR allows the implementation of FP algorithms with any precision, especially for precisions higher than the system data-path bit-width (e.g., 64 bits). In MPFR, GMP performs the computation on floating-point mantissa (e.g., additions, subtractions, and shifts), iterating on the existing hardware operators.

There are different ways to implement multi-precision FP computing. One way is FP expansions. Priest developed the FP expansions arithmetic [71]. Shewchuk did the same in a slightly different way [72]. The main idea is to chain several variables that use the FP formats available in the hardware, to express FP numbers with higher precision. This work discards this technique since it puts in question the whole FPU architecture, and it sees this technique as a software work around to augment data precision.

3.3.2 Existing floating-point units in the state of the art

The Floating-Point Unit (FPU) is part of most modern computing systems, and a vast number of applications use it to perform computation. Before the publication of the IEEE 754 standard [1], the FPUs proposed in the market supported custom Floating-Point (FP) formats in memory. Nowadays, the IEEE 754 standard standardizes the behaviors of FPUs[3]. Many embedded processors do not have hardware support for FP operations. In this case, software libraries (e.g., soft-float support in the GCC compiler) perform FP operations. These libraries are not detailed in this work since the focus is on hardware architectures. This section presents the three main architectures options available in the state of the art of FPUs.

In a computational system, FPUs can be either tightly coupled or loosely coupled. Like in the Ariane core [73], the *tight coupling* option integrates the FPU directly in the main core pipeline. Thanks to hardware optimizations techniques (like *forward* or *out-of-order* execution), this architecture guarantees a reactive interaction between the FPU and the main core. Without taking any precaution, the main drawback of this solution is that the FPU can be part of the system's critical path, slowing down the system's working frequency.

The *loose coupling* option embeds the FPU as a coprocessor of the main core. The coprocessor integration allows us to decouple the main core and the FPU in terms of latency and critical path. Its main drawback is the higher clock-cycle latency due to the synchronization mechanism between the main core and the FPU.

The third and last option is a *hybrid* approach, like in Rocketchip [7], which integrates the FPU in the system as a unit parallel to the main processor, but with a dedicated bus interface. Thanks to this dedicated bus interface, the overhead latency is reduced due to the synchronization mechanism.

FPUs generally compute on a FP Register File (RF). The FP RF hosts FP numbers. They are loaded either from the main memory or through the RF of the main core. The FP RF can be updated either by the Load-and-Store-Unit (LSU) or by the FP operators.

Its physical location depends on the type of hardware architecture used. Architectures with the FPU embedded in the main processor pipeline, have the FP RF in the decode stage of the pipeline [73], and it is accessed in both decode and write-back stages. Other architectures with the FPU external to the main core (e.g., Rocketchip [7]) have the FP RF within the FPU.

The size of this RF and the size of the RF entries are architecture-dependent. To optimize the addressing of the RF, the number of RF entries is a power of two. This number can span from 8 (in the Intel 8087) to 128 (in the Intel Itanium 9700). The principle is simple: the more FP registers are available, the less the data stored back in memory. However, a large RF can be critical in terms of area-footprint and access time.

Depending on the architecture, the bit-width of each FP RF entry may vary. The minimum bit-size of a RF entry is the bit-size of the largest FP format supported in memory (e.g., double, 64 bits). Additional bits may be added to the RF entries to make explicit some information such as the hidden bit, or some flags (e.g., NaN and inf).

The bit-length of the RF entries can be enlarged to extend the FP mantissa precision and reduce the computational error of the results. According to Higham [23], having higher precision in the RF entries can help to achieve better result precision in computation. Like the FPU

[3] FPUs architectures differ in their internal organization, which is also fairly standard.

of the Intel Itanium 9700, modern FPUs support 80 bits mantissa in the FP RF entries. However, supporting larger mantissa in the FP RF entails double rounding: the first one during FP operations and the second during store operations.

The RF of the FPU hosts FP numbers loaded from memory by the FPU LSU. FPUs usually support more than one FP format. Thus they support mixed-precision computing (Section 2.1.1). Please note that also the mixed-precision support implies to perform double rounding.

In FPUs, FP operations are handled internally by the hardware FP operators. Since the FP basic operators require separated data paths (as explained in Muller [8]), they treat exponent and the mantissa fields separately. Some hardware architectures have been proposed to implement FP operators. The simplest one is based on a shared multiply-shift-add pipeline and implements FP operators through micro-code [49].

Others FPUs support in hardware as many FP operators as needed [7]. The implementation of all the FP operators is not mandatory. The minimum set of FP operators needed in hardware is made of the addition/subtraction and the comparison. The other arithmetic operations, such as multiplication and division, can be implemented using the existing hardware[4]. At the cost of having more area for the silicon chip, implementing other FP operators can speedup applications execution. The final speedup depends on the application. It must be evaluated by analyzing the probability of using all the FP operators simultaneously, for instance, by analyzing the frequency of FP operations.

Data bit-width in modern computing systems

The power-of-two lengths of the Floating-Point (FP) formats, supported in modern FP units (FPU), are due to the memory hierarchy organization. FPUs are fed with data from the main memory, which comes through one or more layers of cache memories. Cache memories are designed to reduce the latency of load and store operation. They leverage different memory hardware technologies and different cache protocols algorithms.

Cache memories are organized in lines. Each line hosts a power-of-two number of words with a bit-width equal to the one of the system data bus (e.g., 64 bits). Smaller power-of-two words can be exchanged within the cache. The shorter bit-length usable in the cache memory is the byte (8 bits). This choice simplifies the address decoding inside cache memories.

To keep data aligned and compact, the IEEE 754 standard defined FP formats with power-of-two bit-lengths (on 16, 32, 64, and 128 bits). In this way, the FP data can be manipulated easily during memory operations within the cache, avoiding data fragmentation among cache lines.

3.3.3 Kulisch: eliminate the round-off error using long accumulators

Previous sections provide an introduction to the existing hardware architecture for mixed-precision Floating-Point (FP) computing. This section and the next one (Section 3.3.4) present the two leading hardware architecture families for Variable-Precision (VP) FP computing. This section focuses on the Kulisch [4], [51] based architectures.

The Kulisch architecture leverages a fixed-point accumulator, large enough to host the whole exponent dynamic of an FP number. It can improve the result accuracy of FP applications since the accumulator can map all FP values without precision loss. If the dynamic range is larger than the dynamic of FP products, this accumulator architecture guarantees an error-less dot product operation.

All the FP operations are done within this accumulator through exact fixed-point basic operations. The Kulisch architecture loads and stores data in memory using a memory format

[4]This is also true for additions in some software libraries (e.g., softfloat)

that follows the IEEE-754 standard (i.e., with a 64-bit double). Thus, the only computational error source of this architecture comes from the rounding of the data when they have to be converted in the FP memory format.

Kulisch proposes to implement operations between FP numbers by iterating on sub-portions of this long accumulator (i.e., on 64 bit). A micro-programmed machine on a shared multiply-and-accumulate pipeline performs these operations. In this way, the hardware complexity is kept under control, and the system realization in hardware is feasible. However, it is not possible to generate the outputs within a single clock cycle.

The Kulisch accumulator can be several thousand bits wide. For example, to support the 64-bit IEEE 754 FP format, the accumulator must be 2099 bits (1023 to represent the positive exponent values, 1022 to represent the negative ones, 53 to represent the mantissa, and 1 for the hidden bit). This bit-size must be doubled to support exact dot products. Doubling the bit-size raises *two* main drawbacks. For the *first* one, hardware operations on this long accumulator may take several clock cycles. This latency entails operations with high latency and low throughput. Moreover, the Kulisch accumulator does not support single-cycle accumulations.

For the *second* drawback, for applications where more than one accumulator is needed, it may be complex to make room for the next accumulation by storing the internal accumulator's content in memory. Every time the content of the accumulator is stored in memory, it is necessary to either use the IEEE-754 format losing all the precision gained inside the accumulator or store its content at the expense of wasting several KB of memory. Even worse, during a context switch, the operative system needs to save the machine status in memory to make room for the next application. If it is not possible to store the accumulator's content in memory as is, the programmer has no control over the computation precision because the accumulator content may be cast into an IEEE-754 format.

3.3.4 Schulte: contain the round-off error extending the mantissa precision

The Schulte architecture [49] takes an orthogonal approach from the Kulisch one. Contrary to the Kulisch architecture that encodes its internal registers in fixed point, the ones in the Schulte architecture have a Floating-Point (FP) representation. Instead of zeroing the effect of rounding errors in FP operations as in Kulisch's architecture, Schulte's approach reduces their effect by enlarging the mantissa precision during FP computation. This architecture works on a Register File (RF) that can host FP numbers with Variable-Precision (VP) mantissa. All the RF entries are self-descriptive VP FP numbers: the mantissa precision is encoded within the number.

This architecture can be seen as a hardware version of MPFR [9], with the MPFR code written in micro-code. The main differences in the number format comparing to Kulisch are two. For the first one, the exponent supports large values and is represented in two's complement (on 16 bits). For the second one, the mantissa is divided on more words of p bit each. Each number is variable-length: there is one field (L) which encodes the number of words that the mantissa has; There are some flags that encodes distinguished values (NaN, ∞, 0, $normalized$, etc.).

Schulte presents a microprogrammed coprocessor based on an internal RAM as a register file. Like Kulisch, the Schulte architecture is based on a micro-programmed machine on a shared fixed point multiply-shift-accumulate pipeline. FP numbers are stored in memory with standard 32 or 64 bits FP formats. All the algorithms are implemented on this pipeline through polynomial approximations. An additional feature of this architecture is that it supports interval computation, which can track the running algorithm's computational error.

This architecture is potentially faster than the Kulisch one because the number of 64-bit words involved in the operation is lower than the ones in the Kulisch accumulator. Moreover, more operation on FP entries can run in parallel in the Schulte architecture. This parallelism is not possible in the Kulisch architecture since only one accumulator is available at a

time. Neither the Kulisch architecture nor the Schulte one allows storing the content of their high-precision internal registers or accumulators. However, if they would support the store in memory of these values, the Schulte architecture would have less impact on memory footprint (compared to the Kulisch accumulator) while having a small computational error compared to a conventional IEEE-754 FP unit. This lower impact is proven, for instance, during context switch if the size of all the FP registers in the Schulte architecture is less than the size of the Kulisch fixed point accumulator.

Chapter 4

System architecture for the variable-precision computing unit

This work proposes a general-purpose hardware architecture for VP FP computing supporting different programmable variable-length FP data types, in memory, and inside the FP unit. Introducing VP in existing FP computing architectures opens questions such as:

1. What does it mean to support VP in hardware?
2. Which FP format should be used to encode VP FP numbers in memory?
3. Which hardware architecture should be used to accelerate VP FP computation?
4. How should the VP be exposed to the programmer (C level)?

Section 4.1 answers the first question. It introduces all the issues that may arise in realizing a hardware unit that fully supports VP computing.

Sections 4.2, 4.3, and 4.4 address the second question. The VP FP format used in memory is a revisited version of the UNUM type I format [2]. Section 4.2 introduces the constraints of a modern system and explains the modifications performed on the UNUM format to make it usable in a real system. Section 4.3 proposes a different encoding for the UNUM format. However, supporting VP FP formats in hardware may cause several issues. Section 4.4 lists all these issues.

Sections 4.5, 4.6, and 4.7 covers the third question. Section 4.5 proposes a hardware architecture to accelerate VP FP applications. This architecture is implemented as a RISC-V coprocessor. Section 4.6 shows how it is possible to program the VP coprocessor, providing also some code examples. Section 4.7 presents how to program the coprocessor at the assembly level through an expansion of the RISC-V ISA (Instruction Set Architecture).

This chapter concludes with Section 4.8, which provides the answer to the fourth and last question, by specifying a new programmable FP data type named `vpfloat`. This data type can be customized at compile time on several aspects (e.g., precision, memory footprint, and memory format). As the main feature, the programmer can set its memory footprint with byte-granularity at compile time. Thus, the programmer has more accurate control over the precision of variables, while the compiler, which controls the variables' memory-footprint, can efficiently allocate them in the main memory. This work is aware that a hardware unit can support VP FP variables up to a given size. Above this limit, the compiler can provide VP support by implementing VP computations through software libraries (i.e., MPFR) and alternating the computation between hardware and software depending on the operations' input or output precision.

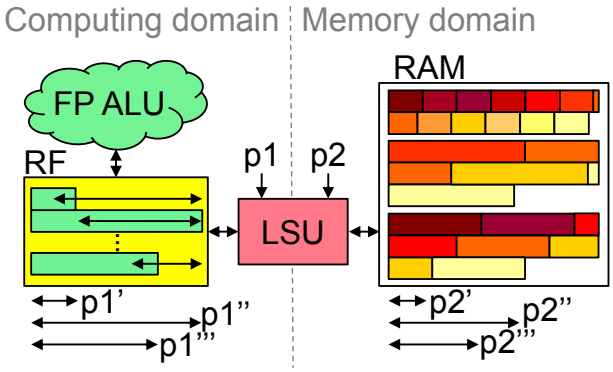


Figure 4.1: A variable-precision architecture supporting two different floating-point formats with different precision. One to represent numbers in main memory, and one to represent numbers in the internal register file

4.1 Supporting variable precision in hardware

The capability to bound the relative error in an expression (or code segment) is the essential requirement for the forward and backward error analysis. As seen in Section 2.3.2, it is useful to have several precisions at our disposal.

4.1.1 Supporting different variable precision formats

However, there are some constraints in embedding all possible precisions in a hardware VP FPU. This work identified two primary constraints. The first constraint is related to the impossibility of computing data inside a FPU with a precision that can change with a bit granularity. This constraint is due to the hardware blocks, used to implement the hardware operators of the FPU, cannot change their bit-length.

A solution to this problem is to fix the size of these hardware blocks to a given size (e.g., 64 bits) and constraining the precision granularity to be modular on the same size. Operations on mantissa are performed by dividing the mantissa on chunks of the same size (e.g., 64 bits). All the hardware operators, including the rounding unit, will behave according to the predefined mantissa precision.

The second constraint is related to the memory subsystem. The memory subsystem is designed to support (almost) all the possible data that a programmer may store. However, data must pass through a data cache (conventionally) organized with a minimum p-bit granularity. In most of the caches, for example, the ones developed in a RISC-V environment, work with a minimum granularity of p=8 bits. This minimum granularity dictates the bit-width granularity that VP data can have in memory.

The solution to these constraints presented in this work is to support two different VP FP formats. One format used to represent data in memory (e.g., RAM), and one used to represent data inside the computing unit (its Register File, RF). Figure 4.1 depicts the high-level overview of our VP FP architecture. The two formats have two independent precisions, p1 and p2. By controlling these precisions, the user can have more precise control over the final results' computational error.

Having these two levels of precision opens the possibility to have tunable latency and precision during computation while having tunable latency and precision during memory operations. It is possible to exploit these tradeoffs and make exact and high-speed computations inside the VP FPU while having compact data in memory. This tradeoff can potentially increase the computation's precision while minimizing the latency in moving the data in the memory system.

In a computing system, the Load and Store Unit (LSU) is the unit in charge of moving data between the computing and memory domains. Its internal operations are transparent to the programmer. For this reason, we decided to use a dedicated LSU to cross the two VP domains. This dedicated LSU handles the conversion between the two formats. The architecture can work with different precisions while supporting fused operations internally. Thanks to these two precisions, in addition to better control of the result computational error, it is possible to exploit the accuracy-latency tradeoff during computation.

As for drawbacks, the coexistence of two formats with different precisions involves an additional rounding operation during store operations in memory. Moreover, the support of data with lengths that are not a power of two implies to adapt the LSU to support misaligned operations in memory. A misaligned operation in memory is when the user wants to access (load or store) data with a length not aligned on the address of the data in memory. In other words, misaligned operations in memory may require to access data on different cache lines. It is possible to see this problem by looking at Figure 4.1, assuming that the RAM in the figure corresponds to the SRAM inside an L1 cache, and all the red lines are several cache lines hosting VP FP data.

According to Section 3.1, VP FP formats, such as UNUM [2] or posit [3], are designed to be as compact as possible to minimize the footprint of variables in memory. They dynamically compress the exponent field while maximizing the fraction field bit-length. However, they are not hardware friendly since the positions and the lengths of the exponent and mantissa FP fields are data-dependent. Section 4.4.1 addresses this point.

4.1.2 Controlling the interval explosion effect in a UNUM computing unit

This work adopts the UNUM format as a memory format. UNUM supports Interval Arithmetic natively (IA, please refer to Section 2.1.3 for more details). Thus we propose a hardware unit that supports interval operations internally. However, algorithms implemented with IA may be affected by the interval explosion effect. To overcome this side effect, Gustafson proposed to use the `guess` operator that computes the midpoint of an interval [2][1].

This work supports the `guess` operator with the result rounded to nearest (half away from zero, RTN). The first usage of this operator can be to implement RTN by alternating interval and guess operations. This usage gets rid of intervals while overcoming the interval explosion issue in IA. Using guess to support RTN is excessive since the rounding could be implemented directly during operations. For this reason, we decided to support both interval and scalar computing. The scalar computing supports several rounding modes, including RTN.

Alternatively, the `guess` operator can be used to compute the midpoint of a midpoint-radius interval representation. The midpoint-radius interval representation can be more convenient than the one with two distinguished internal endpoints since it can have less memory footprint. This because, instead of representing two distinct interval endpoints with high precision, it is possible to have a vary precise midpoint with a less precise radius representation. A more compact encoding to represent intervals in memory allows us to increase the average precisions of the intervals stored in memory and postpone the interval explosion side effect. For this reason, to compute the radius of a midpoint-radius interval representation, we also decided to support the radius operator providing the width of an interval[2].

In conclusion, the hardware unit presented in this work supports both interval and scalar computing natively. In addition to the basic IA operators, during interval computing, the hardware unit supports the `guess` and `radius` operators to convert intervals in midpoint-radius

[1] Please note that the use of guess in the middle of an algorithm based on IA destroys the main IA objective of bounding the computational error within an interval.

[2] Please note that this operator can also be used to observe the accuracy (and computational error) on the ongoing computation since it evaluates the width of a variable.

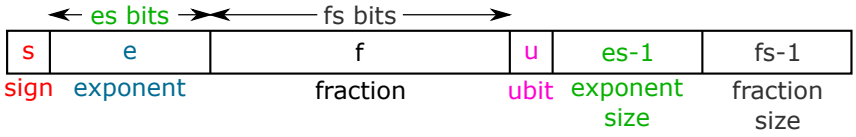


Figure 1.2: The UNUM variable-precision floating-point format (repeated from page 2)

representation. For the scalar computing, the unit, and its hardware operators, support several rounding modes for scalar computation, including RTN.

4.2 Optimizing variable-precision floating-point formats for memory

The computing precision in scientific applications may be decided at run time, for example, depending on the algorithm's computational error (Chapter 2). Thus, this work targets a Floating-Point (FP) format where the user can tune its size (in particular the sizes of the exponent and fraction fields). The support of this format leads to create a VP FP-Unit (FPU) that supports VP FP numbers stored in the main memory with a given representation. This section focuses on choosing the VP FP format supported in the main memory for the target VP FPU.

This work adopts the UNUM type I format for storing numbers in memory (Figure 1.2, Section 3.1.3) because the challenge of studying a FP format that is conceived to be a "better" replacement for the standard IEEE 754 FP formats (claim of Gustafson [2]) was appealing. This thesis aims to verify if this claim is right and to evaluate the cost of supporting this memory format in hardware.

This format supports IA natively using the "uncertainty" bit field *u*. The ubound data structure (tuple of UNUMs, Section 3.1.3) represents intervals. The lengths of the *es-1* and *fs-1* fields, *ess* and *fss*, define the maximum length of a UNUM. This tuple (ess, fss) is called UNUM Environment (UE).

The UNUM type I format has two peculiarities that are not hardware-friendly. The first one (Section 4.2.1) is related to the original order sequence of the UNUM fields. The second one (Section 4.2.2) is related to the data organization in memory of UNUM array elements. Section 4.3 proposes a solution for these two points. It proposes a modified version of the UNUM format, providing a solution that guarantees affine random access to VP FP arrays (in this case, based on the UNUM format modified).

4.2.1 Mapping of the UNUM fields in memory

The current FP formats available in state of the art are designed to fit within the data-path bit-width of the architecture (32 or 64 bits). In this way, the data can be moved between memory and FPU with a single operation with the processor Load and Store Unit (LSU). The VP FP UNUM format may not have a power-of-two bit-length. This not-power-of-two bit-length brings two memory-related problems: the number address may be misaligned on its size, and the number may have a bit-length spanning multiple memory addresses.

For a UNUM-ubound that spans multiple addresses in a little-endian memory system (like in the RISC-V architecture), it is crucial to have the descriptor fields present in the lower addresses. The conventional UNUM field organization (Figure 1.2, [2]) is not ideal since the position of the fixed-length fields (u, es-1 and fs-1) changes according to the bit-width of the variable-length ones.

Figure 4.3 depicts the field organization chosen for the UNUM (Figure 4.3a) and ubound (Figure 4.3b) formats. The fields are re-organized in such a way that, for both the interval endpoints (left and right), the variable-length fields (exponents and fractions) are placed in

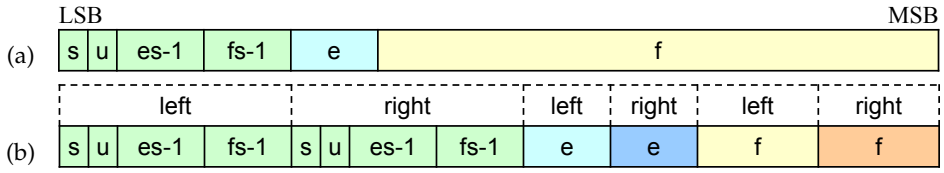
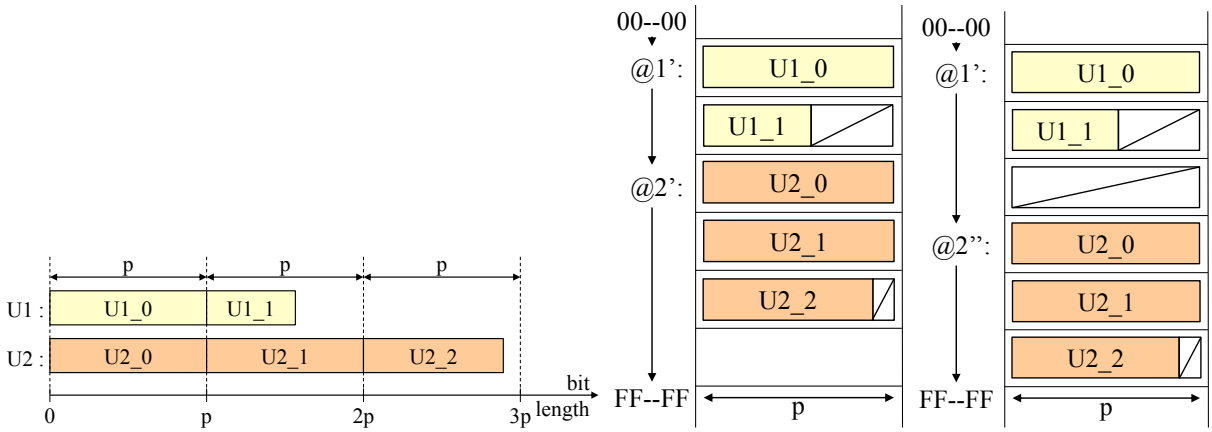


Figure 4.3: Binary encodings for the UNUM and the ubound formats in memory



(a) Two VP FP numbers organized in p-bit chunks. p is the access granularity of the memory subsystem, for instance, p=8 in RISC-V

(b) Two VP FP numbers stored in memory with the compacted addressing mode

(c) Two VP FP numbers stored in memory with the slot-aligned addressing mode

Figure 4.4: Alternative memory addressing modes (4.4b, 4.4c) for variable-length numbers 4.4a

memory after the fields that are fixed-length (u-tags). For ubounds (Figure 4.3b), the affinity with the left or right interval endpoint is noted above the fields.

With this field organization, during load operations in memory, the position and the lengths of the UNUM fields can be decoded from the first loaded bytes: it is possible to know the sizes of the variable-length fields within the first data chunk read from memory. Even when reading intervals from memory, ubounds, Figure 4.3b, the extraction of the fixed length and exponent fields can be done within one clock cycle with standard hardware.

This work provides hardware support for VP FP variables with up to 16 and 512 bits of exponent and mantissa to explore scientific computing algorithms. In UNUM, this choice corresponds to fix the maximum supported UNUM Environment (UE) to (ess=4, fss=9).

4.2.2 Main memory organization of UNUM array elements

To preserve the existing software infrastructure (e.g., compilers), the address of a random element of an array, in main memory, must be known a priori (e.g., at compile-time). This property is mandatory to guarantee affine access to random positions in arrays. Without this, the access time of data in memory is no longer deterministic, and the schedule of operations becomes not predictable in real-time systems.

Therefore, all the existing memory data types and all the elements of arrays have fixed memory-footprint, and array elements have all the same bit-length. The UNUM format violates this rule since, as depicted in Figure 4.4a, each number may have a different bit-lengths depending on the lengths of the exponent and the mantissa fields. In this study, every VP FP number in memory is seen as a chain of *chunks* of p-bits. The memory subsystem fixes the p granularity (Section 4.1). Since this work targets a RISC-V environment, this work fixes $p = 8$ bits.

According to the UNUM format (Section 3.1.3), a UNUM may have different lengths depending on the precision needed by the number itself. This variable-length feature may lead the reader to have, in memory, *compact arrays* of UNUM with different bit-length elements, where each element changes its precision depending on its needs. Software applications using compact UNUM arrays are affected by memory fragmentation and by arrays re-allocation. Those effects appear if a UNUM-array element changes its bit-length after an operation. Its bit-length can increase (e.g., after multiplications where the output bit width tends to double) or (more rarely) decrease (e.g., after subtractions between values with same exponent values, or after a division when the divisor is a multiple of the dividend). In both cases, array re-allocation requires expensive system calls.

Figures 4.4b and 4.4c depict the two addressing modes used in this work to address UNUMs, designed to avoid the system calls usage. The first one (Figure 4.4b) supports *compact arrays* in memory. It concatenates array elements in order to minimize the memory slots occupation. Memory accesses are done sequentially since each array element (@1') points to the address of the next one (@2'). This addressing does not support in-place algorithms nor random access to arrays, as access to array elements is strictly sequential due to their size, which can vary. This addressing mode can be used for long-term storage of arrays in memory, without losing precision on array elements.

The second addressing mode, Figure 4.4c, is more similar to the addressing mode used for conventional data types and supports random access on data arrays. It *aligns* the array elements on slots of a fixed size, which are multiple p bits. In this way, unlike the previous addressing mode, the array elements addresses (@1' and @2") are data-independent and can be computed at compile time. With the UNUM format, both addressing modes may waste memory bits (empty boxes ▨ in Figure 4.4), but compact arrays waste less.

This work focuses on iterative algorithms that require high-precision in memory and the computing unit. These algorithms generally tend to increase the length of their variables. Therefore, the space initially unused in Figures 4.4b or 4.4c is eventually used. This work exposes these two addressing modes to the user but, in practice, the one pointed in Figure 4.4c is more convenient to use in VP FP programs. This addressing mode is used in the experiments illustrated in Chapter 6.

To maximize the accuracy of a UNUM variable, the user must dedicate a specific number of p-bit slots large enough to fit the variable's maximum bit width. If the maximum precision possible to reach with a given set of a VP variable is not required, this work allows the programmer to customize the bit-lengths of variables in memory by specifying the number of p-bit slots in which they are to be stored. Section 4.3 provides further details on how to apply this feature to the UNUM format.

4.3 The Bounded Memory Format: fitting UNUM in a modern memory hierarchy

This section introduces an adaptation of the UNUM format to make it usable in memory. This adaptation is named "*Bounded Memory Format*" (BMF). BMF is a VP FP format that extends a self-descriptive VP FP format. The main objective of BMF (unlike UNUM type I) is to provide the user with a way to preserve affine memory accesses, thus affine access to random positions in VP FP arrays. To do that, BMF keeps the data in main memory aligned on boundaries (or frontiers), specified by the programmer (e.g., BMF allows us to define an array of VP FP numbers of 7 bytes each). These boundaries are guaranteed by *re-rounding* the data to fit within the specified boundary width (the rounding mode can be programmed). They are defined by the programmer defining the MMB environment variable (Definition 1).

Definition 1 *The "Maximum Memory Budget", or MMB, defines the maximum memory footprint that a VP FP number (e.g., UNUM type I) may have. The MMB value can be expressed in any form (e.g., bit-length, byte-length, ...). MMB can be renamed as "Maximum Byte Budget" (MBB), or "Maximum Bits Budget" (MbB). It expresses a specific number of bytes, or a specific number of bits, respectively.*

Since most common memory subsystems are organized in bytes, this work uses MBB to align data in memory. Under the software point of view, every VP FP variable is associated with a MBB value. This value encodes the maximum size that the variable may have during load operations from the main memory. In particular, during store operations, the MBB value encodes the bit-length of the memory slot in which the stored number in the main memory must fit.

If the bit-length of the number to be stored exceeds the MBB value, it must be re-rounded to fit in MBB. Unique values (e.g., $\pm\infty$, NaN , ...) can be encoded as unreachable values (with respect to MBB) of the length-descriptor fields of the VP FP number (in UNUM *es-1* and *fs-1*). The introduction of the BMF concept can be seen from three points of view:

- This work introduces BMF as a new VP FP format that extends an existing self-descriptive VP FP format. This format aligns VP FP data in memory according to boundaries defined by the user.
- This work introduces a VP FP computing system that automatically casts VP FP numbers in the main memory according to boundaries defined by the user using the BMF VP FP extension.
- This work opens the possibility to work with two different precisions for memory (MBB) and internal operations (ideally higher than memory).

4.3.1 BMF, a memory-friendly version of the UNUM type I format

According to Definition 1, the MBB value defines the maximum memory footprint that a UNUM data can have in the main memory. From the user perspective, MBB is a global parameter that must be defined before memory operations (load or store) that involves the system's main memory. Once that the value of MBB is set at the system level (e.g., by setting the corresponding system status register), all the posterior memory operations use the MBB value to properly support the BMF format (until when the MBB value is modified again).

BMF changes the rules to decode or encode a UNUM number from (or to) main memory (Section 3.1.3). When the bit-length of a UNUM (dictated by its fields and its UNUM Environment) is larger than MBB, it has to be *re-rounded* to a value with a bit-length compatible with MBB. The next sections describe the BMF encoding of UNUMs. In particular, they describe:

- How to apply BMF to the UNUM type I format when the MBB boundary is *larger* than the maximum UNUM memory-footprint (Section 4.3.2).
- How to apply BMF to the UNUM type I format when the MBB boundary is *smaller* than the maximum UNUM memory-footprint (Section 4.3.3).
- How is the BMF encoding for the UNUM type I format in detail (Section 4.3.4).

4.3.2 BMF encodings when MBB is larger than the UNUM bit-length

Figure 4.5 depicts the UNUM format's BMF format when the MBB boundary is larger or equal than the maximum memory footprint of a number encoded with the UNUM type I format, on its UNUM Environment (UE [2]). The horizontal axis represents the bit-length of a UNUM

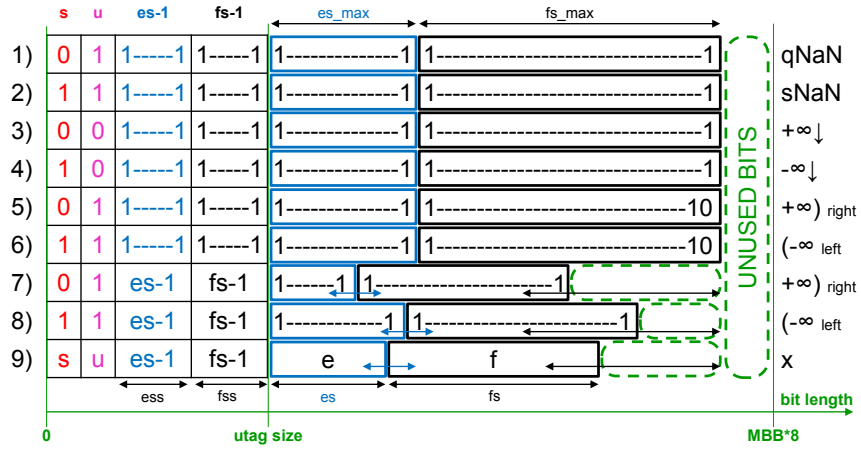


Figure 4.5: BMF binary encodings for the UNUM format, when MBB is larger or equal than the maximum bit-length of a UNUM in the (ess, fss) UNUM Environment. Unique values follow the notation of Table 3.1

stored in memory. The vertical axis represents the possible combinations of representation that a UNUM may have (from 1 to 9) in the (ess, fss) UE.

This section focuses on the case where the MBB value is greater or equal than the maximum bit-length that a UNUM in the (ess, fss) UE may have. The UE is equal for all the representation cases (1-9, in the vertical axis), and the bit-length that a UNUM has to respect is MBB*8 bits (since MBB expresses a specific number of bytes).

As introduced in Section 3.1.3, the UNUM format can encode distinguished values ($qNaN$, $sNaN$, $\pm\infty \downarrow$ or $\pm\infty \ldots$, lines 1-8) or normal ones (line 9). Lines 1-6 encode the distinguished values with the *maximum length* of a UNUM. The distinguished values in lines 5-8 represent (positive and negative) "almost infinity", they can be encoded either with the maximum bit-length (lines 5-6) or not (lines 7-8).

More in details, lines 5-6 show the representations of negative and positive almost-infinity ($\pm\infty \ldots$, for the left and right interval endpoints respectively) in case of not-exact endpoints ($u = 1$) that have maximum size endpoint values. While lines 7-8 show the representations of negative and positive almost-infinity in case of not-exact endpoints that have their sizes shorter than the maximum length that a UNUM can reach in its UE. Line 9 shows the UNUM encoding for a standard value.

In Figure 4.5, the horizontal double-arrows and the green-dashed boxes have special meanings. The horizontal double arrows highlight that the exponent and fraction fields' bit-lengths can change according to the number descriptor (the utag). The green dashed boxes highlight the unused bits in the data slot allowed by the MBB (MBB*8 bits). Their value can be any value. In other words, those boxes highlight the data alignment in memory according to the MBB value.

4.3.3 BMF encodings when MBB is smaller than the UNUM bit-length

Description: For MBB values smaller than the maximum UNUM bit-length, some UNUM encoding may be cast, or rounded, to fit in the MBB boundary. Figure 4.6 depicts all the scenarios (lines 1-9) in this situation. The axes, the horizontal double-arrows, and the green-dashed boxes have the same notation of Section 4.3.2. The orange circles highlight the MBB boundary when it is tighter than the starting UNUM bit-length. Here, the usage of different UNUM Environments (UE) highlight different truncation scenarios.

Scenarios 1, 2, and 3 share the same UE (ess', fss'). Scenario 1 depicts when MBB is larger than the encoded UNUM, it covers what presented in Section 4.3.2. Scenarios 2 and 3 depict the

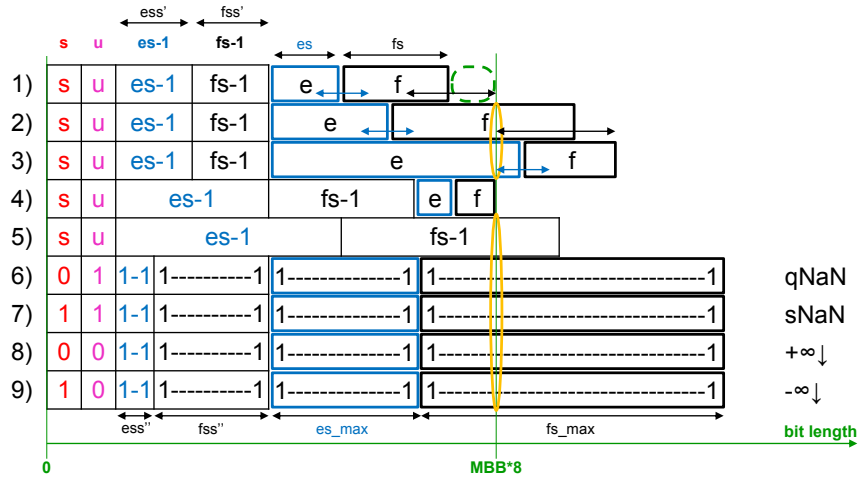


Figure 4.6: Truncation effects caused by the MBB value in different UNUM settings

cases when the MBB boundary truncates the fraction field, or both the exponent and fraction fields. Both cases may happen during computation.

Scenarios from 4 to 9 (Figure 4.6) depicts the truncation effects of MBB for different UE settings. The scenarios 4 and 5 show the consequences when the MBB value is "too small" according to the actual UE. Scenario 4 depicts the case where MBB is the minimum value that allows us to encode a UNUM. The bits available to encode the exponent and the fraction are only 2. Thus, only the NaNs and $\pm\infty$ values can be encoded. Scenario 5 depicts the case where MBB is too small: there are not enough bits to represent even the smallest UNUM under a given UE. Since this scenario is seen as a user programming error, BMF does not support this case.

Scenarios 6, 7, 8 and 9 depict the cases where, for some UEs (e.g., (ess'', fss'')), the MBB value does not allow to encode $qNaN$, $sNaN$ and $\pm\infty$ $\downarrow$. Section 4.3.4 describes alternative encodings for these values.

Solutions: Figure 4.6 depicts the scenarios when the MBB value leads to cast or round the UNUM to be stored (lines 2-3 and lines 5-9). In scenario 5, the UNUM is simply not encodable in BMF: there is no meaning to store a VP FP number with no exponent or mantissa. In scenario 2, the mantissa has to be rounded (toward $\pm\infty$) to the maximum footprint allowed by the MBB value. In scenario 3, there are not enough bits to express the magnitude (exponent) of the UNUM to be stored. Its value is saturated to "almost infinity" (positive or negative according to its sign).

BMF provides a not-ambiguous binary encoding to support all the UNUM distinguished values (lines 6-9). These encodings use unreachable utag encodings in the UNUM number. Section 4.3.4 provides further details about the BMF encodings.

4.3.4 Putting all together: The BMF encoding for the UNUM format

This section presents the BMF encoding rules designed to support all possible combinations of MBB and UNUM Environments (UE). As basic rules, under a given UE, it is not allowed to set MBB values smaller than a minimum size: the size of the utag plus 3 bits.

Figure 4.7 shows the BMF encodings for the UNUM format for all the possible UE and MBB values. The notations used are the same as Figure 4.5 and 4.6. Encodings 1-9a depict the cases where the maximum UNUM bit-length, in the (ess', fss') UE, does not exceed the MBB boundary (the ones of Section 4.3.2).

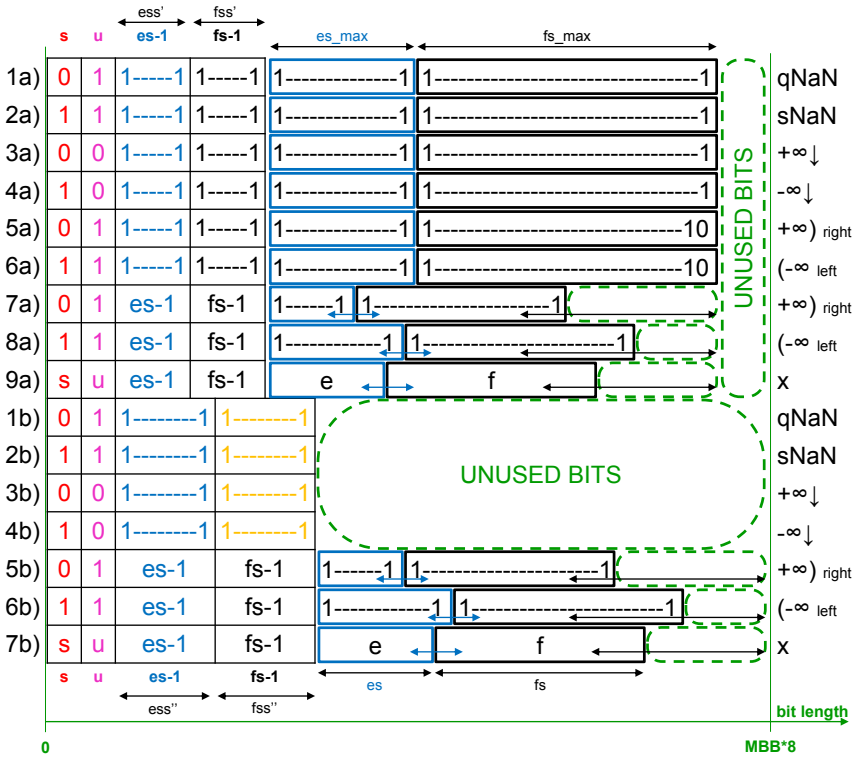


Figure 4.7: BMF binary encoding varying the MBB and the UE parameters

Encodings 1-7b depict the cases where the maximum UNUM bit-length, in the (ess'', fss'') UE, exceeds the MBB boundary (the ones of Section 4.3.3). For this reason, they have a UE lager than the UE of cases 1-9a. Encodings 1-4b represent the UNUM distinguished values.

The encodings 1-4b exploit the following property: if the maximum memory-footprint of the UE (ess'', fss'') is larger than the one imposed by the MBB value, the fields `es-1` and `fs-1` can not be equal to $2^{ess''}$-1 and $2^{fss''}$-1 (all ones), respectively. With this property, it is possible to encode the UNUM distinguished values ($qNaN$, $sNaN$ and $\pm\infty$ $\downarrow$) setting both the `es-1` and `fs-1` fields to their maximum value. A dedicated configuration of the sign and ubit fields differentiates each distinguished value.

Like encodings 7a and 8a, encodings 5b and 6b are used to map the "almost infinity" values. One of the two is selected depending on the sign of the cast or rounded value. Encodings 5-7b behave like the encoding 7-9a and reflect the same reading rule of the scenario 7-9 of Figure 4.5 in Section 4.3.2.

Figure 4.7 depicts the BMF encodings supporting the UNUM type I format for all the supported UEs and for all the legal MBB values (that respects the rules described in these last sections). This configuration uses as few bits as possible to encode $qNaN$, $sNaN$, and $\pm\infty$ $\downarrow$ for each MBB value. Moreover, it encodes $\pm\infty$ ... with the 5b and 6b encodings (always of Figure 4.7). Algorithms 2 and 3 show the algorithms in pseudocode that the BMF load and store hardware block should implement. The *leftNotRight* input parameter is used to perform correct rounding, and to handle distinguished values, during interval computation.

An alternative BMF encoding

The BMF encodings presented until now are the ones used in the final demonstrator. However, other encodings may be used. For example, Figure 4.8 shows an alternative of the BMF encoding presented in Section 4.3.4. The only difference compared to Figure 4.7 is in encodings

Algorithm 2 BMF load

```
1:  function BMFLD(address, UE, MBB, leftNotRight)
2:    unum U                                              ▷ loaded UNUM
3:    utag ← LDUTAG(MEM[address], UE)                     ▷ get the utag from main memory
4:    utagWidth ← UE.ess + UE.fss +2                      ▷ get the utag memory footprint
5:    eWidth ← utag.es                                    ▷ get the exponent memory footprint
6:    fWidth ← utag.fs                                    ▷ get the fraction memory footprint
7:    if MAXUSIZE(UE) < MBBSIZE(MBB) then   ▷ UNUM max size can fit in the current MBB
8:      e ← LDEXP(MEM[address+utagWidth], eWidth)         ▷ load the exponent
9:      f ← LDFRAC(MEM[address+utagWidth+eWidth], fWidth)  ▷ load the fraction
10:     switch [utag, e, f, leftNotRight] do              ▷ check loaded data
11:       case ISSNAN(utag, e, f)
12:         U ← sNaN
13:       case ISQNAN(utag, e, f)
14:         U ← qNaN
15:       case ISEXACTINF(utag, e, f)
16:         U ← ±∞ ↓
17:       case ISNEXACTINF(utag, e, f, leftNotRight)
18:         U ← ±∞ ...
19:       case others
20:         U ← EXTRACTFIELDSUNUM(utag, e, f, leftNotRight)   ▷ according to Figure 4.7
21:   else                                  ▷ UNUM max size can not fit in the current MBB
22:     switch [utag] do                                  ▷ check loaded utag
23:       case ISSNAN(utag)
24:         U ← sNaN
25:       case ISQNAN(utag)
26:         U ← qNaN
27:       case ISEXACTINF(utag)
28:         U ← ±∞ ↓
29:       case others
30:         e ← LDEXP(MEM[address+utagWidth], eWidth)       ▷ load the exponent
31:         f ← LDFRAC(MEM[address+utagWidth+eWidth], fWidth)  ▷ load the fraction
32:         switch [utag, e, f, leftNotRight] do            ▷ check loaded data
33:           case ISNEXACTINF(utag, e, f, leftNotRight)
34:             U ← ±∞ ...
35:           case others
36:             U ← EXTRACTFIELDSUNUM(utag, e, f, leftNotRight)   ▷ according to Figure 4.7
      return [U]
```

Algorithm 3 BMF store

```
1:  function BMFST(U, address, UE, MBB, leftNotRight)
2:    utagWidth ← GETUTAGWIDTH(UE)                    ▷ get the utag memory footprint
3:    eWidth ← U.esm1                                 ▷ get the exponent memory footprint
4:    if MAXUSIZE(UE) < MBBSIZE(MBB) then             ▷ UNUM max size can fit in the current MBB
5:      switch [U, leftNotRight] do                   ▷ check data to be stored
6:        case ISSNAN(U)
7:          MEM[address] ← sNaN                       ▷ store full length sNaN
8:        case ISQNAN(U)
9:          MEM[address] ← qNaN                       ▷ store full length qNaN
10:       case ISEXACTINF(U)
11:         MEM[address] ← ±∞ ↓                       ▷ store full length ±∞ ↓
12:       case ISNEXACTINF(U, leftNotRight)
13:         MEM[address] ← ±∞ ...                     ▷ store most compact encoding ±∞ ...
14:       case others
15:         Unorm ← BMF(U, UE, MBB)                   ▷ normalize and round U according to MBB
16:         MEM[address] ← Unorm.utag                 ▷ store the BMF-ized utag
17:         MEM[address+utagWidth] ← Unorm.e          ▷ store the BMF-ized exponent
18:         MEM[address+utagWidth+eWidth] ← Unorm.f   ▷ store the BMF-ized fraction
19:   else                                            ▷ UNUM max size can not fit in the current MBB
20:     switch [U, leftNotRight] do                   ▷ check data to be stored
21:       case ISSNAN(U)
22:         MEM[address] ← sNaN′                      ▷ store compact sNaN
23:       case ISQNAN(U)
24:         MEM[address] ← qNaN′                      ▷ store compact qNaN
25:       case ISEXACTINF(U)
26:         MEM[address] ← ±∞′ ↓                      ▷ store compact ±∞ ↓
27:       case ISNEXACTINF(U, leftNotRight)
28:         MEM[address] ← ±∞ ...                     ▷ store most compact encoding ±∞ ...
29:       case others
30:         Unorm ← BMF(U, UE, MBB)                   ▷ normalize and round U according to MBB
31:         MEM[address] ← Unorm.utag                 ▷ store the BMF-ized utag
32:         MEM[address+utagWidth] ← Unorm.e          ▷ store the BMF-ized exponent
33:         MEM[address+utagWidth+eWidth] ← Unorm.f   ▷ store the BMF-ized fraction
```

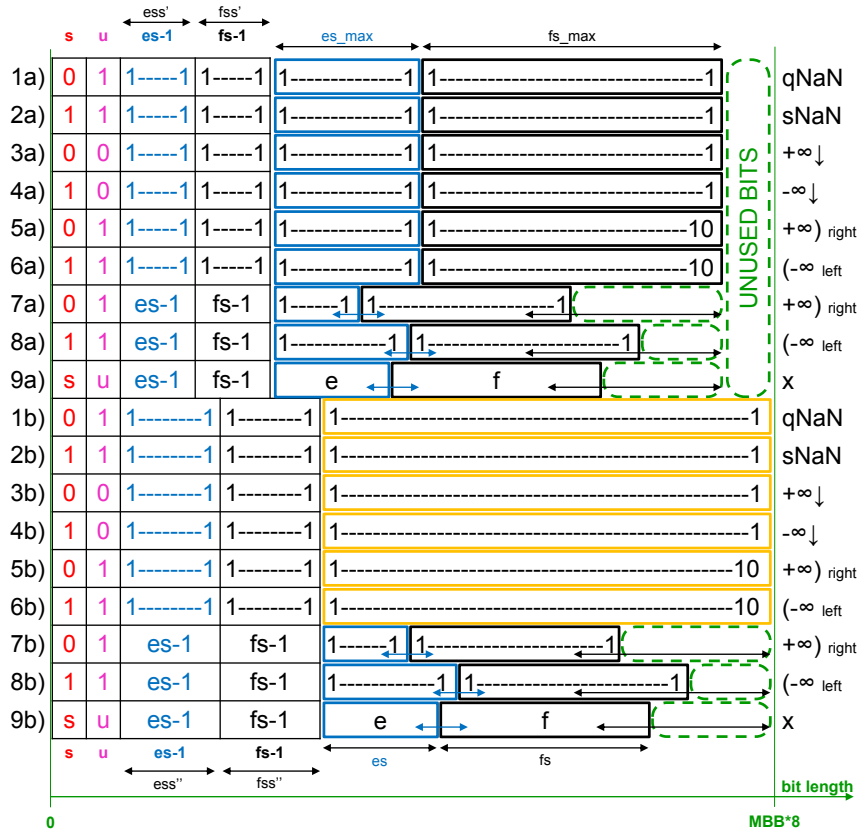


Figure 4.8: Alternative BMF binary encoding varying the MBB and the UE parameters

1-6b. The idea used here is to use some or all the remaining bits (in addition to the utag) to differentiate distinguished values.

4.4 Issues on supporting variable-precision formats

According to the previous sections, the encoding of Variable-Precision (VP) Floating-Point (FP) formats is more complicated than conventional floats. This complexity raises several issues both under the hardware realization and precision perspectives. Sections 4.4.1, 4.4.2, 4.4.3, and 4.4.4 describe these issues. This section expands Section 2.4 by adding all the issues identified during the design phase of this work.

4.4.1 Hardware overhead due to variable-length fields

The choice of modifying the UNUM VP FP format in BMF weakens Gustafson's claims, but it is a necessary condition for using it in a real computing system. Unlike IEEE 754 numbers, in VP FP formats, the exponent and fraction fields have variable bit-width, and their position is not known a priori. FP units (FPU) require fixed-length interfaces, fixed-size hardware internal operators, and separated exponent and fraction fields for the FP computation. The VP FPU hardware, to support VP FP formats, needs *conversion functions*, between the memory interface and the FPU core, to encode and decode the exponent and fraction fields between the memory format and the internal one used for computation.

These conversion functions (not needed for the IEEE formats) complicate the VP FPU hardware increasing the circuit area, the latency, and the power (and energy) consumption. At the cost of reducing the VP FPU throughput, the latency can be masked by adding pipeline stages

(i.e., during load and store operations). Some hardware optimizations can minimize the power consumption overhead (e.g., register clock gating).

These conversion functions need additional hardware in the VP LSU to convert the FPU internal registers that, by definition, have a fixed bit-width. This circuit area overhead, due to these conversion functions, cannot be avoided, it can only be minimized.

4.4.2 Instruction encoding issues

Another hardware-related potential issue is that VP FP numbers require some additional information needed for the encoding and decoding of numbers in memory. For example, for UNUMs, the UNUM Environment (UE [2]), and MBB if the BMF is applied. In the hardware FPU, these types of information must be stored somewhere. They could be stored either within each instruction of the Instruction Set Architecture (ISA) of the unit or in dedicated status registers defined by the ISA.

Encoding this information in the instruction implies to increase the number of supported instructions significantly. All the instructions regarding load and store of VP FP data in memory must be re-defined for every supported value of UE and MBB. This redefinition is a potential problem because there is the possibility of not having enough free opcodes to encode all the load-store instructions in the ISA. Moreover, this complicates the FPU decode unit and the compiled assembly.

This work follows the other option that encodes these types of information in *status registers* inside the VP FPU. These registers require dedicated instructions (and additional clock cycles) to be updated. However, the number of instructions added in the ISA is negligible, and the generated assembly is independent of the data precision. This ISA strategy allows the recycling of the software code when the data precision must be changed.

4.4.3 Data-dependent error bounds

Variable-Precision (VP) Floating-Point (FP) formats are designed to compact their FP exponent field for small exponent values. For example, representing the exponent value +1 on 16 bits is needless: two bits are enough. In this case, the fraction precision of a FP number can be augmented (ideally) of 14 bits. In other words, for smaller exponent values (in magnitude), the exponent bits can be "harvested" and used to expand the fraction FP field.

On one side, the exponent "harvesting techniques" used in VP FP formats (Section 3.1) intuitively make the reader think that it can reduce the computational error at the end of a computation. On the other side, the non-constant number of mantissa bits can make it challenging to bound the error of arithmetic operations linearly. This statement is generally valid for all the VP FP formats known in state of the art: UNUM and posit.

4.4.4 Encoding overhead of variable-precision

Some bits of the VP number must encode somehow the threshold between the exponent and fraction fields (nor their lengths). This field can have fixed-length (e.g., the *exponent size* in the UNUM format [2]) or can have variable length (e.g., the *regime bits* in posit). In both cases, the bits used to express this field cannot express the FP fraction. For exponent values of magnitude above a threshold that depends on the setting of the VP FP format, this brings a systematic precision loss, expressed in a given number of fractional bits, compared to an IEEE-754-like FP format of the same size. This encoding overhead behaves worse when the memory subsystem imposes a granularity larger than the bit.

The standard UNUM format [2] is inefficient to support in hardware since affine accesses in memory cannot be guaranteed (Section 4.3). The modified UNUM format specified in Section 4.3.4 guarantees affine memory accesses, and, if the MBB parameter is large enough, a minimum number of mantissa bits can be guaranteed. However, this number of bits is lower than the one that could be available in a custom IEEE-like float of the same size (due to the VP descriptor field). Moreover, the redundant exponent encodings in the UNUM format lower, even more, this number of mantissa bits. For state of the art on forward and backward analysis perspective, this potentially requires VP FP numbers on more bits than the IEEE 754-like FP numbers with custom exponent and fraction sizes.

Posit is not exempt from these issues, it just avoids the redundancy of the exponent encoding, at the cost of sacrificing some bits of precision. It simplifies the conversion functions designing their algorithms around a leading-zero-one-count hardware block. However, for large exponent values, posit requires more bits than UNUM to encode the same exponent value. Other posit-related issues are listed in [74].

4.5 Hardware architecture for the variable-precision computing unit

This section hosts this study's main contribution: the specifications for a VP hardware architecture that handles VP data in memory and inside the VP FP Unit (FPU). The final target of this work is to have a VP computing system capable of running scientific computing applications on memory data. Its size can vary from less than ten bits to several hundred. To achieve that, this work leverages on a hardware architecture based on four main principles:

- VP FP formats are used to express variables in memory with possibly different precisions than those supported by the IEEE 754 format.
- VP FP variables in memory have a *programmable* footprint at compile time. Using IEEE 754 FP formats, it is possible to vary the precision of variables by using different memory formats. With VP FP formats, the memory footprint (and the precision) of VP FP variables, belonging to the VP FPU, can be programmed by the user, but it must be static after compile time. If not, the compiler can not allocate VP FP data structure in memory (e.g., VP FP arrays).
- The *representation* of VP FP variables in memory and inside the VP FPU is different. The reason for this is twofold: 1/ the precision used in memory may be different from that used for internal computation; 2/ the best representation of data in memory is different from the best representation of data inside the VP unit. For the first point, the programmer can use different precision settings for internal computations and memory data. For the second point, unlike VP memory formats, the VP exponent and fraction fields must be in a fixed position to be processed. The latter is a hardware constraint.
- The VP FPU must be as close as possible to the main processor, and it must be used directly by *extending* the system Instruction Set Architecture (ISA). In this way, the VP FPU can be used like conventional FPUs, and the control code that involves VP FP values is executed without slowing down the main core pipeline. The instructions added in the system ISA must also include conversion instructions between the existing data types (i.e., float, double) and new ones (i.e., UNUM, ubound).

4.5.1 Overview of the RISC-V-based system

Figure 4.9 depicts the global overview of the computing system implemented in this work. RISC-V was chosen as the development platform due to its configuration flexibility provided

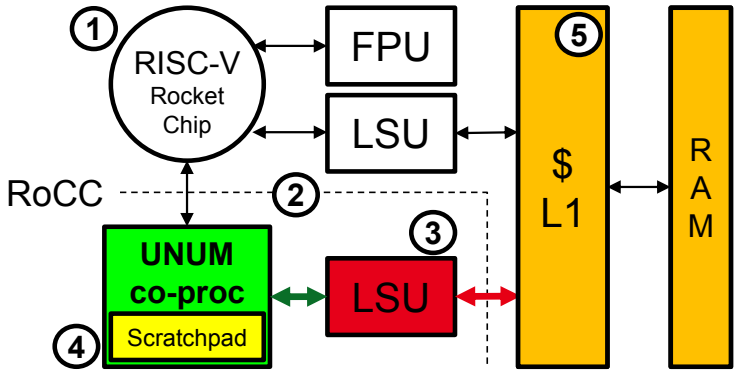


Figure 4.9: High-level overview of the variable-precision computing system

by the Rocketchip RISC-V generator [7]. This platform can easily integrate new hardware blocks as RISC-V coprocessors. Thus, the VP FPU will be embedded in the platform as a *coprocessor*. The choice to adopt a RISC-V architecture is purely opportunistic because it is open-source. Other architectures like GPUs or hardware accelerators are not considered since they do not guarantee a fast interaction with the main core, especially in the control code that exploits FP computation.

The system platform is made of a RISC-V core ① connected to its native system: its 64 bits FPU, its memory hierarchy, and its peripherals. The VP coprocessor ④ has dedicated interfaces with the main core and the memory subsystem ⑤ (L1 cache memory), which guarantees much flexibility from the design perspective and ensures higher performance during memory operations (compared to a loosely coupled coprocessor). These interfaces are grouped into the RoCC interface ② [75]. The RoCC interface is a hardware interface, provided by the Rocketchip generator, that provides to the user to embed custom coprocessors in a RISC-V environment.

The design principle embraced in the hardware realization of the VP coprocessor is to have two different VP FP formats for the numbers in memory and the ones inside the coprocessor scratchpad (its register file). The VP FP memory format is designed to be as compact as possible to maximize the precision of variables while minimizing the algorithm's computational error. The length of the memory format can be tuned with a byte-granularity. The VP FP format used inside the coprocessor Register File (RF) is less compact and less flexible. Like conventional FPUs, the mantissa and exponent fields are kept separate. The exponent has fixed length, and the mantissa length can be tuned with a 64-bit granularity. In this way, the coprocessor can work internally on data with fixed-size representations, having standard hardware for internal hardware operators. For more details about the RF format, please refer to Section 4.5.3.

The conversion between those two formats is handled, during load and store operations, by a dedicated Load and Store Unit (LSU ③). All the complexity related to the VP FP memory formats (UNUM and ubound), in particular the positions of the variable-size fields, are supported by the LSU. This LSU encodes and decodes the supported VP FP memory format(s) to be processed inside the coprocessor. It supports BMF as an extension of the UNUM type I format (for details see Section 4.3) with MBB boundaries expressed in a specific number of bytes (Maximum Byte Budget according to Definition 1). The converted data go into the internal coprocessor RF. Once the converted data are in the coprocessor RF, its entries are used among operations.

4.5.2 Architecture of the accelerator

The coprocessor architecture is inspired by Schulte [49] and is pipelined with a fixed 64-bits internal data-path (comparable with conventional 64 bits FPUs). The mantissa fields of the coprocessor RF entries are divided into chunks (words [49]) of 64 bit each. Thus, the scratchpad

precision can be tuned with a 64-bit granularity. The 64 bit bound is chosen to keep under control in terms of the silicon area (required by the synthesized chip) and the clock cycle latency, of the coprocessor hardware arithmetic logic (e.g., adders, shifters, multipliers). Each pipeline stage of the coprocessor is based on a stop-and-wait protocol to process the RF mantissa fields.

In order to better support interval VP formats, the coprocessor provides hardware acceleration for interval computation. In this way, the programmer can use intervals like scalar variables without having to write software functions to support Interval Arithmetic (IA) basic operations. To properly support IA, the coprocessor RF hosts VP FP intervals.

The coprocessor supports the basic FP arithmetic hardware operators. It embeds a comparator (to compare RF entries), an adder (to support addition and subtractions between RF entries), and a multiplier (to support multiplications between RF entries). Division and other FP operations that implement polynomials are implemented by software routines iterating on the existing hardware. All the hardware operators support both interval and scalar computation. For the latter, to save energy and computation time, the arithmetic inside the hardware operators is simplified by using only one interval endpoint data-paths out of the two available, and by bypassing all the hardware logic needed to support intervals. Moreover, only one of the two endpoints of a register-file entry is used.

The design implementation is parametric so that many aspects (for instance, the internal data-path bit-width, the maximum variables dynamics, and the RF size) can be customized based on the application. The set of parameters chosen in this article is as follows:

- The internal coprocessor data-path bit-width of is set to 64 bit, as previously exposed.
- The coprocessor RF contains 32 VP FP intervals (ubound). This number of entries matches the one specified for integer and FP computing in the RISC-V ISA.
- VP FP memory variables use the modified UNUM type I format (with BMF, Section 4.3) supporting all the UEs from the (1, 1), which corresponds up to 2 bits for exponent and mantissa, up to the (4, 9) one, which corresponds to a maximum of 16 bits for the exponent and 512 fractional bits. For efficiency constraints, the hardware designer can tune the maximum supported length for exponent and mantissa with a power-of-two granularity.
- Each RF interval endpoint can express FP mantissa fields up to 512 of precision. This number must be large enough to support the maximum supported UE, but its size can increase with a power-of-two granularity.

All the results reported in the next sections correspond to this set of parameters.

This section is structured as follows: Section 4.5.3 details the format used in the coprocessor RF and the RF internal hardware organization. Section 4.5.4 explains how to have easy control of a VP FPU employing status registers.

4.5.3 Choice of the variable-precision format of the register file

On the one hand, the coprocessor has to support data in memory spanning between less than ten bits up to several hundred. These data use a VP FP format in which the exponent and mantissa sizes and positions (within the number) are encoded within the number. These data use a VP FP format in which the number self-encodes (within the number) the exponent and mantissa sizes and positions. On the other hand, an efficient FP hardware needs to have the mantissa and exponent fields in two separate fields, with their length known and explicit a priori. A dedicated Load and Store Unit (LSU) takes in charge of the complexities of re-aligning the VP FP fields from the memory format into the one of coprocessor scratchpad. This section specifies the FP format used in the coprocessor scratchpad, its Register File (RF), and its internal organization.

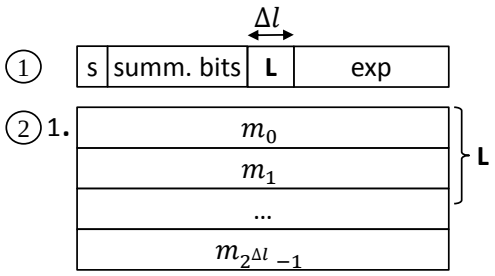


Figure 4.10: The gbound endpoint (gnumber) binary format

The coprocessor internal data-path bit-width is fixed to 64 bits. With this, the coprocessor LSU has to extract the mantissa from the VP data in memory and map it into a new mantissa with a granularity of 64 bits.

This granularity implies to have in the coprocessor RF mantissa fields with different precisions depending on the VP data loaded from memory. Instead of having several ISA instructions of the same type (e.g., `add`) that support several input data precisions, this work proposes one single instruction per operation where the data precision information is the data pointed in the scratchpad. Thus, the described VP FP coprocessor works with pointer-based computation. For memory accesses, every access to the RF entries is done through pointers, and the number self-encodes its mantissa length.

For each VP FP number in the coprocessor RF is divided into two parts: the *header* ① and the *mantissa* ②. The *header* contains all the FP fields that are not the mantissa. It is made of a sign s, an exponent *exp*, six summary bits, *summ. bits*, and the L field. The *mantissa* field is divided in chunks (of 64 bits each). According to the maximum UNUM length supported in memory (512 bits of mantissa), this work supports eight chunks of 64-bits each. In this work, all the mantissa fields are normalized (the hidden bit is always set to 1) to facilitate the computation inside the VP hardware operators.

Among all the header fields, the L one is the most important. It gives the number of used mantissa chunks in the VP FP number. In a VP number inside the coprocessor RF, the most significant chunk is m_0, and the least significant one is m_L. Thus, the number of chunks that moved into the coprocessor is minimized according to its precision. For the implementation of this, its length (Δl) is three bits.

The *exp* header field hosts the exponent value of the VP number. It is encoded in two's complement, and its length is two bits longer than the maximum length that the exponent can have in memory (16+2 bits). The two additional bits are used to encode the sign of the exponent and to be able to encode all the subnormal values supported by the VP memory format, having the mantissa fields expressed in normal form.

The *summary bits* are flags used to encode particular values. The first one is `is_zero`, which is set if the VP number is zero. This bit is needed because, with a normalized mantissa, it is impossible to represent the value 0. The next three are `is_nan_quiet`, `is_nan_signaling`, and `is_inf`. They are used to representing Not a Number and infinity ($\pm\infty$) values, which may arise during computation.

The last two summary bits are used to provide Interval Arithmetic (IA) support within the VP coprocessor: they are `is_nan_quiet` and `is_close`. The `is_nan_quiet` flag is used to represent a value between the maximum representable one and infinity [2]. The `is_close` flag is an endpoint termination flag: it is set if the interval endpoint is closed, it is unset if the interval endpoint is open.

Figure 4.11 depicts a simplified scheme of the internal organization of the coprocessor RF. It is mainly divided into two separated memories: the one to host the VP numbers headers, and the one to host the VP numbers mantissa fields. Since the coprocessor supports IA, the header

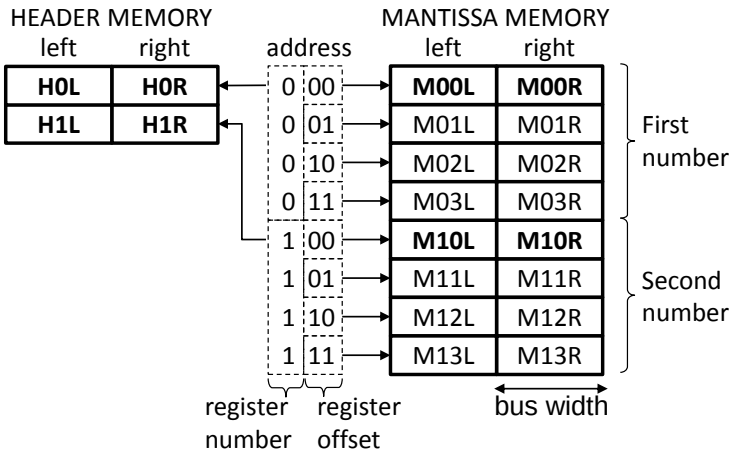


Figure 4.11: Internal organization of the register file

and the mantissa memories are split in two, to host the two endpoints of the interval (left and right) for each RF entry. In this work, the VP RF hosts 32 intervals with up to 8 mantissa chunks per each interval endpoint. For simplicity, this figure represents a RF with only two entries and four mantissa chunks per interval endpoint.

The RF entries are accessed through an *address*. This address is made of two parts, one that points the *register number* and one that points the *register offset*. The register number addresses the entire coprocessor RF. The register offset addresses a specified couple of mantissa chunks within a RF entry. For a given read-write operation on the RF, the read-modified headers (HxL, HxR) and mantissa chunks (MxyL, MxyR) are the ones pointed by the register number (x, e.g., 1) and the register offset (y, e.g., 01) fields of the input address.

Operations on higher precisions than 64 bits are done by iterating on the existing hardware. The protocol implemented by the RF to handle high-precision data is simple: In a RF read operation, the first chunk of the RF entry, pointed by the register number bits of the RF address, is provided to the VP hardware operator. If the L fields of the headers in input to the VP operator is different from zero, the VP operator has to require the access to the coprocessor RF and ask for all the missing chunks in the input (one per clock cycle) providing in the RF address all the missing register offsets.

In interval computation, it is assumed that the VP operators can handle the two endpoints of the interval in output in parallel. For this reason, the RF provides in output both interval endpoints. The coprocessor can also work in scalar mode. In this mode, it only uses the left endpoint.

4.5.4 Improving code efficiency through status registers

One of the main problems in VP computing is writing software code (e.g., C) where the kernel of the program (the central VP computational part) is unchanged and, at the same time, the precision of its variables may change. This problem is driven by the fact that software languages are typed and, depending on the variable datatype, specific (data-dependent) assembly instructions are generated by the compiler (e.g., `ld` or `lw` to load 64 or 32 bits data respectively on RISC-V). This section proposes to host the information about the configuration of VP formats (in memory and the coprocessor scratchpad), the variables memory-footprint, and the operation rounding mode, in *Status Registers* (SR) inside the VP coprocessor.

Having a computing model based on SRs allows generating assembly code independent of the data type of variables. In other words, the computing precision information inside SRs allows making assembly operations (e.g., load and store) independent on the type of the variables in memory (or in the coprocessor RF). The VP coprocessor has five internal SRs: DUE,

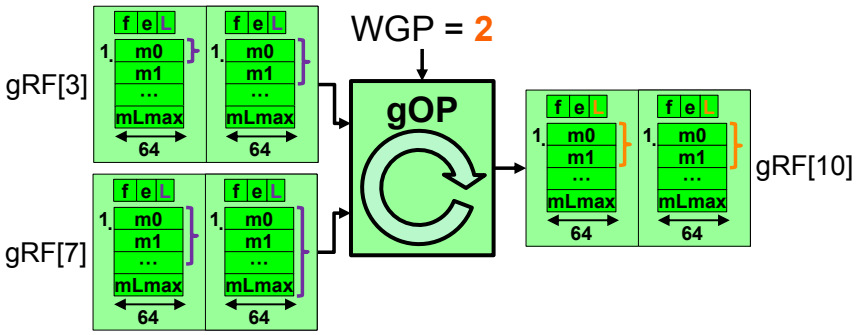


Figure 4.12: Example of usage of the Working G-layer Precision status register

SUE, MBB, WGP, and RND. The first three are related to the memory number representation, and the last two tune the computation inside the coprocessor.

The *Default UNUM Environment* (DUE) SR stores the default UNUM Environment (UE, expressed in the tuple of values ESS and FSS) [2] used during the load-store of UNUM-ubound operations from-in the main memory.

In the case of UE conversion operations among data, the modification of the DUE status register for every number would have a significant impact in terms of execution time. To speed up the UE conversion operations in main memory, the *Secondary UNUM Environment* (SUE) SR stores the secondary (optional) UE that can be used for UNUMs-ubounds load-store operations from-in the main memory. Having two UE SRs (DUE and SUE) allows for faster UE conversion operations and faster loading and storage between different UEs.

The third status register is the *Maximum Byte Budget* (MBB) SR. It supports the BMF format (Section 4.3) that adapts the UNUM type I format for all the possible MBB values. Its content can vary according to the application needs. It controls (at the granularity of p=8 bits), the maximum slot size for read-write in main memory.

While DUE, SUE, and MBB SRs configure the format and precision of data in memory, the *Working G-layer Precision* (WGP) SR configures the computation precision inside the VP coprocessor. It stores the maximum precision (expressed in a given number of 64-bit mantissa chunks) that the result of a coprocessor operation can have. Figure 4.12 depicts the WGP usage. A VP operator (gOP) bounds the output precision of its output (L) at the value encoded in the WGP SR (2). With this SR the user can implement *fused operations*[3]inside the coprocessor by increasing (if higher precision is needed) or reducing (to speed up the operations) the computation precision. In some applications, incrementing the computation precision in the computational unit may reduce the precision required by the variables in memory, leading to a potential speedup of the overall application.

The *RouNDing mode* (RND) SR hosts the rounding mode that all the VP operators (including the Load and Store Unit, LSU) must respect. It can have four possible values: Round To Interval (RTI), Round To the Nearest (half away from zero, RTN), Round-Up (RDU), and Round Down (RDD). If the RND SR is set to RTI, the interval endpoint data-paths are enabled within the VP hardware operators. Furthermore, the rounding modes used in both data-paths meet the interval arithmetic requirements (i.e., enlarge the interval endpoints toward $\pm\infty$). If the RND SR is set to RTN, or RDU, or RDD, the VP coprocessor units work in scalar mode, and the rounding rule used is congruous on the selected value in the SR.

Those registers can be driven by the compiler depending on the VP FP variable definition or by the inline assembly. All the SRs values are sampled at instruction fetch. The VP coprocessor makes sure that every SR modification has an immediate effect on the next fetched instruction.

[3] *Fused operations* are a sequence of operations done within the computing unit without passing through the memory subsystems. In other words, fused operations are a section of an assembly program that uses only operands that are coming from the unit register file or the unit scratchpad (e.g., an internal accumulator).

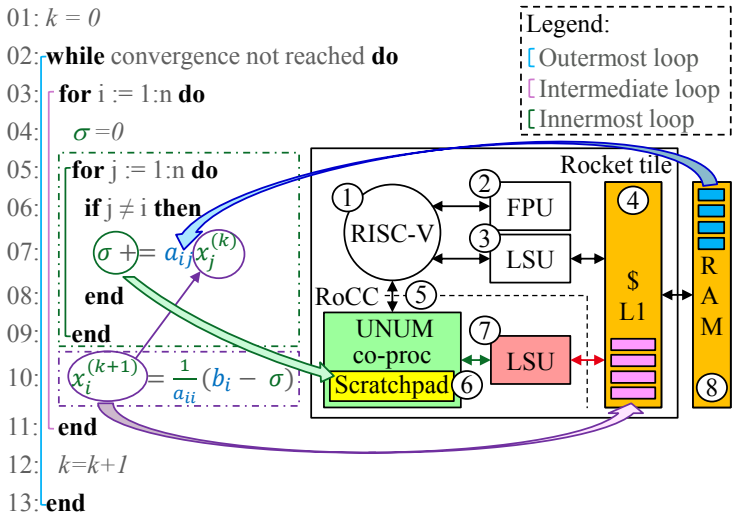


Figure 4.13: The coprocessor architecture and its programming model: structure and variable-length mapping for an iterative solver kernel

In this way, the VP coprocessor user (the assembly programmer or the compiler) has only to focus on setting the right precision of the variables needed in a give section of the assembly code. Section 4.6 provides additional details on the SRs usage.

4.6 The programming model for the variable-precision computing unit

Writing Variable-Precision (VP) software can be very complicated since, with conventional hardware support, for every supported precision, a dedicated software function must be associated. This complexity is because the conventional software languages (e.g., C) are typed, and for each variable, a precision is implicitly associated. This code structure leads to having assembly code (generated by the compiler) that contains assembly instructions dependent on the memory footprint of variables. Section 4.5.4 describes the Status Registers (SR) of the VP Floating-Point (FP) coprocessor, used to host the configuration parameters of VP variables in memory. The use of these SRs is necessary for the writing of VP software, where the precision of variables is not attached to themselves but is stored in status registers.

This section shows how to use the SRs of the VP coprocessor to write software for VP FP computing applications. Our hardware is best suited for applications that follow a standard scheme among VP kernels. Figure 4.13 depicts how to use the VP hardware for a canonical algorithm software example.

To explain the programming model proposed by this work, Figure 4.13 leverages a simplified Jacobi solver as a model example based on several nested loops. In this model, all the input and output data are stored in main memory ⑧ on conventional IEEE 754 FP formats, which can usually express the input-output data with enough precision. The general idea is to use VP variables to improve the computation precision in the algorithm innermost part. This model example is based on several nested loops.

The *outermost loop* level manages the result convergence: according to some criteria, it may decide to restart the process increasing the computation precision (DUE-SUE-MBB or WGP). Every time that the precision must be changed, the VP coprocessor SRs are updated.

We rely on main memory ⑧ for hosting the data used by the kernel (source data, in IEEE format, and working set). Passing through the system cache ④, they are processed in the coprocessor scratchpad ⑥.

The *innermost loop* is usually processed in the coprocessor RF ⑥. It is meant for the accumulation of (possibly) many partial products, e.g., line or column-wise accumulation in linear algebra, or specific interpolation stepper for ODE solvers. Our ISA allows us to create any arbitrary accumulation; the dot product is one example. In our model, this accumulation uses one of the 32 available VP registers (with up to 512 bits of mantissa). Provided that the compiler does an excellent job at static scheduling, these accumulations may be interleaved in the RF as long as space is available. The compiler must maximize the amount of computation in the RF ⑥ to minimize the precision losses and maximize the performance. However, the spilled variables can be stored in the processor L1 cache (if appropriately sized) with the UNUM format (with up to 256 bits of mantissa).

The *intermediate loop* uses the accumulated intermediate results for updating the intermediate vector result. This intermediate vector is usually too large to fit into the coprocessor scratchpad. Therefore, it is often convenient to use the close memory ④ for storage. Thus, the cache size is a primary key parameter here. The usage of VP allows safeguarding enough precision without wasting memory. This process, which may consist of several loops, stops when some convergence criteria are met.

4.7 The ISA for the variable-precision computing unit

Previous sections show the global overview of the VP hardware architecture (Section 4.5), its VP Register File (RF) organization (Section 4.5.3), and its programming model (Section 4.6) that leverages dedicated Status Registers (SR, Section 4.5.4). This section proposes the Instruction Set Architecture (ISA) to program the VP computing system.

4.7.1 Programmer view for the variable-precision computing unit

Before diving into the ISA details, it is necessary to introduce the RISC-V architecture. RISC-V provides the user access to 33, XLEN bit wide, integer registers **x0**-**x31**, and **pc** (this work address XLEN=64). **x0** is hardwired to the constant 0. **x1**-**x31** are 31 General-Purpose Registers (GPR). The **pc** holds the program counter that contains the address of the current instruction.

The Variable-Precision (VP) coprocessor provides to the user (through assembly instructions) access to the VP Register File (RF). According to the nomenclature introduced in [2], since we support intervals in the VP RF, this work defines a *gbound* as the interval hosted in a RF entry, and it defined *gnumbers* as the interval endpoints of one entry. This RF contains 32 entries, named **g0**-**g31**, which hold gbound values. This choice simplifies the internal architecture and the compiler (that have to provide to the high-level programmer the UNUM facilities).

As explained in Section 4.2, the VP coprocessor can work either in interval mode (gbound) or in scalar (gnumber) mode. The UNUM and ubound VP formats are strictly dedicated to data storage in the main memory. A Load and Store Unit (LSU) dedicated to the VP coprocessor handles the conversion between the memory and RF formats. During load operations, the user loads UNUMs (or ubounds) stored in memory and implicitly converts them in gbounds, that will be stored in the RF. During store operations, the user implicitly converts a gbound (or a gnumber) hosted in the RF in the UNUM-ubound format (also controlling its memory footprint). Once that is converted, it is ready to be stored in the main memory.

4.7.2 Base instruction formats

The VP coprocessor is interfaced with the main RISC-V core through the RoCC interface. This interface allows the exchange of data between the main RISC-V core, the L1 cache, and the coprocessor. Figure 4.14 depicts the three main instruction formats allowed by the RISC-V ISA (I10/I5/R). They are divided into several fields:

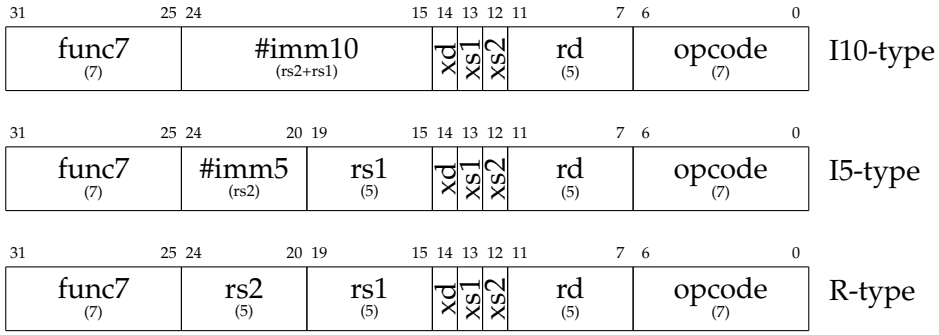


Figure 4.14: RISC-V Instruction Set Architecture (ISA): instruction format types

- opcode: This field contains the opcode of the RISC-V instruction. Only four opcodes are available to identify coprocessors in the system. In this work, only one opcode value (that this work renames as OP-CUST) is reserved to map the expanded instructions to work with the VP coprocessor.
- funct7: Contains the instruction code of the coprocessor instruction; its values are a designer choice.
- rs1, rs2, rd: Source (rs1, rs2) and destination (rd) registers.
- xs1, xs2, xd: These three flags, are set if the source (rs1, rs2) or destination (rd) registers exists. These flags are used to select data from the main processor or provide data to the coprocessor.

The I5 and I10 types allow encoding the immediate values in the instruction.

The RISC-V ISA provides the possibility to extend its standard instruction set with instruction set expansions. In particular, the existing RISC-V ISA can be expanded (on 32-bit instructions) in order to map additional coprocessor instructions. Their encoding must respect the R-type format depicted in Figure 4.14. For this expansion, out of all the possible instruction opcode values provided in the RISC-V ISA, four are reserved to expand the already existing ISA to introduce coprocessor instructions. These four opcodes give the user the possibility to add to the ISA instructions for up to four different coprocessors. When the running instruction has one of these opcode values, the RISC-V activates the target coprocessor and provides to the target coprocessor the instruction and the required data.

When the opcode points a VP coprocessor instruction, this instruction is forwarded up to the VP coprocessor. Once it reaches the coprocessor, it has to be decoded. The RISC-V ISA keeps the sources (rs1 and rs2), and destination (rd), registers in the same position in all the formats to simplify decoding. However, if these fields are not activated by the register selection flags (xs1, xs2, xd), they can be used for other purposes by the UNUM coprocessor. If one of the register selection flags (xs1, xs2, xd) is unset, the correspondent register field (rs1, rs2, rd) can be used from the coprocessor for other purposes. For instance, these unused bits can be used to encode pointers to a memory inside the coprocessor (e.g., a register file), to encode immediate values.

4.7.3 Status registers instruction set

This section presents the instructions designed to read and change the Status Registers (SR) value inside the UNUM coprocessor described in Section 4.5.4. Figure 4.15 depicts how the *status register word* (SR word) encodes all the SRs supported by the VP coprocessor. For both the

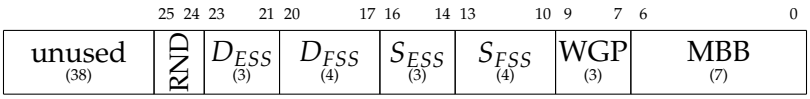


Figure 4.15: The status registers binary encoding: this encoding contains all the status registers supported by the variable-precision coprocessor

	31 25	24 20	19 15	14	13	12	11 7	6 0	
	func7	rs2	rs1	xd	xs1	xs2	rd	opcode	R-type
	7	5	5	1	1	1	5	7	
①	`lusr`	unused	unused	1	0	0	Xd	OP-CUST	
②	`susr`	unused	Xs1	0	1	0	unused	OP-CUST	
③	`ldue`	unused	unused	1	0	0	Xd	OP-CUST	
④	`lsue`	unused	unused	1	0	0	Xd	OP-CUST	
⑤	`lmbb`	unused	unused	1	0	0	Xd	OP-CUST	
⑥	`lwgp`	unused	unused	1	0	0	Xd	OP-CUST	
⑦	`lrnd`	unused	unused	1	0	0	Xd	OP-CUST	
⑧	`sdue`	unused	Xs1	0	1	0	unused	OP-CUST	
⑨	`ssue`	unused	Xs1	0	1	0	unused	OP-CUST	
⑩	`smbb`	unused	Xs1	0	1	0	unused	OP-CUST	
⑪	`swgp`	unused	Xs1	0	1	0	unused	OP-CUST	
⑫	`srnd`	unused	Xs1	0	1	0	unused	OP-CUST	

Table 4.1: Status registers instruction set

DUE and the SUE SRs, three bits encode the ESS UE field, and four bits encode the FSS UE field. This configuration allows us to support the maximum UE supported by the VP coprocessor: (ess=4, fss=9). The WGP SR is encoded on three bits. In this way, the WGP SR can tune the precision of the coprocessor operations with up to 8 mantissa chunks. The WGP SR is encoded on three bits. To support the UE (4,9) and to be able to store it without losing precision, seven bits are dedicated to encoding the MBB SR. The four supported rounding modes (Section 4.5.4) in the coprocessor are encoded on two bits in the RND SR. At the reset of the VP coprocessor, all these SRs are set to zero. To use the coprocessor, the user must set the values of these SRs before any coprocessor operation.

Table 4.1 presents the instructions to handle the internal status registers, and Table 4.2 shows their assembly syntax. `lusr` ① is the instruction that loads the content of the SR word that contains all the coprocessor status registers: DUE, SUE, MBB, WGP, and RND. The values are stored in the GPR pointed by Xd.

`susr` ② is the instruction that updates (set) the content of the SR word: all the DUE, SUE, MBB, WGP, and RND SRs. The new values of the SRs are passed into the VP coprocessor through the GPR pointed by Xs1. With these two instructions, the programmer can update the content of every single SR (or group of SRs) by reading them (LUSR), applying some bit-masking techniques, and updating them back (S-USR).

This sequence of operations can be excessive to update a single SR. For this reason, the ③-⑫ instructions modify atomically single SRs in the SR word. The modification of one of these SRs does not imply any automatic update on the other SRs. If the programmer enlarges only the DUE, the MBB SR (and all the other SRs) keeps its previous value. Thus, if the value of MBB is lower than the maximum mantissa length that the new DUE can have, the programmer must expect an output value with a precision corresponding to the maximum one allowed by the BMF-sized value for the configured DUE and MBB.

All the hardware operators of the VP coprocessor must carry on their pipeline, to guarantee the program execution, the values of the SRs when the assembly operation involving the hardware operator is fetched. In this way, the hardware operators can finish their execution with the correct set of the coprocessor SR.

Access to the coprocessor status registers values	
Assembler:	`lusr Xd`
Action:	Xd = [unused, RND, DUE, SUE, WGP, MBB]
Assembler:	`susr Xs2`
Action:	[unused, RND, DUE, SUE, WGP, MBB] = Xs1
Assembler:	`ldue Xd`
Action:	Xd = [DUE]
Assembler:	`lsue Xd`
Action:	Xd = [SUE]
Assembler:	`lmbb Xd`
Action:	Xd = [MBB]
Assembler:	`lwgp Xd`
Action:	Xd = [WGP]
Assembler:	`lrnd Xd`
Action:	Xd = [RND]
Assembler:	`sdue Xs1`
Action:	[DUE] = Xs1
Assembler:	`ssue Xs1`
Action:	[SUE] = Xs1
Assembler:	`smbb Xs1`
Action:	[MBB] = Xs1
Assembler:	`swgp Xs1`
Action:	[WGP] = Xs1
Assembler:	`srnd Xs1`
Action:	[RND] = Xs1

Table 4.2: Status registers instruction set utilization

	31 25	24 20	19 15	14	13	12	11 7	6 0	
	func7	rs2	rs1	xd	xs1	xs2	rd	opcode	R-type
	7	5	5	1	1	1	5	7	
①	`ldgu/ldub`	unused	Xs1	0	1	0	gRd	OP-CUST	
②	`stul/stur/stub`	gRs2	Xs1	0	1	0	unused	OP-CUST	
③	`ldgu.next/ldub.next`	gRs2	Xs1	1	1	0	Xd	OP-CUST	
④	`stul.next/stur.next/stub.next`	gRs2	Xs1	1	1	0	Xd	OP-CUST	
⑤	`ldgu.s/ldub.s`	unused	Xs1	0	1	0	gRd	OP-CUST	
⑥	`stul.s/stur.s/stub.s`	gRs2	Xs1	0	1	0	unused	OP-CUST	
⑦	`ldgu.next.s/ldub.next.s`	gRs2	Xs1	1	1	0	Xd	OP-CUST	
⑧	`stul.next.s/stur.next.s/stub.next.s`	gRs2	Xs1	1	1	0	Xd	OP-CUST	
⑨	`ldg`	unused	Xs1	0	1	0	gRd	OP-CUST	N/A[a]
⑩	`stg`	gRs2	Xs1	0	1	0	unused	OP-CUST	N/A[a]

Table 4.3: Load and store instructions set

[a] This instruction is not supported in the current coprocessor configuration

4.7.4 Instruction set for load and store operations

As aforementioned, UNUMa and ubounds are strictly considered only as memory formats. For internal operations, the VP coprocessor supports in its RF another FP format: the gbound (Section 4.5.3). The VP coprocessor LSU handles the conversions between these two formats during load and store operations. This section presents the instruction set to program the coprocessor LSU. Sections 4.7.4 and 4.7.4 detail the load and store behavior during memory operations. Section 4.7.4 describes the load and store instructions supported by the VP coprocessor.

The UE and the MBB used during the load operation are hosted in the DUE, SUE, and MBB coprocessor SRs. Load and store operations do not consider the WGP coprocessor SR: data are loaded and stored, giving the precedence to the memory format. Store operations may round the data according to some values combinations of DUE and MBB, or SUE and MBB. The RND coprocessor SR hosts the rounding mode (Section 4.5.4) implemented by the storage unit.

Load behavior

A load operation loads a UNUM-ubounds from main memory and converts it into internal gbounds. The *U2G* sub-unit of the coprocessor LSU does the conversion operation. The converted value is stored into the target register in the RF.

Store behavior

A store operation stores a coprocessor RF entry in main memory on the number of bytes specified by the MBB SR, on the UE specified in the DUE or SUE coprocessor SR. Store operations are performed on two steps.

The G2U sub-unit converts the gnumber-gbound coming from the RF into a UNUM-ubound. The converted result is stored in a buffer internal at the storage unit. The used UE for the conversion is stored into the DUE or the SUE status register. The interval or scalar information for the input-output formats, and the UE selection among the DUE and SUE SRs, are stored within the instruction.

Before going to memory, the converted UNUM-ubound is bounded (in size) by the BMF sub-unit. This unit adjusts the memory footprint of the intermediate UNUM-ubound according to the Maximum Byte Budget defined in the MBB status register. The rounding rule used for this operation is hosted in the RND coprocessor SR.

Load and store instructions

Table 4.3 presents the load and store instructions supported by the VP coprocessor. Table 4.4 shows the assembly syntax for their usage. The `ldgu` instruction loads a (scalar) UNUM stored

Load a UNUM from the Xs1 address into the gRs1 gbound	
Assembler:	`ldgu{.s} gRd, Xs1`
Action:	gRd = U2G(MEM(Xs1), {DUE,SUE}, MBB)
Load a ubound from the Xs1 address into the gRs1 gbound	
Assembler:	`ldub{.s} gRd, Xs1`
Action:	gRd = U2G(MEM(Xs1), {DUE,SUE}, MBB)
Store a gRS1 gbound endpoint in main memory in UNUM at the address Xs1	
Assembler:	`stu(l,r){.s} gRs1, Xs1`
Action:	MEM(Xs1) = BMF(G2U(gRs1[L,R], {DUE,SUE}), {DUE,SUE}, MBB)
Store the gRS1 gbound in main memory in ubound at the address Xs1	
Assembler:	`stub{.s} gRs1, Xs1`
Action:	MEM(Xs1) = BMF(G2U(gRs1, {DUE,SUE}), {DUE,SUE}, MBB) Xd = next_addr
Load a UNUM from the Xs1 address into the gRs1 gbound, providing the next address in Xd	
Assembler:	`ldgu{.s}.next Xd, gRs1, Xs1`
Action:	gRs1 = U2G(MEM(Xs1), {DUE,SUE}, MBB) Xd = next_addr
Load a ubound from the Xs1 address into the gRs1 gbound, providing the next address in Xd	
Assembler:	`ldub{.s}.next Xd, gRs1, Xs1`
Action:	gRs1 = U2G(MEM(Xs1), {DUE,SUE}, MBB) Xd = next_addr
Store a gRS1 gbound endpoint in main memory in UNUM at the address Xs1, providing the next address in Xd	
Assembler:	`stu(l,r){.s}.next Xd, gRs1, Xs1`
Action:	MEM(Xs1) = BMF(G2U(gRs1.[L,R], {DUE,SUE}), {DUE,SUE}, MBB) Xd = next_addr
Store the gRS1 gbound in main memory in ubound at the address Xs1, providing the next address in Xd	
Assembler:	`stub{.s}.next Xd, gRs1, Xs1`
Action:	MEM(Xs1) = BMF(G2U(gRs1, {DUE,SUE}), {DUE,SUE}, MBB) Xd = next_addr
Load a gbound from memory at the address Xs1 and store it in gRd	
Assembler:	`ldg`[a] `gRd, Xs1`
Action:	gRd = MEM(Xs1)
Store the gRs1 gbound in main memory at the address Xs1	
Assembler:	`stg`[a] `gRs1, Xs1`
Action:	MEM(Xs1) = gRs1

Table 4.4: Load and store instruction set

[a] This instruction is not supported in the current coprocessor configuration

	31 25	24 20	19 15	14	13	12	11 7	6 0	
	func7	#imm5	rs1	xd	xs1	xs2	rd	opcode	I5-type
	7	5	5	1	1	1	5	7	
①	`mov.g.g`	unused	gRs1	0	0	0	gRd	OP-CUST	
②	`movll/movlr`	unused	gRs1	0	0	0	gRd	OP-CUST	
③	`movrl/movrr`	unused	gRs1	0	0	0	gRd	OP-CUST	
④	`mov.x.g`	#imm5	Xs1	0	1	0	gRd	OP-CUST	
⑤	`mov.g.x`	#imm5	gRs1	1	0	0	Xd	OP-CUST	

Table 4.5: Move instruction set

in main memory at the address pointed by Xs1 and stores the loaded result, converted in gbound format, in both the endpoints of the gbound pointed by gRd. The `ldub` instruction loads the ubound stored in main memory at the address pointed by Xs1 and stores the loaded result, converted in gbound format, in the gbound pointed by gRd. For both instructions, the loaded data must be in the BMF format (see Section 4.3), its memory footprint must be stored in the MBB SR, and its UE must be defined in the DUE SR.

The `stul` and `stur` instructions store the left or right interval endpoint of the gbound pointed by gRs1 in main memory, converted in UNUM-ubound format, at the address pointed by Xs1. The `l` and the `r` characters in the function name indicate that the gbound endpoint to be stored is respectively the left and the right one. The `stub` instruction stores the gbound pointed by gRs1 in main memory, converted in ubound format, at the address pointed by Xs1. Similarly, for the load operations, for these instructions, the loaded data must be in the BMF format, its memory footprint must be stored in the MBB SR, and its UE must be defined in the DUE SR. In the case of re-rounding (for more details, see Section 4.3.3), the rounding policy will be specified in the RND coprocessor SR.

`ldgu.next`, `ldub.next`, `stul.next`, `stur.next`, and `stub.next` implement similar functionalities of `ldgu`, `ldub`, `stul`, `stur`, and `stub` respectively. The only difference is that they return, through Xd, the address of the first free byte available in the main memory. These instructions support the "compacted addressing mode" presented in Section 4.2.2 (Figure 4.4b).

The `.s` instructions, `ldgu.s`, `ldub.s`, `stul.s`, `stur.s`, `stub.s`, `ldgu.s.next`, `ldub.s.next`, `stul.s.next`, `stur.s.next`, and `stub.s.next`, implement similar functionalities of `ldgu`, `ldub`, `stul`, `stur`, `stub`, `ldgu.next`, `ldub.next`, `stul.next`, `stur.next`, and `stub.next`, respectively. The only difference is that the memory operations use the UE stored in the SUE coprocessor SR, instead of using the UE stored in the DUE SR.

The `ldg`[4] and the `stg`[4] are two coprocessor instructions that can be used to spill the coprocessor RF entries in memory. The `ldg` instruction loads a gbound stored in main memory at the address pointed by Xs1, in the RF entry pointed by gRd. The `stg` instruction stores gbound pointed by gRs1 in the main memory at the address pointed by Xs1. In the current coprocessor configuration, to store a gbound in the main memory, the coprocessor requires a memory slot of 8*2*64+2*64 = 1152 bits. This memory slot is big enough to host the two gbound interval endpoints made of one header and height mantissa chunk (for each endpoint).

The `ldg` and `stg` instructions are used to support the `spill` and the `refill` functions. These functions make a raw copy of the coprocessor RF in the main memory and vice versa. This copy is a *snapshot* of the internal configuration of the coprocessor during a context switch. These snapshots are used to *roll back* to previous coprocessor states. However, they are not supported in this version of the UNUM coprocessor since they can be implemented by a software routine made of `mov.g.x` and `mov.x.g` instructions.

Copy a gbound into another one	
Assembler:	`mov.g.g gRd, gRs1`
Action:	gRd = gRs1
Copy the content of a GPR into a specific gbound chunk	
Assembler:	`mov.x.g gRd{[#imm5]}, Xs1`
Action:	gRd[#imm5] = Xs1
Copy a gbound chunk into a GPR	
Assembler:	`mov.g.x Xd, gRs1{[#imm5]}`
Action:	Xd = gRs1[#imm5]
Set a gbound to a predefined constant value	

Table 4.6: Move instruction set

	31–25	24–20	19–15	14	13	12	11–7	6–0	
	func7	rs2	rs1	xd	xs1	xs2	rd	opcode	R-type
	7	5	5	1	1	1	5	7	
①	`gcmp`	gRs2	gRs1	1	0	0	Xd	OP-CUST	
②	`gadd/gsub/gmul`	gRs2	gRs1	0	0	0	gRd	OP-CUST	
③	`gguess/gradius`	unused	gRs1	0	0	0	gRd	OP-CUST	
④	`gdiv`	gRs2	gRs1	0	0	0	gRd	OP-CUST	N/A[a]
⑤	`gsqrt/gisqrt`	unused	gRs1	0	0	0	gRd	OP-CUST	N/A[a]
⑥	`gneg/gabs`	unused	gRs1	0	0	0	gRd	OP-CUST	N/A[a]

Table 4.7: Arithmetic operators instruction set

[a]This instruction is not supported in the current coprocessor configuration

4.7.5 Instruction set for move operations

This section presents the instructions dedicated to moving data between register from, into, and within the VP coprocessor. Table 4.5 presents these instructions, and Table 4.6 shows their assembly syntax. These instructions are presented in Table 4.5 and their assembly syntax is shown in Table 4.6. The `mov.g.g` ① instruction, like a standard `mov` operation, copies the gbound content pointed by gRs1 in the one pointed by gRd.

The `movll`, `movlr`, `movrl`, and `movrr` instructions (② and ③) copy the content of the gnumber (left for `movll` and `movlr`, right for `movrl` and `movrr`) pointed by gRs1, in the gnumber (left for `movll` and `movrl`, right for `movlr` and `movrr`) pointed by gRd.

The `mov.x.g` instruction ④ copies the RISC-V GPR pointed by Xs1, in the #imm5-th word of the gbound pointed by gRd.

The `mov.g.x` instruction ⑤ copies of the #imm5-th word of the gbound pointed by gRs1, into the RISC-V GPR pointed by Xd. For `mov.x.g` and `mov.g.x` the #imm5 field is mandatory. If its value is 0, the XLEN-bit data pointed by Xs1 contains the gbound header of the left interval endpoint. If its value is between 8 and 15, the XLEN-bit data pointed by Xs1 contains the '(#imm5)-8'-th mantissa chunk of the target gbound. If its value is 16, the XLEN-bit data pointed by Xs1 contains the gbound header of the right interval endpoint. If its value is between 24 and 31, the XLEN-bit data pointed by Xs1 contains the '(#imm5)-24'-th mantissa chunk of the target gbound.

4.7.6 Instruction set for gbound arithmetic instructions

This section describes the instructions defined to do computation between coprocessor RF entries (gbounds). Table 4.7 presents these instructions, and Table 4.8 shows their assembly syntax. Depending on if the RND SR is set to RTI or not, the following operations are done by

[4]This instruction is not supported in the current coprocessor configuration

gbound comparison	
Assembler:	`gcmp Xd, gRs1, gRs2`
Action:	Xd = CMP(gRs1, gRs2)
gbound addition	
Assembler:	`gadd gRd, gRs1, gRs2`
Action:	gRd = rnd(gRs1 + gRs2, WGP)
gbound subtraction	
Assembler:	`gsub gRd, gRs1, gRs2`
Action:	gRd = rnd(gRs1 - gRs2, WGP)
gbound multiplication	
Assembler:	`gmul gRd, gRs1, gRs2`
Action:	gRd = rnd(gRs1 * gRs2, WGP)
gbound guess	
Assembler:	`gguess gRd, gRs1`
Action:	gRd = rnd(GUESS(gRs1), WGP)
gbound radius	
Assembler:	`gradius gRd, gRs1`
Action:	gRd = rnd(RADIUS(gRs1), WGP)
gbound division	
Assembler:	`gdiv`[a] `gRd, gRs1, gRs2`
Action:	gRd = rnd(gRs1 / gRs2, WGP)
gbound square root	
Assembler:	`gsqrt`[a] `gRd, gRs1`
Action:	gRd = rnd(SQRT(gRs1), WGP)
gbound inverted square root	
Assembler:	`gisqrt`[a] `gRd, gRs1`
Action:	gRd = rnd(1/SQRT(gRs1), WGP)
gbound negation	
Assembler:	`gneg`[a] `gRd, gRs1`
Action:	gRd = -gRs1
gbound absolute value	
Assembler:	`gabs`[a] `gRd, gRs1`
Action:	gRd = |gRs1|

Table 4.8: Arithmetic operators instruction set utilization

[a] This instruction is not supported in the current coprocessor configuration

enabling the interval data-paths in the coprocessor operators or not. In the case that the RND SR is not set to RTI, the following operations are done in scalar mode using only the left endpoint of the pointed RF entries. In both cases, the implemented rounding during an arithmetic operation is specified by the RND SR during the instruction fetch.

`gcmp` compares two gbounds pointed by gRs1 and gRs2 and stores all the comparison flags in GPRF at the entry pointed by Xd. The flags contained in the returned data are divided into two groups. The first one is dedicated to comparing intervals: EQual (EQ), Not EQual (NEQ), Not Nowhere EQual (NNEQ), Greater Than (GT), Lower Than (LT). This solution allows us to make all the possible combined comparisons (e.g., NNEQ or GT).

The second group is dedicated to comparing the scalar interval endpoint independently: Left gRs1 endpoint GT Left gRs2 endpoint (LLGT), Left gRs1 endpoint LT Left gRs2 endpoint (LLLT), Left gRs1 endpoint EQ Left gRs2 endpoint (LLEQ), Left gRs1 endpoint GT Right gRs2 endpoint (LRGT), LRLT, LREQ, RLGT, RLLT, RLEQ, RRGT, RRLT, and RREQ.

In both groups, the checks whether the comparison results are true or not, are done in the main core, using binary masks and branches equal to zero (BEZ). This ISA expansion also provides instructions to compare each of these flags individually (e.g., `gcmp.eq` checks if two intervals are precisely equal to each other, `gcmp.rlgt` checks if the right endpoint of gRs1 is greater than the left endpoint of gRs2). These additional instructions facilitate the coprocessor compiler to take branches without masking the result of the more generic `gcmp` instruction.

`gadd`, `gsub`, `gmul` respectively adds, subtracts and multiplies the gbounds pointed by gRs1 and gRs2 together and store the result in the gbound pointed by gRd.

`gguess` converts the gbound pointed by gRs1 in $(o(midpoint), o(midpoint))$ format (both endpoint RTN: rounded to the nearest half away from zero). It stores the result in the gbound pointed by gRd.

`gradius` extract from the gbound pointed by gRs1 the interval width between the left and right interval endpoint. The interval width ($|\lceil right - left \rceil|$) is encoded in the format $(\underline{width}, \overline{width})$ (one endpoint rounded down and the other rounded up). It stores the result in the gbound pointed by gRd. For `gadd`, `gsub`, `gmul`, `gguess`, and `gradius` operations, the output rounding precision (of the computed result) is the one defined into the *WGP* status register.

The following instructions are not implemented in hardware. They are implemented with software routines using the aforementioned embedded operators[5]. `gdiv`[6] divides the gbounds pointed by gRs1 and gRs2 together and store the result in the gbound pointed by gRd. `gsqrt`[6] computes the square root of the gbound pointed by gRs1 and store the result in the gbound pointed by gRd. `gisqrt`[6] computes the inverse of the square root of the gbound pointed by gRs1 and store the result in the gbound pointed by gRd. `gneg`[6] negates the gbound pointed by gRs1 and store the result in the one pointed by gRd. `gabs`[6] computes the absolute value of the gbound pointed by gRs1 and store the result in the gbound pointed by gRd.

4.7.7 Instruction set for variable-precision conversion operations

This section presents the conversion instructions supported by the VP coprocessor. Table 4.9 presents these instructions, and Table 4.10 shows their assembly syntax.

In this implementation, the VP coprocessor exchanges FP values through GPRs. The main drawback of this approach is that, at the software level, the user must handle the re-mapping on floating-point registers. In C, this is not possible to do with simple cast operations. `Union` structures must be used.

The `mov.d.g` ①, `mov.f.g` ③, and `mov.h.g` ⑤ instructions convert respectively the `double`, the `float`, the `half` FP values (`binary64`, `binary32`, and `binary16` in IEEE-754 parlance) hosted

[5]For this purpose a LUT is needed to handle also the exceptional cases (0/inf, inf/inf, ...)

[6]This instruction is not supported in the current coprocessor configuration

	31 25	24 20	19 15	14	13	12	11 7	6 0	
	func7	#imm5	rs1	xd	xs1	xs2	rd	opcode	I5-type
	7	5	5	1	1	1	5	7	
①	`fcvt.d.g`	#imm5	Xs1	0	1	0	gRd	OP-CUST	
②	`fcvt.gl.d/fcvt.gr.d`	unused	gRs2	1	0	0	Xd	OP-CUST	
③	`fcvt.f.g`	#imm5	Xs1	0	1	0	gRd	OP-CUST	
④	`fcvt.gl.f/fcvt.gr.f`	unused	gRs2	1	0	0	Xd	OP-CUST	
⑤	`fcvt.h.g`	#imm5	Xs1	0	1	0	gRd	OP-CUST	
⑥	`fcvt.gl.h/fcvt.gr.h`	unused	gRs2	1	0	0	Xd	OP-CUST	

Table 4.9: Formats conversions instruction set

Convert a `double` variable hosted in a GPR into gbound	
Assembler:	`fcvt.d.g gRd, Xs1`
Action:	gRd = (gbound) Xs1
Convert the left endpoint of a gbound into `double`, storing the value in a GPR	
Assembler:	`fcvt.gl.d Xd, gRs1`
Action:	Xd = (double) gRs1.left
Convert the right endpoint of a gbound into `double`, storing the value in a GPR	
Assembler:	`fcvt.gr.d Xd, gRs1`
Action:	Xd = (double) gRs1.right
Convert a `float` variable hosted in a GPR into gbound	
Assembler:	`fcvt.f.g gRd, Xs1`
Action:	gRd = (gbound) Xs1
Convert the left endpoint of a gbound into `float`, storing the value in a GPR	
Assembler:	`fcvt.gl.f Xd, gRs1`
Action:	Xd = (float) gRs1.left
Convert the right endpoint of a gbound into `float`, storing the value in a GPR	
Assembler:	`fcvt.gr.f Xd, gRs1`
Action:	Xd = (float) gRs1.right
Convert a `half` float variable hosted in a GPR into gbound	
Assembler:	`fcvt.h.g gRd, Xs1`
Action:	gRd = (gbound) Xs1
Convert the left endpoint of a gbound into `half` float, storing the value in a GPR	
Assembler:	`fcvt.gl.h Xd, gRs1`
Action:	Xd = (half) gRs1.left
Convert the right endpoint of a gbound into `half` float, storing the value in a GPR	
Assembler:	`fcvt.gr.h Xd, gRs1`
Action:	Xd = (half) gRs1.right

Table 4.10: Formats conversions instruction set utilization

in the RISC-V GPR pointed by Xs1, in the gbound pointed by gRd. With these instructions, both gbound endpoints are updated with the converted IEEE FP value.

The `mov.gl.d` ②, `mov.gl.f` ④, and `mov.gl.h` ⑥ instructions convert the left endpoint of the gbound pointed by gRs1, respectively into a `double`, `float`, `half` FP value in the RISC-V GPR pointed by Xd. The RND SR hosts the rounding rule in doing this conversion (for more details, see Section 4.5.4). The `mov.gr.d` ②, `mov.gr.f` ④, and `mov.gr.h` ⑥ instructions implement the same functionality of the previous three instructions but on the right endpoint of the gbound pointed by gRs1.

4.8 Compiler prototype: support variable-precision with a vpfloat datatype

New architectures often require software interfaces and programming models to take advantage of the new hardware functionalities. For example, multi-core and many-core systems often involve an OpenMP programming interface [76]. Alternatively, CUDA [77] and OpenCL [78] are widely used for Graphics Processing Units (GPU). Complementary to these programming models, this work proposes a programming abstraction of the Variable-Precision (VP) hardware architecture. This impact the code at individual thread level (it changes the code only inside the thread) and it does not impact the compatibility of multi-threading with parallel programming. This section covers the software support for VP, highlighting the most critical aspects of the language and compiler extensions. This work is the result of a collaboration with Tiago Trevisan Jost.

Supporting VP in software can be done in several manners. A first approach can be to emulate VP computing with software libraries (e.g., MPFR [9]). In this case, the user can have VP support by using library calls. These libraries rely their calculation on data structures in main memory that are used to host VP variables. The software library approach does not require dedicated hardware support but has three main disadvantages.

The first one is that it may saturate the ALU of the processor, preventing other programs from running. The second one is that, at least in MPFR, the Floating-Point (FP) variables mantissa fields are located, with malloc, in different memory regions than the other variables. This dynamic allocation breaks the principle of data locality for which the cache memories are designed. The third one is that software libraries use several intermediate variables that may pollute the processor's cache slowing down the computation performance.

This work proposes to import VP in the C language by defining a new data format to alleviate the issues mentioned above. The main idea is to provide to the programmer an explicit way to tune the memory footprint (and the precision) of FP variables, with byte granularity, in order to be able to boost the numerical stability of algorithms. This work does not claim to fix all the numerical stability of algorithms by increasing the variable sizes. However, it provides new tools to the programmers to investigate new techniques to help users in algorithm error analysis.

The ideal VP FP data format must:

- Leverage all the existing compiler features (such as automatic memory allocation, stack handling, code simplification, and constant propagation).
- Exploit at best the cache of the processor by preserving the data locality principle.

In order to expose at best the VP features in this new FP format, this work proposes the `vpfloat` data format. The main feature of this format is that it is a programmable data format where its length is programmable at compile time with byte granularity (which will be kept constant at execution time). With this property, the programmer has accurate control over the variables'

precision, and the compiler, which controls their memory footprint, can efficiently allocate the main memory variables. The syntax suggested by this work is showed below:

```
vpfloat< <'type'>, <parameters...> > my_variable;
```

The main idea is to provide a VP FP data type which instructs the compiler about the characteristics of every VP FP variable. By selecting the `<'type'>`, the programmer can define the VP FP format to be used in the main memory. The other fields, `<parameters...>`, are additional fields that instruct the format. For example, the following line defines a VP FP variable named `my_var` with a custom IEEE (three fields) format on 17 bytes, where 1 bit encodes the sign, 6 bits encode the exponent, and 129 bits encode the mantissa:

```
vpfloat<'f',6,129> my_var;
```

With this syntax, the programmer assigns the desired precision, and the maximum dynamics, for each variable of the target software application. With this format, the user has a more accurate precision control on each variable in his program.

The compiler has all the types of information that it needs to handle `vpfloat` variables: it knows their size, memory format, and precision. With these types of information, the compiler can easily set up the VP coprocessor setting its status register and allocate the VP variables in memory. The main difficulty on the compiler is to carry the variables settings among the compilation step and to handle VP variables in the stack with different memory footprints. However, since the variables have a fixed memory footprint at their declaration, the issues mentioned above can be addressed.

Handling the coprocessor SRs may be a little tricky. For example, the internal coprocessor precision (WGP) can be chosen: 1/ automatically by the compiler according to the slot size specified in the variable declaration, always respecting the 64-bit granularity; or 2/ manually by the user using the `susr` and `swgp` assembly instructions. However, it is not impossible to find a heuristic to set them. For example, WGP can be higher than MBB since the former must be a multiple of 64, and the latter can be byte-configured always to guarantee the maximum precision in memory.

Thanks to the `type` field, the `vpfloat` data type can support several FP formats. For example, the `vpfloat` data type can also encode other VP formats such as UNUMs [2]:

```
vpfloat<'u',4,8,23> my_unum;
```

where, 4 and 8 define the *ess* and *fss* values of the so-called UNUM Environment (UE, [2]), and 23 defines the maximum size, in bytes, of the variable. The `vpfloat` data type can also encode ubounds (interval made of two UNUMs, [2])

```
vpfloat<'b',4,8,23> my_ubound;
```

or posits [3]:

```
vpfloat<'p',2,23> my_posit;
```

where the 2 identifies the *ES* parameter of the posit variable [3] and 23 identifies the variable memory-footprint (always expressed in bytes).

The compiler can instruct the target hardware unit about the data format characteristics (memory footprint, memory format, and format parameters) through dedicated instructions or hardware status registers. Conversion functions must be provided to the compiler to support VP operations among integers, among conventional IEEE 754 floats and different `vpfloat` data type configurations. These conversion functions convert memory formats into an internal VP representation that can support all the precisions required by all the formats.

This work is aware that a hardware unit can support VP FP variables up to a given limit. In the case that the precision required by the user is higher than the one provided by the hardware, the compiler can continue to provide VP support by implementing VP computations with software libraries (i.e., MPFR) and alternating the computation between hardware and software depending on the operations output precision.

```
1 vpfloat pi(int n) {
2     vpfloat<'u',3,8,42> sum = 0.0;
3     int sign = 1;
4     for (int i = 0; i < n; ++i) {
5         sum += sign/(2.0v*i+1.0v);
6         sign *= -1;
7     }
8     return 4.0*sum;
9 }
```

Listing 4.1: Usage example of the VP unit: the algorithm calculates π iteratively on 42 bytes using the Taylor series $\pi = 4 \cdot \sum_{i=0}^{n} \frac{(-1)^i}{2 \cdot i + 1}$

4.8.1 Example: how to write a variable-precision program

This section shows how the programmer can use the `vpfloat` primitive to write a Variable-Precision (VP) Floating-Point (FP) program in a high-level language (i.e., C). Listing 4.1 shows toy C code to calculate π using an iterative approach based on a Taylor series approximation with `vpfloat` type numbers. In line 2, the `sum` variable is declared as a 42-byte VP FP number on the UNUM format. The `for-loop` iteratively calculates $\pi/4$ on `sum` (lines 4 to 7). Notice that there is no need for casting the `i` and `sign` variables to `vpfloat` since the compiler can automatically handle type conversions when necessary. A constant suffixed by `v` represents a `vpfloat` to differentiate when the user wants constants with high precision. For constants, if not explicit (e.g., through casts), the constant precision will be one of the destination variables, in this case, `sum`.

Listing 4.2 shows a snippet of the assembly code that the compiler, which supports the ISA expansion proposed by this work, should generate for Listing 4.1. Coprocessor instructions are found at lines 8-12, 14-15, 17-18, 24-26 and 30. The code starts by setting up MBB and WGP to their correct values (lines 2-5). Notice that MBB is set to 42 bytes, which corresponds to variable slot size. WGP is set to 384 bits (48 bytes) since it must be a multiple of 64 bits. From lines 7-20, the `for-loop` calculates the pi function through coprocessor instructions. Line 15 calls the function which implements the VP division. As explained in Section 4.7, the coprocessor can operate in either intervals or scalars, as it adopts the UNUM format. The presence of `gguess` instructions indicates that the unit was configured to work with intervals. A compiler flag (*vpfloat-scalar*) configures the system to work with scalars: i.e., it can be used to eliminate the generation of `gguess` instructions.

```
1           ...
2           addi      a3, zero, 42 # MBB = 42 Bytes
3           addi      a4, zero, 6  # WGP = 6*64 bits
4           smbb      a3
5           swgp      a4
6           ...
7   .LBB1_2: sext.w   a0,  s0 # %for.body
8           fcvt.x.g gt0, a0
9           gmul      gt0, gt0, gs1
10          gguess    gt0, gt0
11          gadd      ga1, gt0, gs2
12          gguess    ga1, ga1
13          sext.w    a0,  s1
14          fcvt.x.g ga0, a0
15          call      vpdiv
16          neg       s1,  s1
17          gadd      gs0, gs0, ga0
18          gguess    gs0, gs0
19          addiw     s0,  s0, 1
20          blt       s0,  s2, .LBB1_2
21  # %bb.3:      # %for.cond.cleanup.loopexit
22          lui       a0,  %hi(.LCPI1_4)
23          addi      a0,  a0, %lo(.LCPI1_4) # constant 4.0
24          sext.w    a0,  a0
25          fcvt.x.g ga0, a0 # constant 4.0
26          gmul      ga0, gs0, gt0
27          gguess    ga0, ga0
28          j .LBB1_5
29  .LBB1_4: lui      a0,  %hi(.LCPI1_0)
30          addi      a0,  a0, %lo(.LCPI1_0)
31          ldgu      ga0, (a0)
32  .LBB1_5:      # %for.cond.cleanup
33          ...
34          ret
```

Listing 4.2: The snippet of the assembly code for Listing 4.1: VP instructions are in lines 8-12, 14-15, 17-18, 24-26, and 30.

Chapter 5

Micro-architecture for the variable-precision computing unit

Chapter 4 describes the system-level architecture for the Variable-Precision (VP) scientific computing unit presented in this work (Figure 4.9). It is made of the main core (RISC-V, ①) connected to its native system: its 64 bits Floating-Point Unit (FPU), its memory hierarchy, and its peripherals. Section 4.5 describes the motivations for this architecture. The VP computing unit is implemented as a RISC-V coprocessor ④. Through the RoCC interface ②, the VP unit is connected with the main core and the system L1 cache ⑤. The coprocessor accesses the cache through a dedicated Load and Store Unit, LSU ③. This chapter focuses on the micro-architecture of the VP coprocessor system (③ and ④).

Figure 5.2 depicts the VP coprocessor micro-architecture. It is based on a three-stage pipeline: decode ①, execute ②, and write back ③. Vertical dashed lines ¦ denote synchronization barriers between the pipeline stages. The fetch and memory stages are not needed since the coprocessor is a slave of the central core: the main core does the instruction fetch and the address generation for memory operations.

The ports of the RoCC interface are 64 bits wide. For this reason, the data-flow inside the coprocessor is structured on ports 64 bits wide.

As detailed in Section 4.1, the coprocessor supports two different formats: one dedicated for the coprocessor scratchpad, and one dedicated to the memory. According to Section 4.3, this coprocessor architecture represents Floating-Point (FP) data in memory using a modified version of the UNUM and ubound (intervals of UNUMs) formats. Data in the scratchpad are stored in another format, detailed in Section 4.5.3.

The architecture presented in this work supports interval arithmetic. Thus, the coprocessor scratchpad is a Register File (RF) which hosts intervals. Like in [2], intervals in the scratchpad are named *gbound*, and interval endpoints are named *gnumbers*. Conversions between the scratchpad and memory formats are handled, during load and store operations, by the coprocessor Load and Store Unit (LSU, ③ in Figure 4.9, LSU in Figure 5.2).

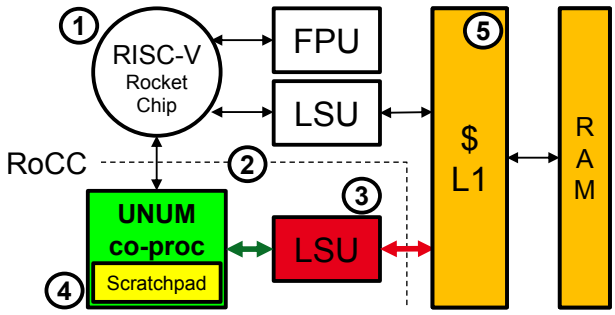


Figure 4.9: High-level overview of the variable-precision computing system (repeated from page 52)

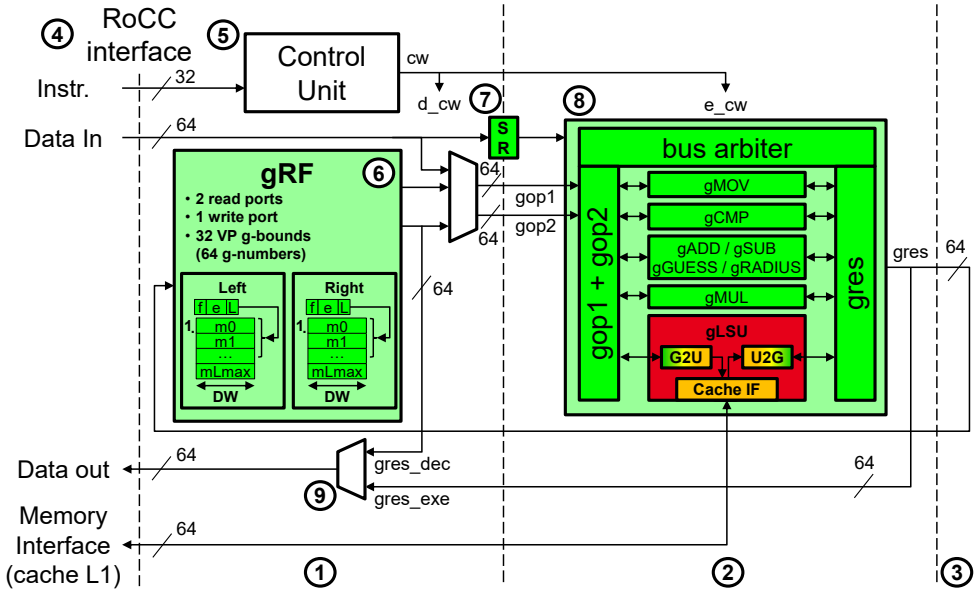


Figure 5.2: The internal pipeline of the variable-precision coprocessor

The decode stage In the coprocessor's *decode stage*, the control unit ⑤ decodes instructions in input and generates the control words for the decode and execute stages. The decode stage hosts the gbound RF ⑥ (gRF)[1] with two read ports and one write port. It hosts 32 Variable-Precision (VP) gbounds with normalized mantissa fields. The number of entries is justified by the 5 bits dedicated to addressing registers of the RISC-V 32-bit instruction format (2^5=32). Section 4.5.3 provides details on the chosen gnumbers format.

The decode control word in the decode stage selects the gRF content (the input data) that has to be propagated in the execute stage. The gRF controller ⑥ keeps track of the running instructions and their data dependencies. Data dependencies (e.g., read after write) are tracked by associating a dirty bit for each gRF gbound. Every time that one instruction requires to write its result in a gRF entry, the dependency tracker sets the corresponding dirty bit.

If one of the required inputs in the decoded coprocessor instruction has its dirty bit set, a data dependency is triggered. In case of data dependency (in this case, read after write), the gRF controller enables a *busy* signal which prevents the main RISC-V core from providing a new instruction to the VP coprocessor. The busy signal tells the main core whether the coprocessor is free to receive a new instruction in the next clock cycle.

Mantissa fields in the gRF are divided into chunks of 64 bits each (Section 4.5.3), and the actual number of used chunks (out of the maximum available) is encoded within the number. By default, the gRF sends, within one clock cycle, the headers and the first mantissa chunk of the gRF data pointed by the input's instruction. If a unit in the execute stage needs more than one mantissa chunk, the gRF can be forced, through a dedicated port, to provide in the next clock cycles the missing mantissa chunks. This mechanism guarantees minimum latency for small numbers (one clock cycle) while minimizing the number of flip-flops involved in moving data within the coprocessor.

The ISA specified in this work (Section 4.7) decorrelates operations from data precision using internal status registers (SR). These registers are DUE and SUE (divided in ESS and FSS), WGP, MBB, and RND (Section 4.5.4). They are located between the decode and the execute pipeline stages ⑦, and they are used to configure the hardware operators in the execute stage. Depending on their value, the operators in the execute stage react accordingly.

[1] The gRF is located inside the coprocessor because the RISC-V architecture, the one generated with Rocketchip [7] (with its RoCC interface), does not allow to add custom RFs in the design easily.

Their values are accessed through dedicated assembly instructions (e.g., with the `susr` and `lusr` instructions, Section 4.7.3). These SRs are located between the decode and execute stage. In this way, their values can be updated in one clock cycle, and the consistency of the order of operations provided in the input is maintained.

The execute stage The execute stage hosts the arithmetic hardware operators of the VP coprocessor ⑧ that execute the ISA instructions specified in Section 4.7. These operators support gbound operations. They are pipelined. In this way, independent operations can run parallel either on different operators or on different operator pipeline stages.

The execute stage hosts six operators that implement the ISA instructions presented in Section 4.7. The gCMP operator handles the `gcmp` instruction. The gMUL operator handles the `gmul` instruction. The gADD operator handles the `gadd`, `gsub`, `gguess`, and `gradius` instructions. Section 4.7.6 presents these instructions. The gMOV operator handles the `mov*` instructions (Section 4.7.5). The gCNVT operator handles the `fcvt.*` instruction (Section 4.7.7). The LSU operator handles the `ld*-st*` instructions (Section 4.7.4), supporting the memory and field organizations introduced in Section 4.2.

These operators ⑧ receive operands and output results distributed on 64 bits chunks. The decode ⑦ and write back time barriers have buses 64 bits wide. All the operators who have inputs or outputs on more than one mantissa chunk have to request the input-output bus control to retrieve-write all the chunks from-into the gRF.

Those requests are handled by the *bus arbiter* units that ensure the correct data propagation among the pipeline and handle the results coming from all the variable-latency operators. There are two bus arbiters: one to control the decode buses (one for each operand), and one to control the write-back bus. Both use a stop-and-wait protocol. They receive the bus requests from the operators in ⑧. Depending on whether the buses are free or busy (taken by another operator), they grant or deny the bus control through some acknowledge signals. Once an operator takes control of the input (or output) bus, the coprocessor pipeline is stalled (the coprocessor is busy from the RISC-V perspective) until when the operator controlling the bus revokes the request. When an operator owns a bus, it can require a pair of mantissa chunks per clock cycle until when the bus request signal is down.

Another alternative could be, in the decode stage, to send to the corresponding operator all the chunks it needs before accepting the next instruction. This approach can save one pipeline stage in the arithmetic operators. The same reasoning can be done for the operator output. Since the implementation of this bus policy alternative requires additional design effort, we chose the previous one for simplicity.

RISC-V interface Some instructions from the coprocessor ISA require communicating with the main RISC-V core. Instructions that require data in input from the main core (e.g., `susr`, `ld*`, `st*`, `mov.x.g`, and `fcvt.*.g`) are feed through the "data in" port from the RoCC interface ④. This input is carried until the execute stage time barrier to be then processed.

Instructions for which their result must go in the main core (e.g., `lusr`, `ld*.next*`, `st*.next*`, `mov.g.x`, `gcmp`, and `fcvt.g*`) rely on the "data out" port ④. The result of these operations is provided either from the decode or the execute coprocessor pipeline stage. For this purpose, a multiplexer ⑨ is instantiated to multiplex results from different pipeline stages. Additional logic is added to handle the case when multiple instructions, from different pipeline stages, output data simultaneously that have to be returned to the main core.

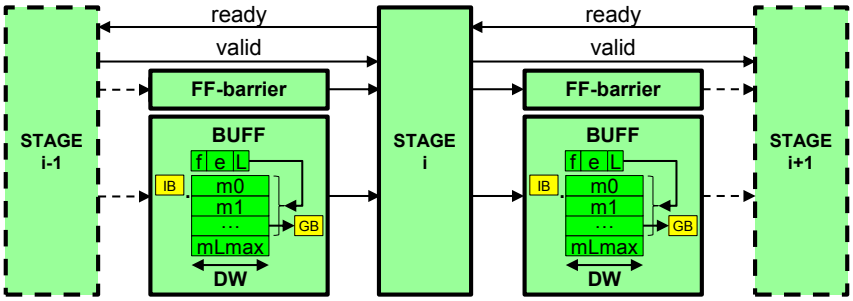


Figure 5.3: The macro pipeline organization in the coprocessor

5.1 Choice of a macro-pipelined architecture

Floating-Point (FP) operators implement algorithms based on a sequence of fixed-point operations on the input exponent and mantissa fields. These operations can be shifts, additions, subtractions, and multiplications. As stated in Section 4.5.3, this work does internal mantissa computations with a data-path bit-width fixed at 64 bits. This bit-width implies that fixed-point operations on mantissa fields are done by iterating on mantissa chunks.

Handling these iterations can be complicated, mainly because the number of mantissa chunks to treat is encoded within the number (in the L field, Section 4.5.3). The most complicated thing is to design the Finite State Machines (FSM), which implement multiple-chunk fixed-point operations, within the FP operator pipeline. This work proposes to pipeline the architecture, by storing the mantissa chunks in internal buffers between each fixed-point mantissa operation, to simplify the design.

The logic between two mantissa buffers, which iterates on mantissa chunks for a variable number of clock cycles, is identified as *macro-stage*. In an architecture pipelined in macro-stages, every mantissa fixed-point operation requires a macro-stage. The number of macro-stages in an FP operator pipeline is equal to the number of mantissa operations needed. The clock cycles latency of a macro-stage, and the unit throughput, are proportional and inversely proportional to the number of input mantissa chunks.

Figure 5.3 depicts the general scheme for each pipeline macro-stage (STAGE i). Each macro-stage does an essential operation on mantissa fields (e.g., move, add, shift, leading-zero-count). Each macro-stage is synchronized with other macro-stages through a ready-valid protocol, and the external input-output synchronization barriers delimit it.

Synchronization barriers are made of a buffer (BUFF) and a Flip-Flop-barrier (FF-barrier). FF-barriers host all the input-output information that is not part of a gnumber (e.g., the shift amount for the fixed-point shift operation). Each buffer hosts a gnumber made of a header, a memory to store the mantissa, integer bits (IB), and guard bits (GB).

The header contains all the gnumber fields except the mantissa. In particular, it hosts the length field (L) that contains the number of used chunks (out of the maximum allowed one) to encode the mantissa. The size of the memory to host the mantissa is the maximum one supported by the gbound Register File (gRF). IB and GB are two additional fields that add extra bits to the mantissa to support correct rounding within the FP operator. In particular, GB extends the last mantissa chunk (pointed by L).

A piece of additional information required by each macro-stage is the *maximum output precision* of the result. This information is used to implement fused operations and exploit the tradeoff between latency and precision (WGP, Section 4.5.4).

With this architecture, the algorithms of FP operators are implemented, in the coprocessor, through macro-pipelines. The complexity to do operations on multiple mantissa chunks is pushed inside each pipeline macro stage, which hosts a generic multiple-precision fixed-point operator. This approach minimizes all the arithmetic units' debug effort, and it maximizes the

component reuse. Since the coprocessor operators support interval arithmetic, and interval endpoints run in parallel macro-pipelines, synchronization steps must be added to realign the interval computation in the pipeline.

Contrary to the micro-architecture of Schulte [49], the architecture presented in this work is pipelined and allows to execute multiple operations at the same time. However, it has some drawbacks addressed in Section 5.2.

5.2 Micro-architecture: drawbacks and solutions

The presented micro-architecture design paradigm is based on macro-stage pipelines. It may not be the best architecture to implement Variable-Precision (VP) computing in hardware: it minimizes the design effort but has several drawbacks that are easy to understand. The first one impacts the operator clock-cycles latency: for simple operations (e.g., shift amount computation), a full macro-stage is needed[2]. Every macro-stage increases the clock cycle latency of the FP operator, and circuits' required are, due to the additional intermediate buffer inserted in the macro-pipeline.

For the second drawback, the latency of the macro-stage pipelines can not be equally distributed among macro-stages because their latency is different. This inequality is due to some operations, like mantissa multiplication, for which their execution requires more clock cycles than those required in other operations, like mantissa shift.

The third and last drawback concerns multiple-chunk computation. When multiple-chunks are involved in computation, only one stage over two is working. The others wait to receive the input or to have the buffer in output free to be used.

Several solutions can be applied to overcome the issues mentioned above. For example, a way to overcome the third drawback is to do double buffering. This solution requires the doubling of the logic required for buffers. For this reason, it was not considered.

Several architectural modifications can reduce the latency and augment the throughput of arithmetic operators. For example, the system throughput can be improved by modifying the pipeline macro-stage operators so that the mantissa chunks are elaborated in the same order they are read. In this way, the throughput is maximized, and only one 64-bit register between stages is needed. However, there are cases where the full mantissa in the output of an operator is needed to start the next mantissa computation. For example, during a FP addition, the mantissa's normalization step requires in input to know if the mantissa provided in output from the addition step overflowed. This information belongs to the most significant chunk of the output mantissa. This dependency of the mantissa chunks brings to have a mixed pipelined and macro-pipelined architecture where each macro-stage is pipelined internally with a dedicated control unit to handle all the internal pipeline stages.

A way to reduce the pipeline latency is to exploit the fact that some macro-stages of the pipeline have positive time slack (the available time in a given data path comparing to the one available at the circuit frequency). There are some macro-stages for which their logic could be enlarged while the system frequency stays unvaried. A way to enlarge the macro-stage units improving the system latency is to enlarge their internal data-path bit-width (e.g., from 64 to 128 bits or more). Enlarging this bit-width reduces the number of clock cycles to provide data in output while the system frequency keeps the same value.

Alternatively, some macro-stages can be merged to reduce the macro-pipeline depth. For example, the shift amount and the shift macro-stages can be merged in a single macro-stage.

[2] A FP operation implements an algorithm based on the input FP fields. It is made of several steps. In this work, every step that involves a mantissa operation requires a dedicated macro stage. For instance, in a FP addition, the input mantissas must be aligned on a common exponent value. This alignment is performed by shifting an endpoint mantissa. The shift hardware operator requires to have the shift amount at the beginning of the operation. Thus, an additional macro stage to compute the shift amount is needed.

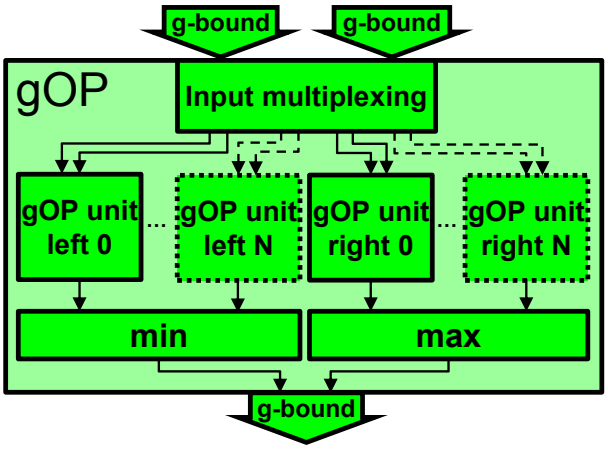


Figure 5.4: The general architecture of a gbound operator

Merging macro-stages reduces the macro-pipeline depth while reducing the amount of instantiated buffers.

In this work, these optimizations are not implemented due to time-related constraints. The next sections focus on the implementation details of all the arithmetic operators in the coprocessor execute stage and all their sub-components.

5.3 Variable-precision architecture for arithmetic operators

This section presents the architecture of all the operators in the coprocessor execute stage. As already mentioned, like the coprocessor Register File (RF), all the operators support Interval Arithmetic (IA) by computing in parallel pipelines the resulting interval endpoints. In this way, there is no difference in latency between computing scalars or intervals.

Figure 5.4 depicts the general macro-architecture for gbound OPerators (gOP). The endpoints of the two input gbounds are used to perform the proper endpoint operations. For some gOP unit (e.g., for the multiplier), it is not possible to look at the input g-bounds and know in advance what are the input g-bound endpoints that generate the outermost output endpoints. Thus, the possible output endpoints are computed, and only the outermost ones are provided in output through maximum and minimum computing units. These two units select the outermost values as endpoints for the output gbound. As it is shown in Section 5.3.3, the number of possible output endpoints, to be compared in the maximum and minimum computing units, can be minimized depending on the input signs of the input endpoints.

Before providing the final gbound in output, the operator handles the Not a Number [8] (NaN) exception. The next sections explain some operator-dependent optimizations on this architecture. The precision and the rounding mode used during the rounding step are programmed in the WGP and RND internal status registers, respectively (Section 4.5.4). Results are rounded with a precision programmable with a granularity of 64 bits (WGP*64) and with programmable rounding mode (RND). In interval computation, the information regarding the inclusion or not of the interval endpoint is carried among the coprocessor pipeline with a dedicated flag. In scalar computation, only one computing unit is used, and all the values are treated as exact values.

The endpoint computations in gOP units follow a classical floating-point scheme [8]: First, the input mantissa fields are aligned to a standard format. If needed, the operation (+,-,*,/) is computed on the aligned operands. Finally, the result is normalized and rounded, and the input exceptions [2] are handled.

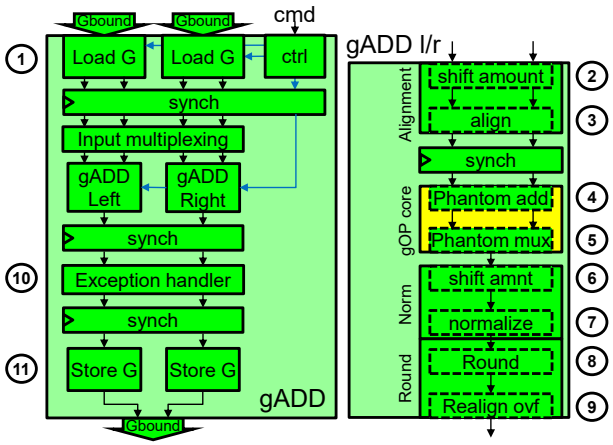


Figure 5.5: The architecture of the gbound adder operator

The next subsections describe the gbound operators' algorithms. The symbols used are: for a gOP unit, $G1$ and $G2$ for the two input gbounds, and Z for the output gbound. For a gbound A, $\underline{A}$ and $\overline{A}$ note its lower and upper endpoints. It is assumed that $\underline{A}$ is lower or equal to $\overline{A}$, and, to simplify the design, all the maximum and minimum computing units are designed under this assumption. For a real x, $\bigtriangledown(x)$ and $\bigtriangleup(x)$ note the rounding of x to the gformat in the $-\infty$ and $+\infty$ directions, respectively.

5.3.1 Move operator

The move operator (gMOV) is the simplest operator in the coprocessor execute stage. It supports the ISA instructions presented in Section 4.7.5. The gMOV operator has a three-stage pipeline.

The first and the last stages, load and store data from and to the coprocessor RF, respectively. These stages are instantiated in all the operators since they are needed to drive the input and output bus arbiters in the coprocessor execute stage. The second stage is in charge of implementing the assembly instructions mentioned above. It copies an interval in input in the result. With some internal multiplexing, it can also copy (left and right) interval endpoints from the input to the output and select some specific data chunks.

The latency of this unit is 3*(L+1) clock cycles, where L is the maximum number of mantissa chunks that the input interval is made.

5.3.2 Adder operator

The gbound signed adder (gADD) is the first arithmetic operator presented in this section. It supports the `gadd`, `gsub`, `gguess`, and `gradius` ISA instructions presented in Section 4.7.6. In the case of signed addition (`gadd`), if the RND status register (Section 4.5.4) is set to round to interval (RTI), the operation implemented by the operator is the following.

$$Z = \begin{cases} \underline{Z} = \bigtriangledown\left(\underline{G1} + \underline{G2}\right) \\ \overline{Z} = \bigtriangleup\left(\overline{G1} + \overline{G2}\right) \end{cases}$$

For the gADD operator, the input multiplexing unit, the minimum, and maximum endpoint selection units are not needed. This because the gbound input endpoints are connected directly to the two endpoint computation units, and their output corresponds to the final interval endpoints.

Figure 5.5 depicts in detail the adder's internal micro-architecture. It is based on 11 macro-stages, and it follows conventional Floating-Point (FP) algorithms for FP addition and FP subtraction [8]. Like the gMOV operator, the first ① and last ⑪ pipeline macro-stages move the data from and to the coprocessor RF. Inside each endpoint computation unit (gADD left or right), the resulting endpoint (left or right) is computed.

The algorithm chosen to sum the two gnumbers is the one shown in [8]. The two input mantissa fields are aligned to the same exponent value (stages ② and ③). Then, the aligned mantissa fields are summed (or subtracted, depending on the input signs) together (stages ④ and ⑤). The final result is normalized (stages ⑥ and ⑦) and rounded (stages ⑧ and ⑨). The last step before storing the computed results in the coprocessor RF is to handle the input exception values (stage ⑩).

Inside the macro-stage pipeline, the intermediate data have some additional integer and guard bits. These bits propagate the eventual overflow or underflow inside the pipeline to have enough bit for an exact rounding. In the case when the RND status register (Section 4.5.4) is not set to RTI, only one of the two parallel adder paths is used.

The latency of this unit is (7*(L+1))+(4*(WGP+1) clock cycles, where L is the maximum number of mantissa chunks that the input interval is made, and WGP is the maximum number of mantissa chunks stored in the WGP status register. The first term is the latency needed by the firsts seven macro-stages. The second one is the latency of the last four macro stages. As it is possible to see, the WGP status register helps to control the precision-latency tradeoff.

The addition core In FP, the resulting mantissa must be positive, since the sign belongs to a dedicated bit. During a signed addition, the difference between the first and the second mantissa may be negative. In this case, the addition algorithm requires a two's complement of the mantissa and a negation of the output sign. The *phantom bit* technique can avoid the two's complement, which requires two serial additions in the addition core.

This technique takes two fixed-point operands, `op1` and `op2`, and computes two operations in parallel, `op1+not(op2)+1` and `not(op1+not(op2))`. If the first one overflows (the phantom bit is set), the second one is output. As shown in Equation 5.1, the first operation computes `op1-op2`, while the second operation computes `op2-op1`. This technique can parallelize the two serial additions mentioned before keeping the same system silicon area.

$$\begin{aligned} op1 + \overline{op2} + 1 &= op1 - op2 \\ \overline{op1 + \overline{op2}} &= \overline{(op1 - op2 - 1)} = (-op1 + op2 + 1) - 1 \\ &= op2 - op1 \end{aligned} \tag{5.1}$$

The presented hardware architecture implements the phantom bit technique in two macro-stages. The first one ④ makes the two parallel additions, and the second one ⑤ checks the overflow bit and selects one of the two operations. The separation of these operations in two macro-stages simplifies the architecture design. With some engineering work, these two macro-stages can be merged in a single one.

Preconditioning of macro-stages To simplify the hardware design, multiple-chunk operators have all the inputs at enable time. Thus, some multiple-chunk operators may require inputs that depend on the actual exponent or the actual mantissa. For example, multiple-chunk shift operations (stages ③ and ⑦) require the shift amount and the shift direction (left or right) before their execution. This order requires the insertion of additional macro-stages before shift operators for the shift amount computation. For instance, stages ② and ⑥ compute the shift amount from the two input exponent values and the leading-zero-count on the mantissa.

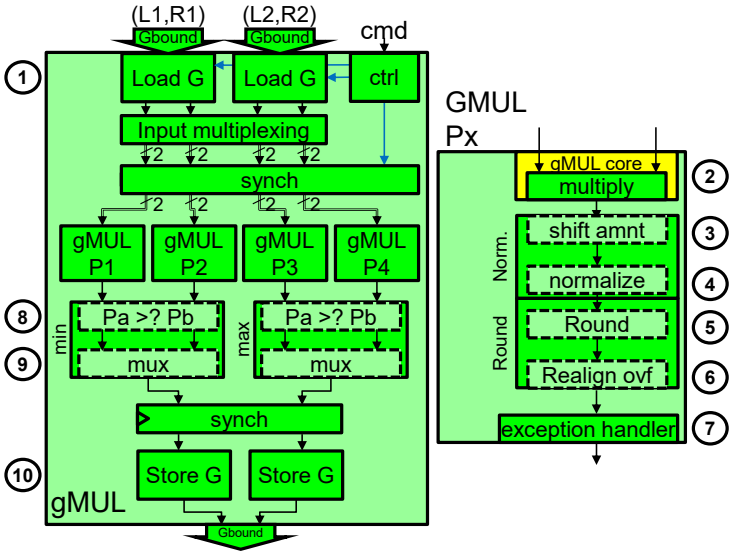


Figure 5.6: The architecture of the gbound multiplier operator

	Input characteristic				Minimum value
	x_l	x_h	y_l	y_h	
ⓐ	$\geqslant 0$	-	$\geqslant 0$	-	$x_l \times y_l$
ⓑ	<0	-	-	$\geqslant 0$	$-((-x_l) \times y_h)$
ⓒ	-	$\geqslant 0$	<0	-	$-(x_h \times (-y_l))$
ⓓ	-	<0	-	<0	$(-x_h) \times (-y_h)$

	Input characteristic				Maximum value
	x_l	x_h	y_l	y_h	
ⓔ	<0	-	<0	-	$(-x_l) \times (-y_l)$
ⓕ	$\geqslant 0$	-	-	<0	$-(x_l \times (-y_h))$
ⓖ	-	<0	$\geqslant 0$	-	$-((-x_h) \times y_l)$
ⓗ	-	$\geqslant 0$	-	$\geqslant 0$	$x_h \times y_h$

Table 5.1: Possible minimum and maximum values for interval multiplications

Normalization and rounding macro-stages The normalization and rounding steps are common to all the Floating-Point (FP) arithmetic operators. At the end of a fixed-point operation (e.g., addition or multiplication), either the resulting mantissa may not be normalized or have more bits than the available one in the output format. These two problems are avoided by normalizing (stages ⑥ and ⑦) and rounding (stages ⑧ and ⑨) the resulting mantissa.

The normalization realigns the mantissa in such a way that it is greater or equal than 1 and less than 2. In the architecture presented in this work uses a special flag of the summary bits to encode the value 0 (Section 4.5.3). The normalization is defined in two macro-stages. The first one ⑥ computes the shift amount of the mantissa (and the shift direction) through a leading-zero-count operation on the mantissa. The second one ⑦ shifts the mantissa, realigns the exponent value, and generates the overflow and underflow (depending on the WGP status register, Section 4.5.4) internal flags.

The rounding takes the outputs of the normalization steps, and, according to the rounding rule chosen, it rounds the output mantissa to be compliant with the register file format. The rounding is defined in two macro-stages as well. The first one ⑧ rounds the normalized mantissa according to the selected rounding rule (in the RND status register, Section 4.5.4) and the generated internal flags. The second step ⑨ shifts right the rounded mantissa in the case of overflow due to the rounding. This step requires to increase the exponent value and eventually detect the saturation to infinity (positive or negative).

5.3.3 Multiplier operator

The gbound multiplier (gMUL) is the second arithmetic operator instantiated in the coprocessor's execute stage. It computes the multiplication of the two gbounds provided in the input. Like for the adder, the RND coprocessor status register hosts the rounding mode used for the operation.

$$\begin{cases} \underline{Z} = \min\left(\bigtriangledown(\underline{G1}\underline{G2}), \bigtriangledown(\underline{G1}\overline{G2}), \bigtriangledown(\overline{G1}\underline{G2}), \bigtriangledown(\overline{G1}\overline{G2})\right) \\ \overline{Z} = \max\left(\bigtriangleup(\underline{G1}\underline{G2}), \bigtriangleup(\underline{G1}\overline{G2}), \bigtriangleup(\overline{G1}\underline{G2}), \bigtriangleup(\overline{G1}\overline{G2})\right) \end{cases}$$

Table 5.1 shows the possible endpoint multiplications for the left and right output endpoints. Looking only at the signs of the operands of these endpoint multiplications, and considering that x_h and y_h must be greater than x_l and y_l, respectively, it is possible to eliminate two inputs (out of four) for both left and right endpoints. For the minimum operator (left endpoint) only ⓑ and ⓒ can coexist at the same time. The maximum operator (right endpoint) only ⓔ and ⓗ can coexist at the same time.

Consequently, with some multiplexing of the inputs' endpoints, it is enough to instantiate four endpoint multiplication units: two for both the left and right endpoint computation paths. Moreover, out of sixteen input endpoint sign combinations, only nine are respecting valid intervals. Out of these nine sign combinations, only one (where the two intervals contain zero) requires two multiplications per endpoint. All the other sign combinations require only one multiplication per endpoint. This work exploits these properties to reduce the average power consumption by minimizing the number of parallel multiplications dynamically at the same time.

Figure 5.1 depicts in detail the internal micro-architecture of the multiplier. It has a ten macro-stages pipeline, and the algorithm implemented inside it is a classical FP algorithm for FP multiplication [8]. As for the adder, the first ① and the last ⑩ pipeline macro-stage move the data from and to the coprocessor RF. Inside each endpoint computation unit (gMUL P1-P4), the possible values for the left and right result endpoints are computed.

In this case, there is no need to align input mantissa fields: the two operands can be directly multiplied (macro-stage ②). After that, the intermediate result is normalized (stages ③ and ④) and then rounded (stages ⑤ and ⑥). After the input exceptions are handled ⑦ in all the four parallel multipliers, the next macro-stages (⑧ and ⑨) compute the minimum and maximum of the four computed values. A possible improvement of the coprocessor architecture is implementing a bypass around these last two macro-stages when parallel multiplications are not needed. This bypass reduces the multiplication latency for most of the operations.

Like for the gADD operator, inside the macro-pipeline, the intermediate data have some additional integer and guard bits. Furthermore, the gMUL operator adopts the same unit for normalizing and rounding the intermediate results as for the gADD operator.

The minimum and maximum operators are implemented in two separated macro-stages to minimize the design time. The first one ⑧ subtracts the two possible values, and, according to the sign of this difference, the second macro-stage ⑨ select the outermost one. In the case when the RND status register (Section 4.5.4) is not set to RTI, only one of the four parallel multiplier paths is used.

The latency of this unit is (L+1)+((L+1)^2+1)+(2*2*(L+1))+(6*(WGP+1)) clock cycles. L and WGP have the same meaning as the ones for the adder latency. The first term is the latency needed by the first macro stage. The second term is the latency needed by the multiple-chunks mantissa multiplier. Since the multiplier iterates on mantissa chunks of 64-bit each, the number of multiplications required is proportional to L^2. The third term is the latency needed by the result normalizer that has to treat the chunks output from the multiplier. The remaining term is the latency needed by the remaining six macro-stages.

5.3.4 Comparator operator

On intervals, the comparison $A < B$ may be true (if $\overline{A} < \underline{B}$), false (if $\underline{A} \geqslant \overline{B}$), or "I can't say" (in the other cases)[3]. To allow for that, the gbound comparison unit (gCMP) has to compute the six following comparison flags [2]: Lower Than (LT), Greater Than (GT), EQual (EQ), Nowhere EQual (NEQ), Not Nowhere EQual (NNEQ). To do that, it compares each gbound input endpoint with the two endpoints of the other input. Depending on how the two input

[3] As a side note, all the programming languages must be modified to support this third option to support UNUM. However, this is out of the scope of this paper.

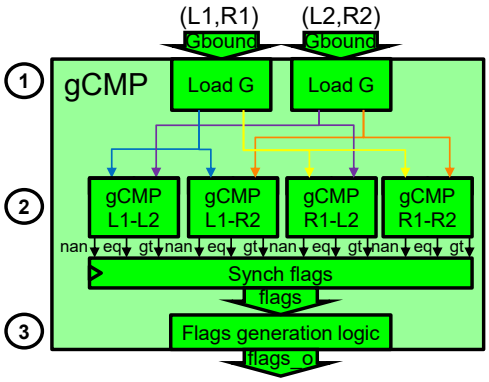


Figure 5.7: The architecture of the gbound comparator operator

gbounds intersect (or not) each other, the four output flags are set or unset, taking care of the distinguished values (e.g., NaN).

Figure 5.7 depicts the hardware implementation of this function. It is based on three macro-stages. Like the operators mentioned above, the first stage ① loads data from the coprocessor RF. The second macro-stage ② consists of four parallel comparator operators (gCMP x1-y2). Each of them is made of two parallel units that check whether the first operand is greater than the second one or whether the first operand is equal to the second one. The first check is made by performing a difference between the two inputs. The second check is made by comparing the two inputs chunk by chunk. The four parallel comparators provide twelve output flags (greater than, equal, or not-a-number).

These flags are used in the last macro-stage ③ to drive a combinational logic that generates the six output flags mentioned above. Depending on the input instruction, the gCMP unit can output only one flag for all the possible endpoint-to-endpoint comparisons.

The latency of this unit is (2*(L+1))+1 clock cycles. L and WGP have the same meaning as the ones for the latency of previous operators. The first term is the latency needed by the first two macro stages. The last clock cycle is the latency needed by the last macro stage that needs only one clock cycle to propagate the output flags in the next pipeline stage.

5.3.5 Conversion operator

The conversion operator (gCNVT) implements all the ISA conversion instructions presented in Section 4.7.7. It has been introduced in the architecture for debug purposes (to print variables values on the terminal) and to facilitate the integration of the presented coprocessor in a conventional FPU. The gCNVT operator is a simple unit divided into two parallel paths.

The first path takes care of the IEEE-754 to gbound conversions instructions. It is made of two macro-stages: one to convert from IEEE-754 to gbound format, and one to store data in the coprocessor register file. The second path takes care of the gbound to IEEE-754 conversions instructions. This path is made of two macro-stages: one to load data from the coprocessor register file, and one to convert from gbound to IEEE-754 format.

The latency of this unit is two clock cycles. For both conversion operations, they need one clock cycle to make the conversion and one clock cycle to communicate with the gRF.

Both conversion operations do the conversion within one clock cycle latency since they have to move only one mantissa chunk (the largest IEEE-754 format supported is the 64-bit one). IEEE-754 input numbers set the exact summary bit in the output gbound (Section 4.5.3). The RND coprocessor status register specifies the rounding mode used in the gbound-to-IEEE-754 conversion operations (Section 4.5.4).

If the RND is set to round-to-nearest (half away from zero, RTN), depending if the target format is half-float, float, or double, the result is rounded looking the 11th, the 24th, or the 53rd

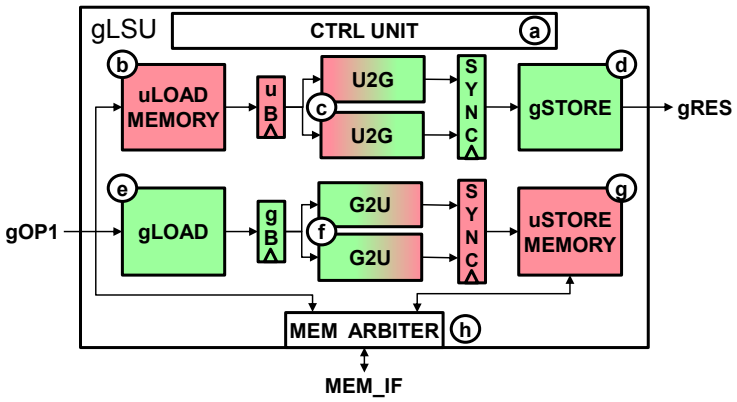


Figure 5.8: The architecture of the coprocessor load and store unit

mantissa bit. If the target bit is one, the output mantissa is rounded up. Otherwise, the output mantissa is truncated. If the RND is set to round-up (RDU), the output mantissa is rounded up if the output sign is positive, and the part of the mantissa that would be truncated in the output format is different from zero. The output mantissa is truncated in other cases. If the RND is set to round-down (RDD), the output mantissa is rounded up if the output sign is negative, and the part of the mantissa that would be truncated in the output format is different from zero. The output mantissa is truncated in other cases.

For RTN, there is no necessity to look at the values of the truncated bits. For RDU and RDD, all the truncated bits have to be taken into account. The rounder unit receives additional signals to speed up this test. Every signal carries the information if a given chunk is equal to all zero or equal to all 1. These signals avoid spending extra clock cycles in checking the truncated mantissa chunks.

5.3.6 Load and store unit

The coprocessor gbound Load and Store Unit (gLSU) is in charge of handling all the coprocessor load and store operations in memory specified in the coprocessor ISA (Section 4.7.4). In this work, the gLSU handles the format conversions between the gbound format of the coprocessor register file (Section 4.5.3) and the UNUM and ubound memory formats described in Section 4.3. The DUE, SUE, and MBB coprocessor status registers encode the configuration of the format used to load and store data in memory (Section 4.5.4).

The DUE and SUE status registers have two internal fields (ESS and FSS) that describe the maximum exponent dynamics and precision that a UNUM (or a ubound endpoint) can have. The MBB value encodes the length, in bytes, of numbers in memory.

Figure 5.8 depicts the gLSU hardware architecture. The internal control unit ⓐ drives the load and the store units according to the input load or store instruction. The gLSU has two parallel pipeline paths inside: the load path and the store path.

The *load path* (units ⓑ, ⓒ, and ⓓ) handles all the load memory instructions. The *store path* (units ⓔ, ⓕ, and ⓖ) handles all the store memory instructions. Both paths take care of moving the data between the memory and the coprocessor register file, and it takes care of the conversion between their two Floating-Point (FP) formats. A memory arbiter ⓗ ensures the correct execution order between load and store instructions. The next subsections describe in detail the units mentioned above.

This unit subdivision was done in order to support in the future additional Variable-Precision (VP) FP formats. To support additional FP formats is enough to add parallel format

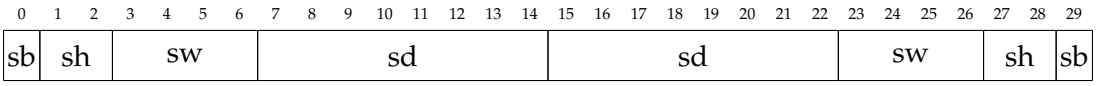


Figure 5.9: Worst case scenario of storing a 30 Bytes data

conversion functions in the load and store paths. In this way, the user can make computation using different memory formats, with different precision configurations, by leveraging the shared gbound coprocessor register file.

Memory data alignment As already mentioned in Section 4.3, unlike conventional systems, data in memory is aligned on bytes, and they can be expressed on an arbitrary number of bytes. The MBB status register must contain the length, expressed in bytes, of the data to be loaded or stored by the gLSU. As specified in Section 4.7.4, the gLSU must be able to load and store data with any MBB length on every address. Thus, the gLSU must be able to deal with misaligned addresses.

The RISC-V L1 cache of Rocketchip supports only aligned memory operations. To overcome this limitation, the coprocessor LSU splits memory operations with misaligned addresses into several aligned memory operations. For example, as depicted in Figure 5.9, to store in memory 30 bytes, the gLSU may do on the L1 cache (in the worst case) an 8-bit store, a 16-bit store, a 32-bits store, two consecutive 64-bit stores, one 32-bit store, one 16-bit store and one 8-bit store. The number of required memory operations can be minimized if the L1 cache has an interface with byte enables.

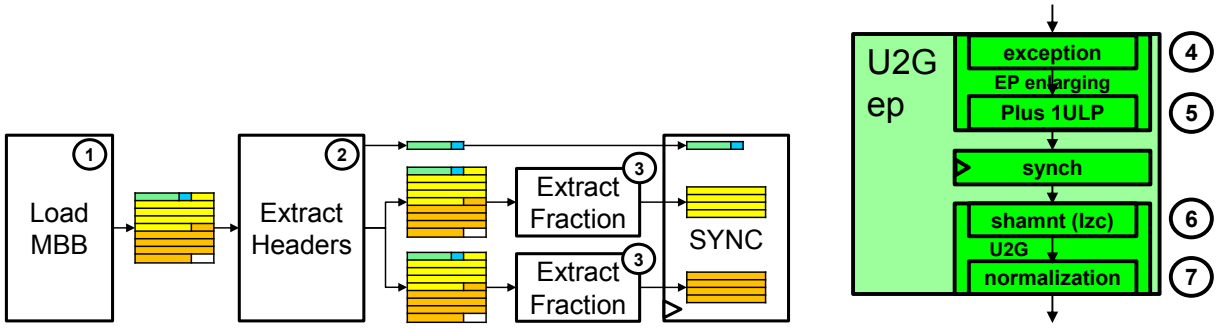
The load path

The gLSU load path loads UNUMs or ubounds from the main memory, it converts them in the gbound format, and it stores the converted result in the coprocessor register file. The load path is divided into three main parts. The first one ⓑ loads a UNUM or a ubound from main memory, in the formats defined in Section 4.2.1, and it extracts all their subfields. The second part ⓒ is made of two parallel units (to support both UNUM and ubound) that convert UNUMs into gnumbers. Finally, like for the previous arithmetic operators, the last part ⓓ stores the converted data into the coprocessor register file. The gLSU load path macro-pipeline is made of eight macro-stages.

The latency of this path is (8*(ceil(MBB*8/64)))+ϵ+γ clock cycles. The first term is the latency of the eight macro-stages of the load pipeline. The latency is a function of the length of the data in memory. The ϵ term is the additional clock cycle latency due to the additional memory requests if the data is not aligned in memory, while γ is the one due to the cache latency.

Figure 5.10a depicts the first part of the gLSU load path, the UNUM load (uLOAD) unit (ⓑ in Figure 5.8), which contains the firsts three macro-stages of the gLSU load path pipeline. The first macro-stage ① loads MBB bytes from memory, and it stores them into a buffer. Depending on MBB and the address alignment, it may generate multiple load operations in order to load the MBB bytes from memory. It supports misaligned addresses. The following macro-stage ② extracts the subfields of the UNUM (or ubound) header (`s`, `u`, `es-1`, `fs-1`, and `e`) and compute the shift amount needed to extract the mantissa field. The last macro-stage ③ extracts the fraction fields by shifting them to the left according to the precomputed shift amount. After this stage, all the UNUM (or ubound) fields are well separated, and they are ready to be converted into the coprocessor register file format.

Figure 5.10b depicts the second part of the gLSU load path, the UNUM to gbound endpoint (U2G ep) conversion unit (one of the two units labeled with ⓒ in Figure 5.8). It is made of 4 macro-stages. The first macro-stage ④ triggers possible distinguished value encodings (e.g.,



(a) UNUM load (uLOAD) unit: it loads UNUMs and ubounds from the main memory and extract all their fields

(b) UNUM to gbound endpoint (U2G ep) conversion unit: it converts a UNUM into gbound

Figure 5.10: Load path units of the coprocessor load and store unit

NaN, infinity) and decode the exponent from the UNUM encoding to two's complement one, like the one of a gnumber. If the UNUM ubit is set, and if the RND status register is set to RTI (Section 4.5.4), the second macro-stage ⑤ increments by one ULP the mantissa, to the outermost value of the corresponding UNUM with the ubit set.

Since gnumbers do not support subnormal representations, the third macro-stage ⑥ computes, through a leading-zero-count operation on the mantissa, the shift amount needed to normalize the mantissa. Finally, ⑦ the mantissa is normalized (as well as the exponent value) with a shift left. The eighth macro-stage stores back the converted gbound in the coprocessor register file (ⓓ in Figure 5.8). After this, the loaded gbounds are ready to be input to the gbound operators (gOP) detailed above.

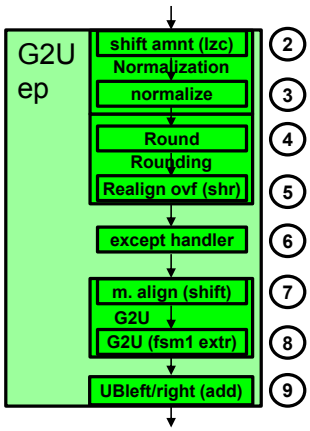
The store path

The gLSU store path stores gbounds from the coprocessor register file in memory in the UNUM (or ubound) format. The store path (Figure 5.8) is divided into three main parts. Like the previous arithmetic operators, the first part ⓔ loads a gnumber (or a gbound) from the coprocessor register file. The second part ⓕ converts a gnumber (or a gbound) into a UNUM (or ubound). It is made of two parallel units to support ubound. The last one ⓖ converts the obtained UNUM (or ubound) into the BMF format (Section 4.3). It compacts the converted fields into a single bitstream, for then storing it in main memory. The gLSU store path is pipelined in fifteen macro-stages.

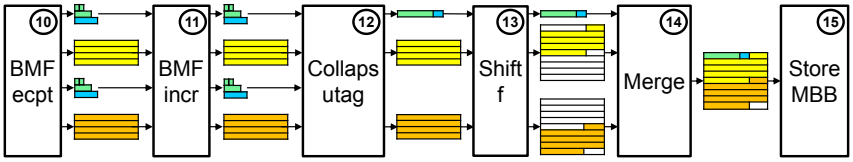
The latency of this path is (3*(L+1))+(7*(ceil(2^{FSS}/64)))+(5*(ceil(MBB*8/64)))+ϵ+γ clock cycles. The first term is the latency of the firsts three stages of the store pipeline. The latency is a function of the length of the data to be stored. The second term is a function of the selected UNUM Environment, in particular FSS. The third element is a function of the length of the format to be stored, MBB. The last terms, ϵ and γ, have the same meaning as for the load path.

The first macro-stage belongs to the unit fetches the gnumber (or gbound) that has to be stored from the coprocessor register file. The following eight macro-stages belong to the G2U units depicted in Figure 5.11a (ⓔ in Figure 5.8). It converts a gnumber (or gbound) into a standard UNUM (or ubound).

The first step of this conversion rounds the input gnumber delimiting its exponent range according to the chosen ESS value, and its fraction size according to the FSS set. The datatype status register (DUE or SUE) stores the ESS and FSS values needed by the used memory instruction (Section 4.7.4). As for previous operators, this rounding requires four pipeline macro-stages (stages ②-⑤). After the rounding, the following macro-stage ⑥ handles all the distinguished values (e.g., infinity and NaN) that may occur after rounding.



(a) gbound to UNUM endpoint (U2G ep) conversion unit: it converts a gbound into a UNUM



(b) UNUM store (uSTORE) unit: it stores gbounds in main memory in the UNUM and ubound formats

Figure 5.11: Store path units of the coprocessor load and store unit

Subnormal	**Normal**
`es-1`=$\log_2$(2-EXP) `e`=0	`es-1`= $\begin{cases} \lceil \log_2(\text{2-EXP}) \rceil & \text{, if EXP<2 (EXP}\leqslant 1) \\ \lceil \log_2(\text{EXP}) \rceil & \text{, if EXP>0 (EXP}\geqslant 1) \end{cases}$ `e`=2^{es-1}+EXP-1
Required conditions	
• EXP<2 (EXP<=1) • 2-EXP=2^i, i is integer $\geqslant 0$	None

Table 5.2: The equations used, and their required input conditions, to convert a two's complement exponent number (EXP) into the `e` and `es-1` UNUM exponent encoding

The next two macro-stages convert the rounded gnumber in the UNUM format (Section 3.1.3). The first step of this conversion ⑦ tries to sub-normalize the mantissa (to maximize the number precision), and it converts the gnumber exponent in the `e` and `es-1` UNUM fields. Table 5.2 shows the equations (and the required input conditions) used to convert the two's complement exponent of the input gnumber (EXP) in the UNUM format representation. In the UNUM format all the exponent values can be represented in the normal format (`e` > 0). The subnormal formats have some restrictions since the `e` field must be zero. Between normal and subnormal representations, the presented unit chooses subnormal to maximize the mantissa precision.

The second step of the G2U unit ⑧ computes the value of the `fs-1` UNUM field, minimizing the mantissa length according to its trailing zeros. This step cannot be embedded in the previous macro-stage because the shift of the mantissa must finish before computing the trailing zero count. The last macro-stage of the G2U unit ⑨ decrements of one ULP the obtained mantissa when the number is not exact in order to be compliant with the ubit direction (see `ubleft` and `ubright` operators in [2]).

Once the input gnumber (or gbound) is converted into UNUM (or ubound), the converted data is stored in memory through the store unit (ⓖ in Figure 5.8) This unit, depicted in Figure 5.11b, is made of 6 macro-stages. The first two stages (⑩ and ⑪) convert the obtained UNUM (or ubound) in the BMF format (Section 4.3).

These two stages implement a re-round of the converted UNUM (or ubound) guaranteeing that the bit-length of converted number fits in the byte budged specified in the MBB coprocessor status register (Section 4.5.4). These units principally round the UNUM mantissa to have a bit-length so that the global UNUM footprint does not overpass MBB bytes. This double rounding is needed since the final mantissa length depends on the lengths of the `e`, `es-1`, and `fs-1` UNUM fields, depending on the overflow that may occur in the UNUM conversion unit ⓕ.

The first BMF step ⑩ computes the maximum length of the final mantissa and detects if the input number has some distinguished values (e.g., infinity, NaN) or if the output will be a distinguished value. The maximum mantissa length is used in the second BMF step ⑪. It re-rounds the mantissa according to the computed maximum length and updates the UNUM fields according to the rules specified in Section 4.3.

The next three macro-stages (⑫, ⑬, and ⑭) format the UNUM (or ubound) in the output of the BMF units in the chosen format (Section 4.2.1). The first of these three macro-stages ⑫ starts the output format configuration. It collapses the fixed-length UNUM fields (`s`, `u`, `es-1`, and `fs-1`), the exponent ones (`e`), and it computes the mantissa shift amount according to the chosen memory format (Section 4.2.1).

The next macro-stage ⑬ shifts the left and right mantissa fields (in the case of an ubound store) separately. The last macro-stage ⑭ merges all the precomputed fields in a single bitstream. At this point, the UNUM (or ubound) number fits the MBB boundary, and it is ready to be stored in the main memory ⑮.

Memory consistency hardware mechanisms

Like in all the units which load and store data in the main memory, the coprocessor storage unit has to face the classical memory consistency issues. For instance, these issues appear when data that not yet stored are loaded from memory. To avoid these issues, this work leverages the memory arbiter (MEM ARBITER, ⓗ in Figure 5.8) of the coprocessor load and store unit. The main issue that the memory arbiter has to consider is that both the load and store macro-pipeline have a not-negligible depth and variable latency.

The memory arbiter adopts a simple memory protocol. It allows pipelined sequences of either load operations or store operations. If a load operation arrives after a store (or vice versa),

① Unit	② stg	Area (μm^2) ③	Area (μm^2 buff) ④	Total Power (mW) ⑤	Total Power (mW buff) ⑥
Rocket_tile	-	1553 (100%)	n.a. (n.a.)	95.2 (100%)	n.a. (n.a.)
–RISC-V	-	23.09 (1.5%)	n.a. (n.a.)	0.78 (0.8%)	n.a. (n.a.)
–64bit_fpu	-	**53.1 (3.4%)**	n.a. (n.a.)	**1.43 (1.5%)**	n.a. (n.a.)
–d_cache	-	487.6 (31.4%)	n.a. (n.a.)	12.72 (13.4%)	n.a. (n.a.)
–i_cache	-	425.6 (27.4%)	n.a. (n.a.)	8.51 (8.9%)	n.a. (n.a.)
–if/periph	-	102.3 (6.6%)	n.a. (n.a.)	3.83 (4.0%)	n.a. (n.a.)
–coproc	3	**461 (29.7%)**	**307 (66.5%)**	**17.0 (17.9%)**	**4.2 (24.7%)**
—s_decode	-	131.1 (8.4%)	130.3 (99.3%)	2.29 (2.4%)	2.16 (94.5%)
——gRF	-	130.8 (8.4%)	130.3 (99.6%)	2.27 (2.4%)	2.16 (95.1%)
—s_execute	-	327.5 (21.1%)	176.5 (53.9%)	13.6 (14.3%)	2.03 (14.9%)
——gMOV	1	4.242 (0.3%)	3.061 (72.2%)	0.17 (0.2%)	0.02 (14.1%)
——gCMP	3	30.73 (2.0%)	24.97 (81.3%)	0.83 (0.9%)	0.25 (29.9%)
——gADD	11	66.77 (4.3%)	43.61 (65.3%)	2.3 (2.4%)	0.42 (18.3%)
——gMUL	10	143.4 (9.2%)	60.06 (41.9%)	6.76 (7.1%)	0.98 (14.4%)
——gLSU	3	81.35 (5.2%)	44.77 (55.0%)	3.5 (3.7%)	0.36 (10.4%)
———LD	3	29.96 (1.9%)	16.95 (56.6%)	1.51 (1.6%)	0.14 (9.3%)
————load	3	15.47 (1.0%)	9.344 (60.4%)	0.86 (0.9%)	0.08 (9.6%)
————u2g	4	10.53 (0.7%)	4.579 (43.5%)	0.56 (0.6%)	0.04 (6.6%)
———ST	3	51.03 (3.3%)	27.81 (54.5%)	1.95 (2.0%)	0.22 (11.5%)
————g2u	8	27.35 (1.8%)	16.96 (62.0%)	1.07 (1.1%)	0.11 (9.8%)
————store	4	17.29 (1.1%)	10.84 (62.7%)	0.6 (0.6%)	0.09 (14.9%)

Table 5.3: Synthesis results of the variable-precision architecture using a 28nm FDSOI library

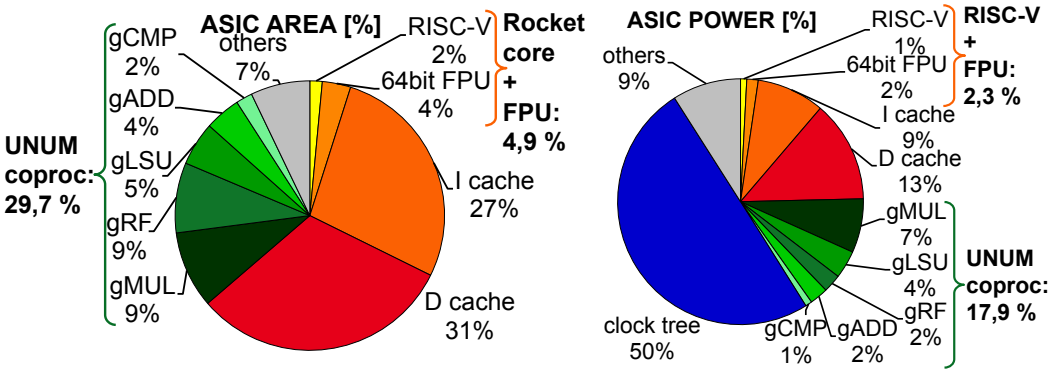
it only starts its execution when the store (or the load) finishes. There are more sophisticated memory consistency protocols in state of the art. However, we did not consider other memory consistency protocols due to time constraints.

5.4 ASIC synthesis results and FPGA integration

This section shows the techniques used to validate the design, its integration in FPGA, and its ASIC synthesis results.

5.4.1 Validation of the units

The architecture design is validated at multiple levels starting from the basic components up to the top unit (Rocket tile). Each subunit is described in VHDL and is validated in simulation (with *Questasim*) against 50 million pseudo-random generated input vectors varying all the input parameters. The output values of the units under test are compared with those generated by executable specifications written in high-level behavioral VHDL code. We used a Xilinx Virtex 7 FPGA as a final validation test. Since an exhaustive validation on FPGA of the whole system needs full compiler support for UNUMs, the final validation is based on calling hand-written instructions to check the correct inputs propagations into the coprocessor. With these validation techniques, no errors or bugs were found during the execution of several benchmarks, including those presented in Chapter 6.



(a) Chip area distribution (b) Chip power distribution

Figure 5.12: Area (5.12a) and power (5.12b) distribution of the synthesis results shown in Table 5.3

5.4.2 FPGA integration

The integration on FPGA is done through a microblaze system generated with Vivado from Xilinx. This system is a wrapper for the architecture mentioned above. It connects the RISC-V system with the DRAM and with a dedicated UART interface, both available on the FPGA board.

The microblaze system is programmable through another UART interface. At boot, the microblaze loads the compiled code to be executed, in the RISC-V system, from an SD card into the DRAM. Once the DRAM contains the executable code for the RISC-V, the microblaze resets the RISC-V module and then enters into sleep mode.

With the reset, the RISC-V module automatically starts to fetch the execution code from the DRAM. A terminal connected to the RISC-V UART interface shows the printed outputs produced from the FPGA emulation. The user can see what is produced inside the RISC-V system during the FPGA emulation through prints in the terminal attached to the UART interface.

The FPGA system can run at 50MHz. This frequency matches the Rocketchip specifications [7].

5.4.3 ASIC integration: synthesis results

The architecture described in this work was synthesized, using Design Compiler from Synopsys, and targeted to the *28nm* FDSOI library from ST-microelectronics (cmos028FDSOI). Register re-timing and clock gating optimizations are enabled during synthesis. For simplicity, all the memory elements are synthesized with flip-flops.

The timing constraint is set at *1ns* (*1GHz*). To take into account connections external to the chip, a latency corresponding to the *70%* of the clock cycle period is added to all the chip inputs-outputs. The debug inputs of the chip are disabled. The 'flatten' option of some subunits is disabled to be able to report area and power estimations after synthesis. The power estimation is performed by assuming a random switching activity on the inputs.

The best clock period achieved is *1.697ns* (*589MHz*). The critical path passes in the Floating-Point (FP) multiplier and adder operator of the RISC-V's FP unit.

Table 5.3 shows the synthesis results of the synthesized system. The *Rocket tile* is the name of the top unit. Column ① hosts the names of the synthesized components and subcomponents units. The tabulations in this column denote the component hierarchy. Column ② indicates the number of pipeline stages-macro-stages for each unit. Subcomponents that are not showed

in this table have 1 stage latency. Column ③ indicates the area of the unit[4] expressed in μm^2. The percentages refer to the area taken by a unit compared to the one at the top (*Rocket_tile*). Column ④ depicts the area taken by mantissa buffers in the macro-pipeline for each unit (Section 5.1). The percentages refer to the area taken by the unit buffers (column ④) compared with the one taken by each unit (column ③). Column ⑤ shows the power consumption required for each unit expressed in mW. Like ③, the percentages refer to the unit power consumption compared with the one of the top. Column ⑥ shows the power consumed by the intermediate buffers of each unit expressed in mW. The percentages refer to the power consumption of the buffers compared with the one in each unit. Figure 5.12a and Figure 5.12b depict the area and power distribution for some subunits.

From an area perspective, the Rocket tile requires $1.5mm^2$ of a silicon surface. The area takes into account the 50% of unused space for the place and route. The main contributors for the area footprint are the units that contain memory: instruction and data caches and the RISC-V coprocessor. Columns ③ and ④ show that the main area footprint contributors for the coprocessor are the units that contain memory: the gbound Register files and the macro-pipeline buffers. These results express the hardware cost of having a pipelined variable-precision coprocessor.

In terms of power consumption, the Rocket tile consumes $95mW$. The clock tree consumes the most significant part of estimated power ($47.5mW$). The chip components consume the rest ($47.6mW$). Like the area, the main contributors for the power consumption are the instruction and data caches and the RISC-V coprocessor. The unit which consumes more is the RISC-V coprocessor with $17mW$. Mantissa buffers consume one-fourth of this power. Logic and standard flip-flops consume the rest. This high energy consumption is mostly due to the high number of pipeline stages of internal operators and by the additional logic of the finite state machines which handle multiple-chunk mantissa computations.

The RISC-V coprocessor is nine times bigger and consumes twelve times more than the RISC-V FP unit. The next chapter evaluates the performance of this unit (in terms of speed but also precision) on linear algebra benchmarks.

[4] The adder (gADD) requires one pipeline macro-stage more than the multiplier (gMUL). However, its area is lower than the one of the multiplier. This difference is because the support of interval arithmetic. In fact, the multiplier hosts four parallel mantissa multipliers while the adder hosts only to mantissa adders.

Chapter 6

Experimentation

The main challenge in modern scientific computing systems is the scaling up of the maximum problem size that the system can compute. In linear systems, scaling up the problem size corresponds to scale up the maximum size of the matrices in memory that the system can compute. Increasing the matrices size implies to prefer indirect (iterative) methods instead of direct ones. Indirect methods iterate to solve a linear system, and, at every iteration, magnify the round-off error for the internal variables. It is necessary to increase the variables precision and memory footprint to compensate for the round-off error of the algorithm results. This challenge has to face the memory wall problem that limits the data throughput and the amount of data possible to store in memory. This challenge requires to scale up the hardware infrastructure of the computing system. As stated in Chapter 2, Variable-Precision (VP) may help achieve this challenge for two reasons.

Firstly, the VP can support high-precision and compact formats in memory. These formats require improving existing hardware architectures in computing precision, allowing them to store high-precision data in memory while reducing applications computational error. This property can scale up the hardware infrastructure, and, in a different use case, it can reduce the computational error of applications. Secondly, the VP can encode numbers with higher precision than conventional formats with fixed size [1]. This higher precision is thanks to using a larger mantissa, with possibly a little bonus if bits are harvested from the exponent around exponent values centered around zero[1]. The questions[2] that we are trying to answer in this chapter are:

- Can VP formats reduce the memory footprint of applications?
- Can VP scale up the maximum problem size that the system can compute?

This chapter evaluates the benefits of the VP Floating-Point (FP) hardware architecture, described in Chapters 4 and 5, by testing it on software benchmarks. This architecture can break the memory wall by supporting compact VP FP data in memory. It differs from conventional FP architectures on three main points.

Firstly, it supports (simultaneously) several VP FP formats configurations for data in memory. Secondly, it supports VP memory operations with misaligned addresses. Finally, it is possible to tune the internal computing precision at run time. This chapter aims to study how these features can improve the convergence of iterative solvers for scientific computing applications

[1] Section 3.1 shows the range of values for which VP formats have higher precision that IEEE-like ones.

[2] We consider that reducing the power is a secondary focus of this work. However, supporting VP FP formats in memory reduces the consumed power of the system. Firstly, having smaller data in memory reduces the power dissipated in the memory subsystem. Secondly, in applications that distribute the computation of large matrices in smaller ones, it is possible to reduce significantly the global power budget required for resolving the original problem. This can be achieved thanks to VP capability of scaling up the maximum problem size that the system is able to compute. This scale up allows to increase the maximum sub-matrices' size, reducing the number of the ones involved in the computation, and then the global power budget.

while reducing the computational error in the final result while minimizing the memory data footprint.

Unlike conventional FP architectures, status registers provide information about the precision of data (introduced in Section 4.5.4). These registers work like "*knobs*" that tune data precision and computation latency. There are two major knobs in this VP architecture:

- The first knob controls the memory format by handling the MBB, DUE, and SUE status registers. They configure the bit-length (with a byte-level accuracy), the format, the maximum mantissa precision, and the exponent magnitude of VP data in memory.
- The second knob controls the FP unit computation precision, through the WGP status register. This register tunes the computation precision with a 64-bit granularity.

The VP load-and-store unit of the VP architecture provides the support for memory operations with misaligned addresses.

We exploit the following benchmarks to test the VP architecture: a Gauss elimination solver, a Conjugate gradient solver, and a Jacobi solver. All these algorithms solve the canonical linear problem `Ax=b`, where `A` is a square matrix, and `x` and `b` are vectors. In scientific computing literature, several algorithms solve the same problem exist.

Depending on the nature of the input matrix, its structure, or its conditioning, some algorithms have better output accuracy or convergence latency than others. In particular, some algorithms do not even converge if the input matrix does not respect some properties. For example, the Gauss elimination, the Conjugate gradient, and the Jacobi algorithms require different input matrix properties. The Gauss elimination algorithm is the least strict among the three: it only requires the input matrix to be invertible. The Conjugate gradient algorithm requires the input matrix to be invertible and positive semi-definite, The Jacobi algorithm is even more restrictive because it requires the input matrix to be invertible and diagonally dominant.

The Gauss solver is a direct algorithm. In other words, the number of operations involved in a run of this algorithm is *fixed* and known a priori. The Conjugate gradient[3] and the Jacobi solvers are iterative algorithms. The number of iterations in these algorithms is *variable* and depends on the target precision threshold given in input. If this threshold is not achieved, an additional iteration on the kernel is performed. The Conjugate gradient and the Jacobi algorithms stop iterating either when the target threshold is met, or when the number of iterations matches a limit, specified in the input.

The next sections benchmark the VP architecture by using the three algorithms mentioned above. Section 6.1 benchmarks the VP architecture by showing how the VP can improve computations for direct algorithms. Here the Gauss algorithm is used. Section 6.2 shows how VP can improve computations in iterative methods, focusing on the Conjugate gradient algorithm. Finally, Section 6.3 shows a performance benchmark of the VP architecture presented above, with the state of the art MPFR software library, on the Gauss, Conjugate gradient, and Jacobi solvers[4].

The last section performs a performance evaluation of the VP architecture using all the three solvers. This experiment uses the MPFR [9] software library to benchmark the VP architecture. The choice is because there are no available hardware architectures in the state of the art for which it is possible to reach similar precisions in memory.

[3] The Conjugate Gradient (CG) was originally proposed by Hestenes [79] as a direct (non-iterative) method in theory and in exact arithmetic. It is supposed to terminate in at most `n` steps, where `n` is the size of the problem. But in practice, due to rounding errors, the CG method can take many more than `n` steps (or fail). Therefore, CG is now used as an iterative method, which converges anyway in reasonable number of steps.

[4] It would be a legit concern to define the "right" criteria for selecting a VP algorithm for each individual case. However, this discussion is beyond the scope of this work. Moreover, we did not find any useful usage of the Interval Arithmetic (IA) support for these algorithms. However, this work demonstrated that supporting IA in hardware is feasible at the costs shown in Section 5.4.

Figure 6.1: Stellar [80] hardware platform (with an embedded Virtex-7 FPGA)

The emulation environment The results of this work are conducted on our full-custom hardware "Stellar" board [80] (Figure 6.1). This board hosts an FPGA chip, a Kintex 7 Ultrascale programmable component of Xilinx. Additional board components used in this work are 16 GB of SDRAM memory, and an SD card as secondary storage.

The FPGA hosts the system described in Chapter 4. The Rocketchip tile is surrounded by some standard hardware units from Xilinx that make the system useable in the FPGA. These units are a UART driver, an SDRAM driver, an SD card driver, and a processor (uBlaze, microblaze) core from Xilinx. The UART driver controls the UART interface used to print the RISC-V terminal during program execution. The SDRAM driver connects the system with the board SDRAM. The SD card driver interfaces the system with the FPGA SD card slot.

The uBlaze core is used at the reset of the board. It loads the code from the SD card, interfaced with the FPGA board, into the system SDRAM. Afterward, it resets the Rocketchip RISC-V, then it goes to sleep. With this system, it is possible to emulate bare-metal applications with a plug-and-play test environment on the FPGA software.

The following experiments were written in assembly, using the instruction set described in Section 4.7, with the collaboration of Tiago Trevisan Jost. This collaboration was made to properly drive the development of his compiler, presented in Section 4.8. This compiler will support custom VP datatype, and it will simplify the coding of software VP FP applications.

6.1 Experiment 1: Variable-precision benefits for direct methods

Algorithm 4 Gauss elimination: general algorithm

1: **function** GAUSS(A, b, N) ▷ $A_{n\times n}$, $b \in \mathbb{R}^n$
2: **for** $i \leftarrow 0$ to $N-2$ **do**
3: **if** $A_{i,i} = 0$ **then** break
4: **for** $j \leftarrow i+1$ to $N-1$ **do**
5: $ratio \leftarrow \frac{A_{j,i}}{A_{i,i}}$
6: **for** $k \leftarrow i$ to $N-1$ **do**
7: $A_{j,k} \leftarrow A_{j,k} - (ratio \cdot A_{i,k})$
8: $b_j \leftarrow b_j - (ratio \cdot b_i)$
9: $x_{n-1} \leftarrow \frac{b_{n-1}}{A_{n-1,n-1}}$
10: **for** $i \leftarrow n-2$ downto 0 **do**
11: $x_i \leftarrow \frac{(b_i - \sum_{j=i+1}^{n-1} A_{i,j} \cdot b_j)}{A_{i,i}}$
return x ▷ $x \in \mathbb{R}^n \mid Ax = b$

The purpose of this experiment is to measure the Gauss Elimination (GE) algorithm on a VP computing environment in terms of execution time and computational error. The GE algorithm, depicted in Algorithm 4, is a linear system solver divided into two main steps: the triangulation of the input matrix, lines 2-8, and the backward propagation of the output vector, lines 9-11. The discrete representation of real numbers may affect the output computational error of this algorithm.

Similarly, in the GE solver, the round-off error during operations impacts the output computational error. There is a linear relationship between the condition number of the input matrix and the output computational error. This experiment uses the Hilbert matrix that is well known for being ill-conditioned, to highlight the capability to reduce the computational error of the VP FP architecture[5]. Dealing with Hilbert matrices as a benchmark is a good hint for estimating if the hardware infrastructure can handle real complex physics problems. The other input variables (i.e., the input vector b) are randomly generated with values uniformly distributed between zero and one. We plan to evaluate our computed results by measuring the resulting residual error (computing $\| Ax - b \|_2$).

The VP version of the GE algorithm is inspired by the one that uses conventional IEEE 754 formats (Appendix A.1 lists the C code for this kernel). The pseudocode of this algorithm is available in Appendix A.2. Its current C-assembly implementation is available in Appendix A.3.

```
6     for(wgp=0; wgp<=(2^MAX_FSS)/64; wgp++){
7         for(mbb=min_mbb(MAX_ESS,MAX_ESS); mbb<=max_mbb(MAX_ESS,MAX_ESS); mbb
      ++)
8             call_gauss(MAX_ESS, MAX_FSS, mbb, wgp, N);
9     }
```

Listing 6.1: Gauss precision settings

The GE VP kernel is repeatedly executed, gradually increasing the format memory footprint for data involved in internal calculations and their computing precision. Latency and computational error are measured varying the memory footprint of VP variables (MBB, from 1 to 68 bytes, Section 4.3) and the computation precision (WGP, from 64 to 512 bits, Section 4.5.4). Several runs of the GE algorithm, with different sets of the MBB and WGP parameters, were performed (lines 6 to 9).

The aims of this experiment are two-fold: first, to understand how precision and error behave varying the variables memory footprint, and second, to characterize the relation between data footprint in memory and computational precision. In this experiment, we expose the results regarding the largest UNUM Environment (UE, Section 3.1.3) supported by the VP architecture: `ess=4` and `fss=9`[6]. This memory format supports up to 16 and 512 bits to represent exponent and mantissa, respectively. The UE is set to the maximum one so we can further ignore it and control the precision in memory with MBB and inside the coprocessor with WGP[7].

The GE algorithm follows a direct method. Consequently, to get all the precision benefits from VP computing, all the internal variables must be represented with a VP format.

[5] The condition number of a Hilbert matrix of dimension n is $\mathcal{O}\left(\frac{(1+\sqrt{2})^{4n}}{\sqrt{n}}\right)$.

[6] Looking at Appendix A.3, lines 53-63, the GE kernel is performed on several UEs. We decided to show the results regarding the largest supported UE since it is the one that shows the most significant results. The experiments with the other UEs show the same results but saturated to the maximum MBB that the target UE can support. For small UEs, like for small IEEE 754 formats, the computational error of the result of the GE kernel is so that no one output binary digit is correct.

[7] The UNUM Environment defines the maximum bit length that a number can have in memory. MBB drives the actual bit length of the number in memory, and WGP defines the computing precision associated with the current operation.

```
13 void call gauss(unsigned ess, unsigned fss, unsigned mbb, unsigned wgp,
       unsigned n){
14     //declaration of the input-output data
15     vpfloat<unum, ess, fss, mbb> *A //input matrix
16         = (vpfloat<unum, ess, fss, mbb>*)malloc((n*n)*sizeof(vpfloat<unum,
       ess, fss, mbb>));
17     vpfloat<unum, ess, fss, mbb> *b //input array
18         = (vpfloat<unum, ess, fss, mbb>*)malloc((n)*sizeof(vpfloat<unum, ess
       , fss, mbb>));
19     vpfloat<unum, ess, fss, mbb> *x //result array
20         = (vpfloat<unum, ess, fss, mbb>*)malloc((n)*sizeof(vpfloat<unum, ess
       , fss, mbb>));
21
22     //set_ess_fss_mbb_rnd(ess, fss, mbb, rnd=RTN); // implicit set of the
       ESS, FSS, and MBB status registers
23     set_wgp(wgp); // set the WGP status regster
24
25     generate_hilbert(ess, fss, mbb, wgp, A, n);
26     rnd_uniform(ess, fss, mbb, wgp, b, n);
27     gauss(ess, fss, mbb, wgp, A, b, x, n); // call the gauss kernel
28     print_result_and_errors(ess, fss, mbb, wgp, A, b, x, n);
29
30     // free allocated elements
31     free(A); free(b); free(x);
32     return;
33 }
```

Listing 6.2: Gauss test environment

For each MBB-WGP configuration, the input-output variables are allocated in memory, lines 15-20. The environment status registers are set according to the environment configuration, lines 22-23, and they are kept constant for all the computation time. Starting from initial data encoded in an IEEE-754 format, the input matrix A and array b are filled with the current environment configuration, lines 25-26[8]. After the GE kernel execution, line 27, the computation error $\| Ax - b \|_2$ is computed, and the final result x is printed. Before starting a new iteration, the previously allocated variables are freed, line 31.

The GE kernel follows Algorithm 4. The functions signature is the following:

```
35 void gauss_vp(
36             unsigned ess, unsigned fss, unsigned mbb, unsigned wgp,
37             vpfloat<unum, ess, fss, mbb> *A,
38             vpfloat<unum, ess, fss, mbb> *b,
39             vpfloat<unum, ess, fss, mbb> *x,
40             unsigned n
41             ) {
42     unsigned k, imin, i, j;
43     vpfloat<unum, ess, fss, mbb> valmin, tmp, p, sum; //allocate temporary
       VP variables
```

Listing 6.3: Gauss function signature

As `gauss_vp` is a driver routine, which means that it may be called directly into the applicative code, its signature is standard C and only contains regular C types. In particular, the configuration parameters, at line 36, and the VP input-output data structures, at lines 37-39. Inside the function, the computation uses variable precision temporary variables, line 43.

[8] As it is possible to see in Appendix A.3, lines 404-428, for each element of the A matrix (H in this function), its Hilbert value is computed in double, line 422, then it is converted in UNUM, lines 423-424, and it is written in the target matrix, line 425.

The kernel of the algorithm follows Algorithm 4, and it is very similar to a plain C implementation (Appendix A.1). It triangulates the input matrix with the GE methods (lines 46-78 in Appendix A.2, and lines 120-235 in Appendix A.3), and it does the backward propagation of the result in the output vector (lines 80-86 in Appendix A.2, and lines 252-288 in Appendix A.3).

All the operations inside the kernel refer to some variables in memory. Thus, each line of pseudocode shares the same pattern. They are translated in some loads for the input values, an arithmetic operation between the loaded inputs, and a store for the output value. However, some cases in which the computation can be kept internally, and some load and store operations, can be avoided.

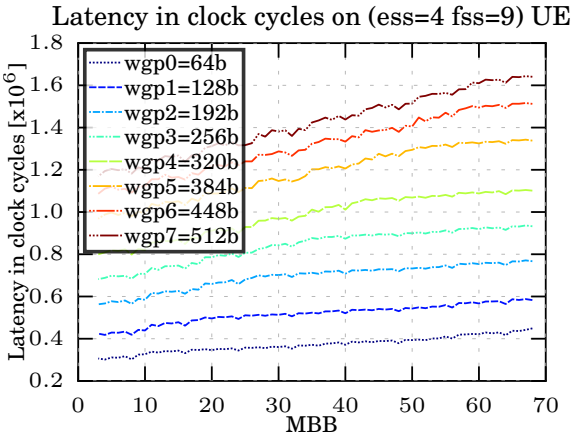
```
72          for(i = k+1; i < n; i++) {
73              p = A[i*n+k]/A[k*n+k]; // the division is implemented in
    assembly on wgp*64 bits leveraging only the coprocessor internal
    registers
74              for(j = 0; j < n; j++)
75                  A[i*n+j] = A[i*n+j] - p*A[k*n+j]; // the MAC operation is
    done on WGP*46 bits on internal registers, the result is stored in
    memory on MBB bytes
76              b[i] = b[i] - p*b[k]; // the MAC operation is done on WGP*46
    bits on internal registers, the result is stored in memory on MBB bytes
77          }
78      }
79      // Solve the triangularized system
80      x[n-1] = b[n-1]/A[(n-1)*n+n-1]; // the division is implemented in
    assembly on wgp*64 bits leveraging only the coprocessor internal
    registers
81      for(i = n-2; i > -1; i--) {
82          sum = 0; // accumulator variable kept in the internal register, its
    content is not stored in memory
83          for(j = n-1; j > i; j--)
84              sum = sum + A[i*n+j]*x[j]; // the MAC operation is done on WGP
    *46 bits on internal registers
85          x[i] = (b[i] - sum)/A[i*n+i]; // both subtraction and division
    operations are done on WGP*46 bits on internal registers, the result is
    stored in memory on MBB bytes
86      }
```

Listing 6.4: Gauss fused operation examples

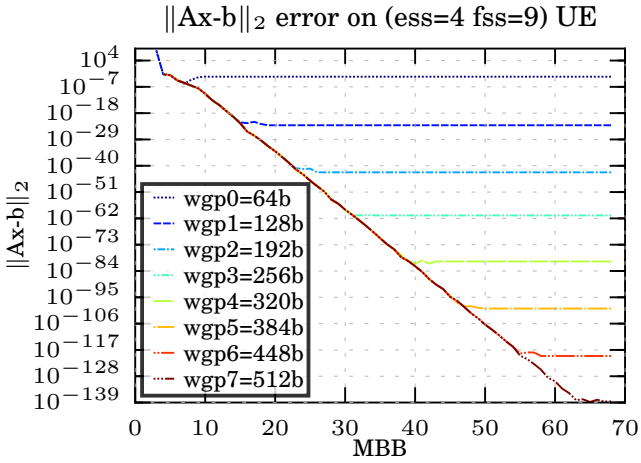
Line 73 hosts a division, this division is done by software using the Newton-Raphson method fully in the coprocessor register. Other examples can be line 75, where the multiply-and-accumulate operation is fused in internal registers, or in lines 82-84, where the `sum` variable is not allocated in memory. The accumulated value is kept in an internal register.

Figures 6.2a-6.2c-6.2e and Figures 6.2b-6.2d-6.2f depict how the latency in clock cycles and the computational error behave according to the computation precision (WGP) and to the memory footprint (MBB) used for variables defined in the `(ess=4, fss=9)` UE. Every color on the graphs corresponds to a different WGP value, and the x-axis denotes the MBB variation. As an example, the computation precision using a 64-bit IEEE FP format can be associated with the resulting point with WGP=0 and MBB=8.

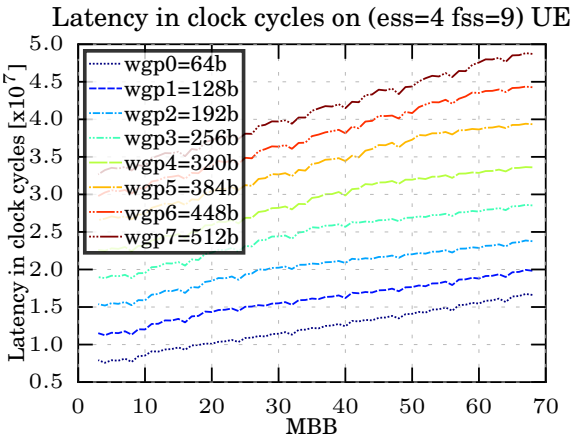
Under the latency performance point of view, with 64-bit precision (WGP=0), the VP unit runs between 5 and 8 times slower than the RISC-V's 64-bit FPU. For graphical representation issues, these points do not appear in the plots. There are three main reasons for this. Firstly, the pipeline chains inside the VP unit are deep. Secondly, the division is implemented in software. Thirdly, the (hand-written) assembly code running on the VP unit does not exploit the coprocessor's internal pipelines. However, the extra latency overhead obtained by increasing WGP is less important than the gained computation precision.



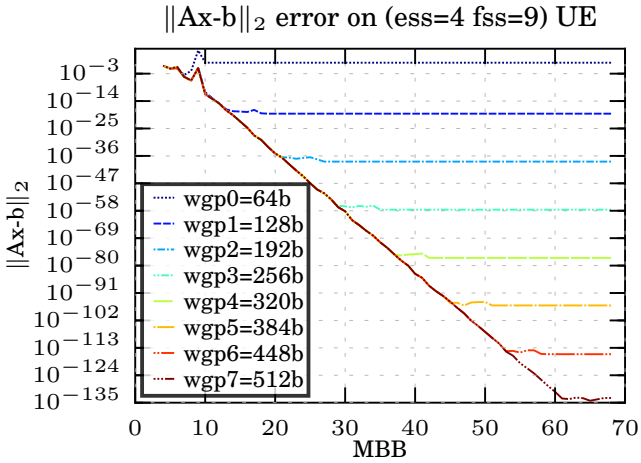
(a) Performance on Hilbert matrix size of 15.



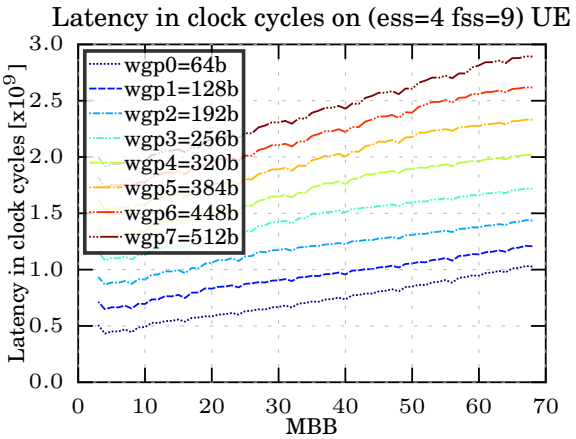
(b) Precision on Hilbert matrix size of 15.



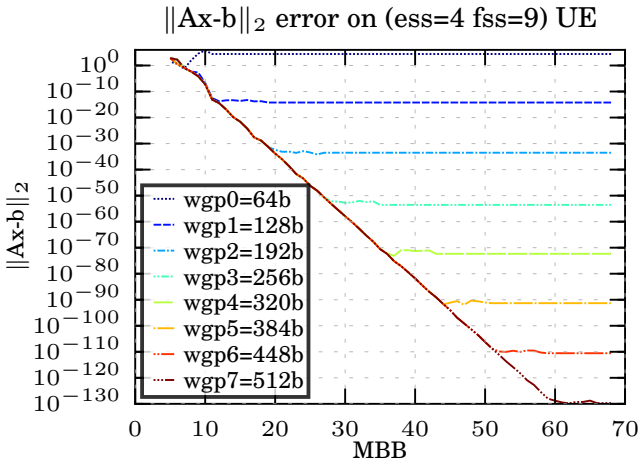
(c) Performance on Hilbert matrix size of 50.



(d) Precision on Hilbert matrix size of 50.



(e) Performance on Hilbert matrix size of 200.



(f) Precision on Hilbert matrix size of 200.

Figure 6.2: Latency and precision measurements for the Gauss kernel (lower is better) using the modified UNUM type I format in the (4, 9) UE, varying the data memory footprint (MBB), the internal computing precision (WGP), and the input Hilbert matrix size.

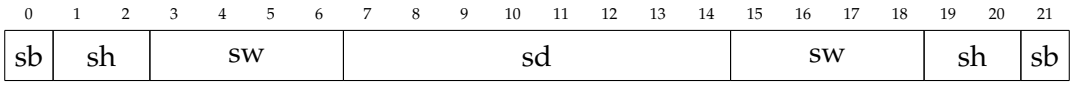


Figure 6.3: Worst case scenario of loading (or storing) of a 22 Bytes from (or in) memory

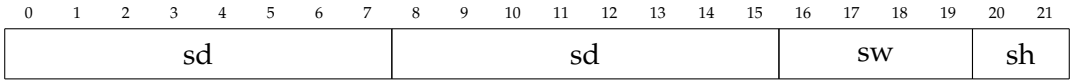


Figure 6.4: Best case scenario of loading (or storing) of a 22 Bytes from (or in) memory

The application latency increases linearly varying the MBB and the WGP values. Even if the latency of a multiplication increases quadratically on WGP, we still get a linear increase because most of the clock cycle latency is due to memory operations (load and stores).

The computational error decreases linearly (in logarithmic scale) varying the MBB and the WGP values. For each WGP value, a flat zone appears when the mantissa precision, corresponding to the chosen MBB value, is greater than the one specified in WGP.

This result shows that there is no gain in storing data in a memory format with greater precision than the one of the data itself. This experiment shows that, given a target error bound, it is possible to choose the optimal WGP-MBB tradeoff to reach it where the latency is minimal while preserving the results output precision. This tradeoff takes into consideration the expected average error to evaluate the results output precision.

This experiment shows that, with our architecture, we can reduce the computational error in FP scientific applications. Figure 6.2f shows this gain: by using the VP architecture, it is possible to reach up to 130 exact decimal digits (MBB=68 and WGP=7). With MBB=8 and WGP=1 precision setting, which corresponds to the precision of the conventional 64-bit IEEE 754 format, the GE algorithm gives a wrong result.

Reaching 130 exact decimal digits with our VP unit is not a significant achievement since it makes no sense to have output results more accurate than the input data. In the case of the 64-bit IEEE-754 format, they can only represent exactly 15-16 decimal digits. The real interest of being able to increase the accuracy of the results is to solve big problems with direct methods (e.g., GE), instead of using iterative methods. Iterative methods, compared to direct ones, degrades the latency performance with a factor of 10. Thus, the support of VP formats helps to push forward the limit for which direct methods can be used (instead of iterative methods). In this case, the latency overhead in using VP formats for direct methods can be lower than the latency of iterative methods with conventional formats. Moreover, the VP unit capability to achieve such high-accuracy increases the chances of obtaining output data with an accuracy similar to that of the input data.

This experience also validates the choice of a Load and Store Units (LSU) that support misaligned memory addresses. The coprocessor LSU can do memory operations in a random byte-aligned address in memory, for all possible data sizes (MBB) and formats. To do this, the coprocessor LSU divides misaligned memory operations in smaller aligned memory operations, one for each sub-portion of the data.

The number of these data sub-portions depends on the data size and the memory address. For instance, a memory operation for a 22-byte data can be divided into seven sub-portions if the address of the data is byte-aligned. This scenario (Figure 6.3) generates a sequence of memory operations on sub-portions of 8, 16, 32, 64, 32, 16, and 8 bits. If the data address is 64-bit aligned, the same 22-byte data is divided into only four sub-portions. This other scenario (Figure 6.4) generates a sequence of memory operations on sub-portions of 64, 64, 32, and 16 bits. Counter-intuitively, the latency plots results show that the overhead of dividing misaligned

memory accesses in smaller memory operations is not dramatic and does not considerably impact the execution latency performance.

This low latency impact is due to our optimized LSU that generates the minimum number of load-store operations depending on each misaligned access value. Furthermore, the reader can also observe the correlation between memory footprint and computation precision. Varying WGP and keeping MBB constant it does reduce the computational error, which means that the programmer must manage the memory footprint and computation precision together. As expected [22], the round-off error is bounded and behaves linearly according to the problem size.

Summary In this experiment, we demonstrated that:

- With our Variable-Precision (VP) architecture, we can push further the limit for which it is still possible to use direct methods, instead of iterative ones.
- The VP unit capability to achieve such high-accuracy (up to 130 decimal digits) increases the chances of obtaining output data with an accuracy similar to that of the input data.
- The support of misaligned memory addresses in the load and store unit of our VP architecture does not significantly impact the overall system performance.

6.2 Experiment 2: Variable-precision benefits in iterative methods

The second experiment presented in this work analyzes the efficiency of the Variable-Precision (VP) architecture when running in iterative methods. This experiment uses the Conjugate Gradient (CG) algorithm. This method solves a system of linear equations for which their matrices are Symmetric and Positive-Definite (SPD). The iterative version of this algorithm iterates on a simple kernel. This kernel, starting from a first (guessed) approximation of the final result, compensates the result computational error of the previous iteration to reduce the distance from the exact solution of the linear system. If at the end of this kernel, the computational error is higher than a threshold value provided at the beginning of the algorithm, it performs an additional iteration.

Algorithm 5 Conjugate gradient: general algorithm

1: **function** CONJUGATEGRADIENT(A, b, N, x_0) ▷ $A_{n\times n}$, $b \in \mathbb{R}^n$, $x_0 \in \mathbb{R}^n$
2: $p_0 \leftarrow r_0 \leftarrow b - Ax_0$
3: $k \leftarrow 0$
4: **while** iteration count not exceeded **do**
5: $\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k}$
6: $x_{k+1} \leftarrow x_k + \alpha_k p_k$
7: $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
8: **if** $\| r_{k+1} \| \leqslant tol$ **then** break
9: $\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
10: $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
11: $k \leftarrow k + 1$
return x_k ▷ $x_k \in \mathbb{R}^n \mid Ax = b$

Algorithm 5 shows the pseudocode for this kernel. This algorithm does not involve any *matrix-matrix* multiplication, but rather one *matrix-vector* multiplication, i.e., $A \times p_k$ at lines 5 and 7, and inner products, e.g., $r_k^T \times r_k$ at lines 5 and 9.

We have implemented the conjugate gradient algorithm, first in Julia, for reference, and then in C. We run it on standard examples taken from the MatrixMarket suite (available at [81], [82]). The code fragments given below show the conjugate gradient code, which is as close as possible from the pseudocode in Algorithm 5. Appendix B.1 lists the complete pseudocode of this function. Appendix B.2 shows its C-assembly implementation with which we obtained the results showed in this section. The function signature is given below:

```
1 int cg_vp(double *x, double *A, double *b, int m, int n,
2           int precision, double tolerance) {
```

Listing 6.5: CG function signature

As `cg_vp` is a driver routine, which means that it may be called directly into the applicative code, its signature is standard C and only contains regular C types. However, inside the main loop, the computation uses variable precision arrays. For example, vector `x_k` is defined as below:

```
13     vpfloat <unum, 4, get_fss(precision), get_mbb(precision)> *x_k =
14         (vpfloat <unum, 4, get_fss(precision), get_mbb(precision)>*)malloc(n
       *sizeof(vpfloat <unum, 4, get_fss(precision), get_mbb(precision)>));
15     //set_ess_fss_mbb_rnd(ess=4, get_fss(precision), get_mbb(precision), rnd
       =RTN); // implicit set of the ESS, FSS, and MBB status registers
16     set_wgp(get_wgp(precision)); // set the WGP status regster
```

Listing 6.6: Initialisation for vector `x_k` and implicit status register set

As for the Gauss elimination algorithm (Section 6.1), the coprocessor status registers (DUE, WGP, MBB, ...) are implicitly set immediately after the variable instantiation. Unlike Gauss, not all the variables are stored with a VP format. For instance, the input matrix A and the input vector b are kept in double format, they are not converted in VP format. This heterogeneity in the memory formats minimizes the memory footprint of variables while improving the result precision.

The core of the algorithm is very similar to a plain C implementation: it calls modified versions of the usual BLAS routines, *i.e.* `vgemvd` for matrix-vector multiplication, `vdot` for vector-vector dot product and `vaxpy` for $A \times x + y$.

```
24     for (nbiter = 1; nbiter < m*3000; ++nbiter) {
25         vgemvd(precision, m, n , 1.0, A, p_k, 0.0, Ap_k);
26         alpha = rs/vdot(precision, n, p_k, Ap_k);
27
28         vaxpy(precision, n, alpha, p_k, x_k); //x = x + alpha*p
29         vaxpy(precision, n, -alpha, Ap_k, r_k); //r = r - alpha *Ap
30         rs_next = vdot(precision, n, r_k, r_k); // rk+1
31
32         if (sqrt((double)rs_next) < tolerance) {
33             copy_vector_d_vp(precision, x, x_k, n);
34             break;
35         }
36         vscal(precision, n, rs_next/rs, p_k);
37         vaxpy(precision, n, 1.0, r_k, p_k);
38         rs_next = rs;
39     }
40
41     free(r_k); free(p_k); free(x_k); // free allocated elements
```

Listing 6.7: CG core loop

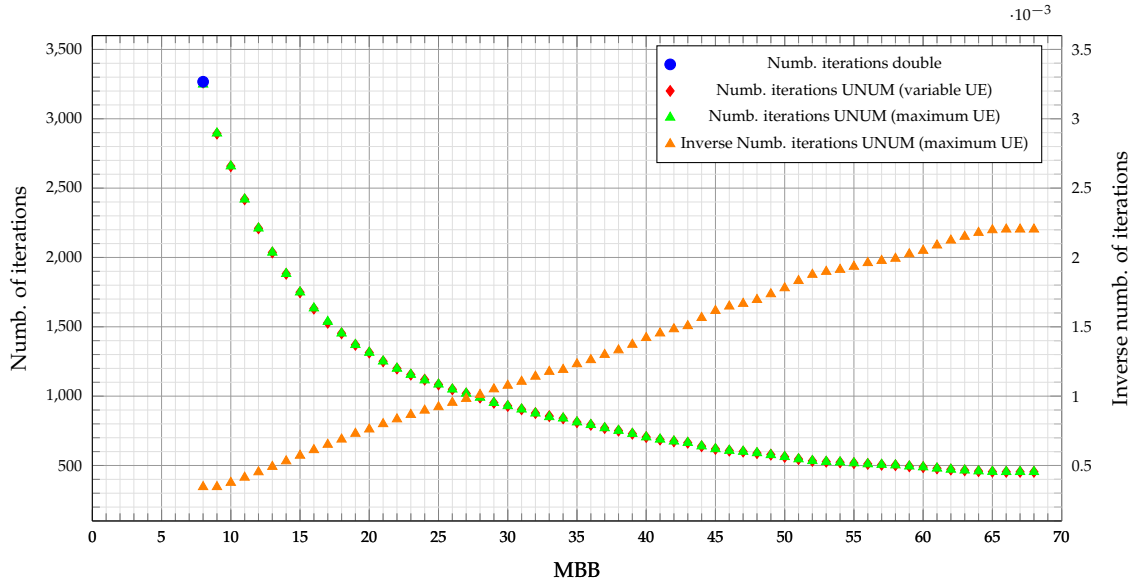


Figure 6.5: Number of iterations, and the inverse of the number of iterations, to run the Conjugate Gradient algorithm on a 237×237 SPD matrix, with tolerance: 1.00e-6

This experiment aims to prove that our VP architecture can reduce the number of iterations by augmenting the precision of intermediate variables. As in the previous experiment, the CG algorithm is slightly adapted to be executed in the VP architecture. As for the Gauss Elimination experiment, the code is written in assembly using the instructions provided in Section 4.7. Unlike direct methods, VP for iterative methods is applied only to the intermediate data. Thus, the input matrix stays untouched, and only the precision of the intermediate data is augmented. This different usage of memory formats reduces a lot the memory footprint of variables in the main memory.

In this experiment, the WGP and DUE (Section 4.5.4) values are minimized depending on the MBB value set. The minimal ess is obtained by running multiple times the CG algorithm, decrementing the ess value, up to the point when the algorithm does not converge anymore. The minimal fss is obtained by subtracting the maximum footprint of the number in memory (MBB), the ess precomputed size, and the other UNUM fields. The WGP status register is configured so that the VP unit arithmetic operations are performed with an accuracy of at least one bit higher than the maximum available in the formats of the variables involved[9]. With this modification, a speedup of the application execution is expected, without varying the number of iterations required to terminate the algorithm.

Figure 6.5 shows the number of iterations required by the CG algorithm, running on the VP architecture, varying the memory footprint of variables (MBB). In this experiment, a 237×237 SPD matrix[10] is used and the tolerance of the algorithm is set to $1.0 \cdot 10^{-6}$. This plot shows the number of iterations required by the CG algorithm to converge with different memory formats. The blue line shows those that use the conventional 64-bit IEEE floats. The red line shows those that use the UNUM format varying the DUE and WGP status registers on MBB. The green one shows those that use the UNUM format fixing the DUE and WGP status registers to their maximum supported values. This plot also shows the inverse of the number of iterations required by the CG algorithm to converge (orange), using the UNUM format with maximum precision (green).

[9] The maximum precision of the mantissa of a variable in the UNUM format can be extrapolated by looking at the values of the MBB, ess, and fss format parameters.

[10] The input matrix selected for this experiment is the NOS1 matrix, available in the MatrixMarket website `https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos1.html`

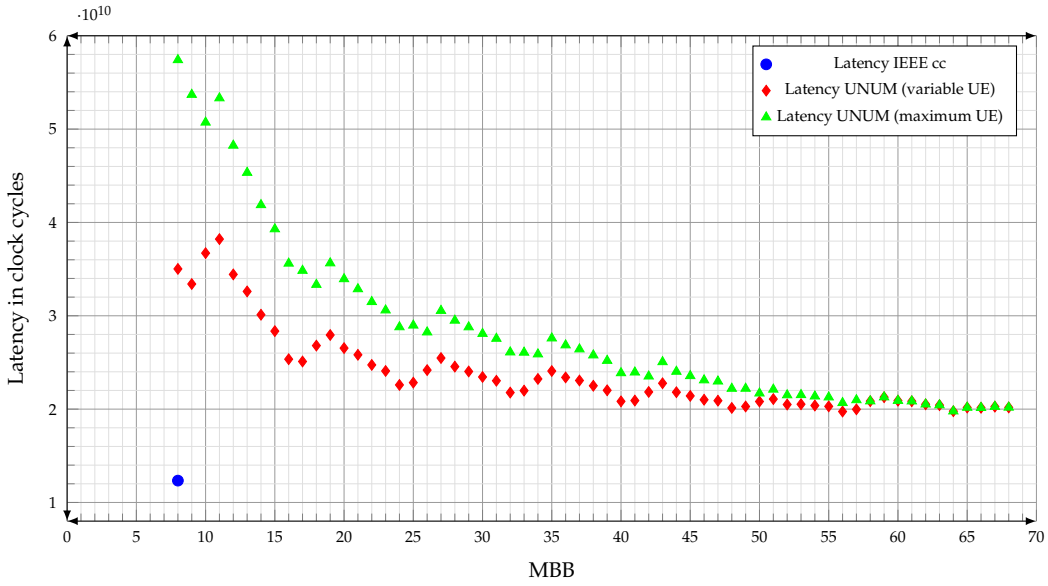


Figure 6.6: Clock cycle latency to run the Conjugate Gradient algorithm on a 237×237 SPD matrix, with tolerance: 1.00e-6

The graph shows that supporting VP Floating-Point (FP) data in memory reduces the number of iterations for the CG iterative algorithm drastically. This latency gain is thanks to the gained precision of the FP representation in memory that compensates the round-off error among iterations. As expected, the inverse of the number of iterations required by the CG algorithm to converge is linear when varying the precision.

Adapting the algorithm to dynamically vary the settings for the DUE and WGP status registers, slightly reduces the number of iterations required for low precision (nearby MBB=9, MBB=17, and MBB=32). This iterations reduction is due to the lower overhead of the utag in the UNUM number. However, this also highlights that the UNUM utag has a non-negligible overhead in the number representation: it takes some bits that could encode the mantissa and improve the number precision.

The number of iterations needed to make the algorithm converge with the IEEE-754 64-bit representation, and the number needed to make the same algorithm converge with the UNUM format (Section 4.3) at MBB=8, for both maximum and variable UE, is comparable. Thus, at least for this application and with this input data set, the utag bit-overhead in the UNUM format does not increase the number of iterations needed to make the algorithm converge. In other words, even if some variables in the UNUM format have less precision than the one of the 64-bit IEEE-754 representation, and the `fs-1` field of the UNUM encoding wastes potential mantissa bits, the number of iterations required by the application to make the algorithm converge does not increase. This result is fascinating because it leaves room for future research work on VP FP formats, for instance, by exploring a modified version of the UNUM format with fixed size and where it has not the `fs-1` field.

Figure 6.6 complements Figure 6.5. It shows the latency, in clock cycles, required by the CG algorithm, on the VP unit, varying the variables memory footprint (MBB). Varying the VP variable setting depending on MBB can bring almost 50% of savings in the application latency. Due to the long pipelines in the VP unit and the long conversion functions in the coprocessor LSU, the latency of the VP unit remains higher than that of an FPU compliant with the IEEE-754 standard, even if we reduce the number of iterations.

In line with the Gauss elimination algorithm, the latency overhead of the VP unit, compared to the latency of a 64-bit IEEE FPU, for the same precision, is between 4× and 7×, for the dynamic and fixed coprocessor status register settings, respectively. However, unlike the

GE algorithm, the increment of the computation precision reduces application latency. This speedup highlights the strengths of the VP architecture presented in this work: the capability of compensating the latency of a classical FPU by improving the computation precision and reliability of the results, for scientific applications.

In this specific experiment, there is an asymptotic $2\times$ latency overhead. This overhead could be reduced by simplifying the conversion functions in the coprocessor LSU and by simplifying VP memory format. These improvements can help to improve the VP unit up to the point that the application latency can be less than the one of the IEEE FPU.

In this section, the comparison between the two units is biased since the FPU supports only one fixed 64-bit format while the VP unit can handle several formats of any length ranging from 1 to 68 bytes. The next section does an unbiased latency evaluation of the VP unit. It compares the VP unit with a software library that can provide the same levels of accuracy.

Summary In this experiment, we demonstrated that:

- Our Variable-Precision (VP) architecture can reduce the number of iterations that an iterative solver requires to converge.
- The number of iterations required by running the experiment with the IEEE-754 double format and with the UNUM format at 64 bits is almost the same. This similarity indicates that even if the `fs-1` UNUM field wastes potential mantissa bits in encoding, the application performance is not degraded.
- Thanks to the augmented precision of our VP architecture, the application latency for low data precisions settings (e.g., 64 data) can be reduced of almost 50% if high precision data are used instead (e.g., 544 bits).

6.3 Experiment 3: performance benchmark MPFR vs. UNUM

Previous experiments benchmark the Variable-Precision (VP) Floating-Point (FP) coprocessor against a specialized 64-bit FP Unit (FPU) compliant with the IEEE-754 standard. This section envisages an unbiased performance evaluation of the VP unit. As a baseline, there is no hardware solution in state of the art available for comparison. For this reason, this experiment compares the VP coprocessor against the MPFR software library [9].

For an unbiased comparison between hardware- and software-based solutions, MPFR was firstly optimized for RISC-V as it already is for other ISAs. The routines used by MPFR use the GMP library [15]. These routines are written in assembly, and some of them were missing for the RISC-V instruction set.

For every supported Instruction Set Architecture (ISA), for instance, MIPS, ARM, and X86, GMP defines different routines that implement the same functionalities and leverage the ISA of every target platform. For the ISA not supported, the application compiler generates an assembly code not optimized, starting from a high-level description of the GMP routine in the C language. Many mathematical functions within the MPFR library use the `umul_ppm` and `mul_1` GMP routines extensively. However, there is not an optimized assembly routine in GMP for the RISC-V ISA. We have re-written these GMP routines to take advantage of the `mulhu` [83] available in the RISC-V ISA. With this new RISC-V routine, the number of instructions of `umul_ppm` and `mul_1` GMP routines was reduced from 25 and 12 to 11 and 3, respectively. Because MPFR uses GMP routines underneath, the implemented routines help to improve MPFR applications.

To benchmark the VP hardware unit, this experiment uses three matrix-based linear solvers as case studies for Variable-Precision (VP): the Gauss elimination (GE), the Conjugate Gradient (CG) and the Jacobi (JA). The Jacobi method is an iterative algorithm for determining the

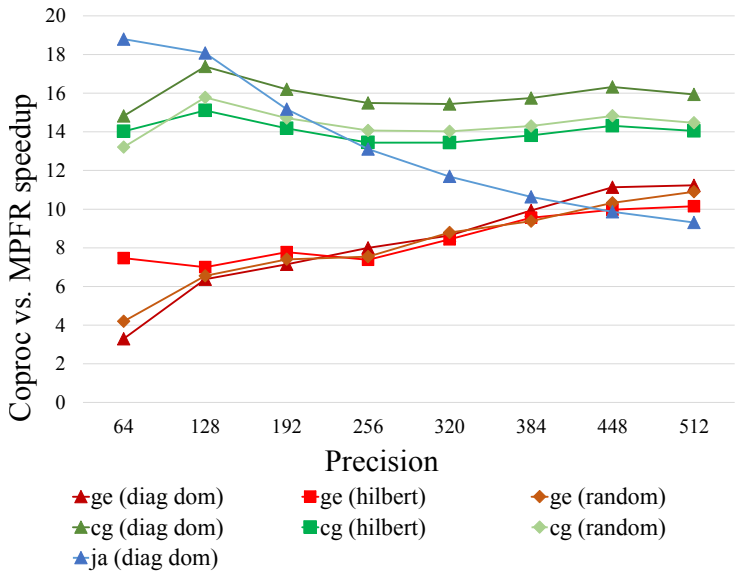


Figure 6.7: Performance comparison between our variable-precision coprocessor unit and the MPFR software library: speedups between 3.5× and 18×

solutions of a strictly diagonally dominant system of linear equations. It is selected as a complement to the CG algorithm for iterative methods.

These algorithms execute long chains of multiply-addition operations that tend to accumulate errors, so they are suitable for experimentation. We selected three different matrices for performance and error evaluation: a Hilbert 15×15 matrix (Hilbert), a randomly generated 24×24 matrix (random), and a 40×40 dominant diagonal matrix (diag dom). Applications GE and CG were compiled and run for all three configurations, while JA used only a diagonal-dominant matrix due to algorithm constraints.

Figure 6.7 shows the performance comparison between our solution and the baseline. For every type of matrix in input, for every solver, this figure shows the speedup of using our VP unit over the optimized MPFR software library for several VP settings (MBB). The input and output data of the linear solvers are encoded in double, while the internal computation is performed with extended precision by varying the MBB parameter.

Having hardware support for higher-than-standard precisions shows a clear advantage over a software solution with speedups between 3.5× and 18×. We notice how speedups vary according to the application and precision used. They depend on the algorithm types, while matrices sizes and types have little influence over them.

Summary In this experiment, we demonstrated that our Variable-Precision (VP) architecture is between 3.5 times and 18 times faster than an optimized version of the MPFR [9] software library.

Chapter 7

Conclusions and future works

Several applications, such as linear system solvers and computational geometry, can benefit from Floating-Point (FP) formats different than those provided by the IEEE 754 standard. Some of them do not even converge with the largest FP format provided by the standard. For these applications, FP formats with different bit-length ratios between exponent and mantissa fields, or those with augmented mantissa precision, can reduce the output computational error. The Variable-Precision (VP) category of FP formats groups all these formats. This work explores VP FP formats and architectures to improve the computation precision of results for scientific computing applications.

The computational error in iterative solvers can be bounded by increasing the precision of intermediate variables to a precision higher than the one of the input data. This property leads either to compute intermediate variables internally on higher precision into the FP Unit (FPU) or, for conventional IEEE 754 FP formats, to promote some variables to a more precise representation in memory. This thesis supports both techniques within the same hardware architecture.

To support this, we propose a VP FP computing system based on three computation layers. The *internal* layer, within the VP hardware unit, is dedicated to computing. The *external* layer, within the memory, hosts fixed-precision input-output values (e.g., IEEE-754 formats). Finally, the *intermediate* layer expands in memory the variables from the internal layer. The latter uses a VP FP format with bit-width granularity expressed in bytes. To the best of our knowledge, this thesis is the first work that proposes a multi-layered hardware architecture, which addresses multi-layered precision, designed to reduce the output computation error of scientific solvers.

Every computation layer leverages a different data representation. The intermediate format in memory leverages a modified version of the UNUM type I VP FP format, supporting several Unum Environments (UE), with up to 16 and 512 bits of exponent and mantissa, respectively. Internally, the VP unit supports a more hardware-friendly format with separated exponent and fraction fields, and which supports 18 bits of exponent and a normalized mantissa of up to 512 bits. Due to hardware design constraints, internal mantissa fields have a precision granularity of 64 bits. The conversions between the intermediate and internal formats are handled on-the-fly by a dedicated Load and Store Unit (LSU). This LSU supports misaligned accesses in the main memory on byte-aligned VP data. It can compact data in memory while being able to control their sizes and format configurations.

This unit is designed to speedup VP FP software routines for high precision scientific computing. Its design targets high-performance computing servers. It is implemented as a coprocessor of a RISC-V processor. The choice to adopt a RISC-V architecture is purely opportunistic because it is open-source. Other architectures could be adopted instead, for example, ARM. All the coprocessor features are exposed to the user expanding the RISC-V Instruction Set Architecture (ISA).

The VP unit supports Interval Arithmetic (IA) since the UNUM format supports it implicitly. Iterative algorithms, or algorithms that compute a long chain of operations, may be affected by interval explosion effects (Section 2.1.3). The main idea is to avoid this effect by using

IA only for computing the computational error of the algorithm with an accurate interval of confidence. For all the other parts of algorithms, we use scalar operations. However, we did not exploit intervals in experiments due to time constraints.

The VP unit may be seen as an FPU for VP. Unlike conventional FPUs, it hosts numbers in an internal Register File (gRF), which can contain 32 intervals (gbound) with up to 512 bits of mantissa per interval endpoint. The gRF entries can be used for internal fused operations before going to memory with a format with lower precision (ubound). The precisions of the formats in the gRF and in memory can be different and it can change at run time. Instead of having dedicated assembly instructions, the settings of the types of format used in memory and the involved precision during internal calculations are done by configuring internal status registers.

The VP unit architecture is pipelined with an internal data-path fixed at 64 bits. While the intermediate format's memory footprint is programmable with a byte-level resolution, the internal computation is tuneable with a 64-bit granularity. Those two parameters allow us to exploit a precision-latency tradeoff.

Compared with a fixed-precision FPU IEEE-754 compliant, our architecture requires several instructions in the ISA to support all the formats supported by the dedicated LSU, synchronization steps for the interval endpoints, and additional operations steps compared to scalar FP operations. These functionalities lead to having the LSU of the VP unit with several pipeline stages to support formats conversions. The number of stages can be reduced by simplifying the VP FP memory format or by not supporting subnormal encodings. The hardware complexity and overhead are a direct consequence of the decision to support both the UNUM format and VP in the LSU of the VP unit. Unlike conventional formats, the supports of VP formats in memory requires an additional rounding step during store operations.

Synthesis results show that the area footprint and the power consumption of the VP unit are 9 and 12 times higher than those of the RISC-V 64 bits FP unit. The large area footprint required by the VP unit is due to the extensive usage of internal memory elements (gRF and pipeline buffers). The VP unit power consumption is dissipated mainly in the mantissa buffers, and in the internal logic to handle VP mantissa fields. These overheads are kept under control thanks to the internal data-path bit-width in the VP unit fixed at 64 bits.

Experiments show that this three-layers VP architecture, thanks to higher precision support both internally and in main memory, gives better accuracy than a conventional FPU based on IEEE 754 format having a reasonable latency overhead. For direct solvers, the possibility to achieve precisions in the VP unit much higher than those of a conventional FPU pushes further the limit for which it is still possible to use direct solvers, instead of iterative ones. This limit is fundamental since iterative solvers require several operations more than direct ones. For iterative solvers, the possibility of achieving such high precisions opens the possibility to make converge solvers that were not converging with a 64-bit double.

In iterative solvers, VP formats help to drastically reduce the number of iterations required by the iterative algorithm to converge. Although the VP unit is, on average, eight times slower than a 64-bit fixed-precision FPU, the increase in computational accuracy with VP format support in memory can reduce application latency by an overhead factor of 2. This speedup highlights the strengths of the VP architecture presented in this work: the capability of compensating the latency of a classical FPU by improving the computation precision and reliability of the results, for scientific applications. This overhead factor could be reduced by simplifying the conversion functions in the VP unit LSU and by simplifying VP memory format.

In iterative methods, our VP unit can approach the working precision (i.e., the precision on which input numbers are defined) close, within a given factor, compensating the bad conditioning of the input data set. This compensation is done by increasing the computation precision (up to 512 bits) in memory and internally the VP unit. The usage of augmented precision may reduce the application latency of up to 50%. If the input data are considered as exact, for the

conjugate gradient iterative algorithm, increasing the computation precision for the intermediate data always reduces the number of iterations needed by the algorithm to converge. This property is valid even for a relatively small problem with an input matrix of 237 $\times$ 237 elements.

The number of iterations required to run the experiment with the 64-bit IEEE-754 format and the UNUM format, with the same memory footprint, is almost the same. This similarity indicates that even though the `fs-1` UNUM field wastes potential mantissa bits in the number encoding, the accuracy of the application does not significantly degrade. A modified UNUM format that does not have the field `fs-1` can improve the accuracy of the output algorithm. Section C.1 describes this format.

Compared with the MPFR software library, the VP unit achieves speedups between 3.5$\times$ and 18$\times$ with comparable accuracy. Additional experiments demonstrate minimal performance degradation when operating on misaligned data.

The software paradigm presented in this work keeps under control the memory footprint of intermediate data, by only storing in main memory intermediate variables that require higher precision. We measured a flops performance eight times lower than a fixed-precision IEEE-754 FPU, but six times faster than a multiple-precision software library (e.g., MPFR) with the same output accuracy. Part of this overhead is due to hardware support for interval arithmetic in the VP unit, which is not exploited for the target applications. Changing the unit to be scalar will halve the area and power values.

With this thesis, we do not claim that VP formats, or our unit, can be used for all types of applications. We measure that VP formats are expensive in latency, especially for high precision applications. This cost is due to the format conversion functions absent in conventional FPUs compliant with the IEEE-754 standard. However, VP formats can significantly reduce the computational error in scientific applications. For this reason, we propose the concept of Adaptive VP: use VP only when it is not possible to converge algorithms with standard IEEE 754 formats.

7.1 Conclusions about the UNUM format

In this thesis, we decided to support the UNUM format, in a hardware computing system, as is defined in the book of Gustafson [2]. In the design phase, we pointed out several implementation issues related to the UNUM format. We have defined a workaround to make the UNUM format hardware friendly and useable in real applications (Section 4.3).

According to Gustafson, if we leave data to choose their size and precision, the computation can speedup. This can be possible by exploiting the ubit in the UNUM format while computing with interval arithmetic. In iterative applications, the mantissa reduces its size at every iteration of the algorithm. Smaller mantissa implies to have smaller data, and smaller data implies to speed up the algorithm execution. Nevertheless, this speedup cannot be exploited for two main reasons: either the interval explosion effect arrives before that the algorithm finishes (e.g., in Gauss also for 15x15 matrices); either the interval explosion effect prevents the algorithm from converging (e.g., in conjugate gradient). In conclusion, having VP features in real applications is useful only if the VP formats are kept under the programmer's control.

Supporting the original UNUM format has several drawbacks. Firstly, the ubit hinders the control of the interval explosion like it is made in the MPFI [24] software library. Secondly, the UNUM support requires a significant hardware overhead as well as multiple additional rounding steps. Thirdly, the latency of load and store operations becomes similar to the one of an FP operation (e.g., addition). Finally, in the case of scalar computation, the round to nearest (round half away from zero) policy can be emulated by alternating arithmetic and guess operations (Section 4.1.2).

These results suggest that an ideal VP computing system would be a Schulte-based scalar FPU supporting several parametrizable VP formats in memory, in combination with a hardware-friendly internal format. The VP formats to be supported can be those proposed in Appendix C, plus a memory format that uses only the exponent encoding from UNUM (a UNUM with parametric length and without the ubit and fs-1 fields). However, separating the VP unit from the main core (e.g., implementing it as a coprocessor), is sub optimal for the overall system performance. Implementing the VP unit as a tightly coupled VP FPU to the main core would improve the overall system performances since there is no need to move intermediate data outside the main core.

The operation flow will be the same presented here: arithmetic operations leverage the internal format, and conversions among this format and the VP ones in memory, are done during load and store operations. The supported memory formats will be more straightforward than the UNUM one. In this VP unit, the programmer will have the freedom to choose the set of formats (with different configurations) which best suits the programming and precision needs. This investigation is left as future work.

7.2 Future works

The results obtained out of this thesis are not an end in themselves. Results showed that Variable-Precision (VP) computing could bring significant advantages in computing systems by improving results precision (Section 6.1), by reducing latency computation (Section 6.3), and by simplifying the software coding (Section 4.8). Nevertheless, there are still a lot of open questions, such as:

- How to optimize the existing hardware platform?
- How to integrate this platform into existing software toolchains?
- Are there any Variable-Precision formats not yet available in state of the art?

The next sections answer these questions individually. Section 7.2.1 lists all the hardware optimizations that could improve the performances on the VP unit. Section 7.2.2 proposes a solution to port already-existing software libraries in the VP hardware architecture. Finally, Section 7.2.3 focuses on the future perspectives on VP FP formats.

7.2.1 Hardware optimizations for the variable-precision architecture

Chapter 6 shows the advantages of using the Variable-Precision (VP) Floating-Point (FP) computing unit presented in this work. Unlike FP units (FPU) that use the IEEE 754 formats, experiments show that the VP unit can significantly increase the precision of results and drastically reduce the number of iterations in iterative solvers. The VP unit is not perfect and requires several hardware optimizations.

The VP unit requires between 4 and 7 times more latency than a conventional 64-bit FPU compatible with the IEEE 754 standard. According to Section 5.4, the synthesis results of the VP unit show that it requires nine times more area surface and twelve times more power consumption than a conventional FPU. These overheads are mainly due to the VP unit interface with the RISC-V, the long pipelines, the not-negligible number of mantissa buffers, and the complex finite-state-machines, inside the VP FPU.

Move the Variable-Precision unit inside the main core The interface between the RISC-V and the VP unit adds latency due to the presence of some queues. A solution to avoid this latency is to change the hardware architecture and to integrate the VP unit in the main processor,

like a standard FPU. A VP FPU will have better performance since it is part of the execution stage of the main core. The overall design is simplified because, having the VP FPU in the execution stage of the pipeline, the main core does not have to reschedule VP unit operations depending on the results of software branches.

Remove hardware support for interval arithmetic Interval Arithmetic (IA) cannot be used to solve scientific computing linear kernels due to interval explosion effects. In high-precision scientific computing, IA can be used to compute not-iterative algorithms, for instance computing the computational error at the end of a kernel of an algorithm. Support IA in hardware is overkill since the evaluation of the computational error is not critical in the overall latency of the algorithm.

An additional design simplification could be removing the hardware support for intervals while providing better control for the rounding modes of the VP FPU. With this modification, the number of macro-stages, needed for every macro-pipeline, is reduced because all the macro-stages that accelerate IA operators (e.g., input endpoint multiplexing, Chapter 5) are not needed anymore. Removing the hardware support for interval arithmetic will improve the area and power consumption with at least a factor of two. The user can keep performing IA operations by computing both endpoints separately, at the cost of increasing the execution latency when performing IA operations.

Maximize performances in the Variable-Precision unit We have to consider several aspects to reduce the execution latency of our VP unit. The latency of the critical path of the VP unit drives the frequency of the computing system. This frequency often belongs to the most complex macro-stage operator (e.g., mantissa multiplier or mantissa adder). Consequently, other macro-stages that require simpler operators (e.g., move or shift) benefit from unnecessary slack time. A solution to reduce this latency gap could be to augment the data-path bit-width of some macro-stages. In this way, the number of clock-cycles required from the macro-stage with augmented data path is reduced, as well as the clock-cycle latency of the overall pipeline.

If, after this step, the VP unit still has some timing paths with positive slack, it may be possible to merge some macro-stages. This merge may be possible for simple macro-stages like a shift amount computation macro-stage followed by a shift macro-stage. If, after merging these macro-stages, the critical path is unvaried, the number of macro-stages and the number of mantissa buffers in the VP unit is minimal. This configuration guarantees the minimum clock-cycle latency in the VP unit pipeline.

The VP unit can reach a throughput of 1 when computing data with only one chunk of mantissa precision. However, the current configuration of the VP unit pipelines uses only one macro-stage over two when performing operations with data on multiple-chunks. Under the throughput perspective, this utilization ratio halves the maximum throughput of the system. A solution to maximize the throughput of the system could be to double the mantissa buffers between macro-stages, at the cost of doubling the chip area. This cost must be analyzed carefully by evaluating the probability of having two chained operations in the same VP unit macro-pipeline.

Optimize low-precision computations Another possible optimization that we envisage is to optimize low-precision computations in the VP unit. When performing low-precision instructions, most of the data-path remains unused, and most of the hardware gates used to handle multiple-chunks, toggle without making any useful computation. A possible solution to optimize the execution latency and to reduce the energy consumed during low-precision VP operations could be to instantiate a fixed-size VP FPU in parallel to the already existing one. This new FPU is vectorized to maximize the usage of its register file 64-bit bit-width. It supports

all the VP FP formats up to 64 bits. It is vectorized so that it can support either 8 VP parallel operations on data defined in 1 byte, or 4 VP parallel operations on data defined in 2 bytes, or 2 VP parallel operations on data defined between 3 and 4 bytes, or 1 VP operation on data defined between 5 and 8 bytes.

This hardware specialized in low-precision operations minimizes the energy-per-operation involved in VP computing. When performing low-precision operations, the multiple-chunk high-precision VP FPU becomes dark silicon. Vice versa, when performing high-precision operations, the low-precision VP FPU becomes dark silicon.

7.2.2 Importing variable-precision support in existing software programs

This thesis does not cover how to integrate this architecture in existing software environments. The main challenge is to integrate this architecture into existing software kernels that may date decades and may not be maintained.

As stated in Section 4.8, this work proposes the `vpfloat` FP data type to adapt software programs that use IEEE 754 FP variables. These programs can be easily modified by changing the data-types of their variables in `vpfloat` with the correct setting of parameters. The only difficulty is to tune the precision of each `vpfloat` variable. In some cases, compilers can automatize this procedure.

High-precision scientific computing programs are commonly written with specialized tools such as Matlab or Mathematica. It is difficult to bring at this programming level the notion of VP formats. These software tools leverage several software libraries underneath that handle FP data types and basic routines. An example of a software library is BLAS.

The BLAS library implements a set of low-level routines for performing classical linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. These operations are written in the C language, and they target every type of architecture (e.g., X86, MIPS, ARM, and RISC-V). We propose to expand the BLAS library to use VP FP data by declaring `vpfloat` variables in the code. The main challenge of this approach is to select the proper precisions and data formats for each declared `vpfloat` variable. This challenge is one of the main future focus of this research.

7.2.3 Future perspectives for variable-precision formats

This thesis describes a hardware architecture that supports a modified version of the UNUM Floating-Point (FP) format (BMF, Section 4.3). This unit should be modified to support multiple VP FP formats in memory. Thus, it is possible to make a proper exploration of Variable-Precision (VP) FP formats. The minimum set of VP FP formats that the unit should support are the custom IEEE-like format, the posit FP format, and the BMF format described in Sections 3.1.1, 3.1.4, and 4.3.

With the experiments ran with the BMF format, we identified several improvements for the UNUM format and other VP FP formats, such as posit. Appendix C describes the improved UNUM format, the improved posit formats, a new format that is a tradeoff between the previous two, and a new family of not-contiguous VP FP formats.

Appendix A

Source code Experiment 1

The GE algorithm is a linear system solver divided into two main steps: the triangulation of the input matrix, and the backward propagation of the output vector. Section A.1 shows its implementation with the conventional `double` format, from the IEEE 754 standard. Section A.2 shows its implementation with the pseudo-type `vpfloat` format for variable precision. Section A.3 shows the C code that implements the same code of Section A.2 but using C and inline assembly. The latter is an extract of the code used to get the results showed in Section 6.1.

A.1 The Gauss kernel in conventional IEEE 754 formats

```
1  void gauss(double *A, double *b, double *x, int n) {
2      int k, imin, i, j;
3      double valmin, tmp, p, sum;
4      for(k = 0 ; k < n-1 ; k++) { // for each column
5          /* Research the minimum not-null element (in absolute value
6          * in the k column with index i>=k                              */
7          valmin = A[k*n+k] ; imin = k ;
8          for(i = k+1 ; i < n ; i++) { // for each line
9              if (valmin != 0) {
10                 if (my_abs(A[i*n+k]) < my_abs(valmin) && A[i*n+k] != 0) {
11                     valmin = A[i*n+k] ;
12                     imin = i ;
13                 }
14             } else {
15                 valmin = A[i*n+k] ;
16                 imin = i ;
17             }
18         }
19         /* Swap the elements of the matrix A and of the array b
20         * between the lines imin and k                                 */
21         for(j = 0 ; j < n ; j++) {
22             tmp = A[imin*n+j] ;
23             A[imin*n+j] = A[k*n+j] ;
24             A[k*n+j] = tmp ;
25         }
26         tmp = b[imin] ;
27         b[imin] = b[k] ;
28         b[k] = tmp ;
29         // Reduce the matrix with the Gauss elimination method
30         for(i = k+1 ; i < n ; i++) {
31             p = A[i*n+k]/A[k*n+k] ;
32             for(j = 0 ; j < n ; j++)
33                 A[i*n+j] = A[i*n+j] - p*A[k*n+j] ;
34             b[i] = b[i] - p*b[k] ;
35         }
36     }
37     // Solve the triangularized system
```

```
38      x[n-1] = b[n-1]/A[(n-1)*n+n-1] ;
39      for(i = n-2 ; i > -1 ; i--) {
40          sum = 0 ;
41          for(j = n-1 ; j > i ; j--)
42              sum = sum + A[i*n+j]*x[j];
43          x[i] = (b[i] - sum)/A[i*n+i];
44      }
45  }
```

./Chapters/Chapter6_gauss_double.c

A.2 The Gauss kernel for variable precision in pseudocode

```
1   #define MAX_ESS 4
2   #define MAX_ESS 9
3   #define N 200
4   void main(){
5       unsigned mbb, wgp;
6       for(wgp=0; wgp<=(2^MAX_FSS)/64; wgp++){
7           for(mbb=min_mbb(MAX_ESS,MAX_ESS); mbb<=max_mbb(MAX_ESS,MAX_ESS); mbb
        ++)
8               call_gauss(MAX_ESS, MAX_FSS, mbb, wgp, N);
9       }
10      return;
11  }
12
13  void call gauss(unsigned ess, unsigned fss, unsigned mbb, unsigned wgp,
        unsigned n){
14      //declaration of the input-output data
15      vpfloat<unum, ess, fss, mbb> *A //input matrix
16          = (vpfloat<unum, ess, fss, mbb>*)malloc((n*n)*sizeof(vpfloat<unum,
        ess, fss, mbb>));
17      vpfloat<unum, ess, fss, mbb> *b //input array
18          = (vpfloat<unum, ess, fss, mbb>*)malloc((n)*sizeof(vpfloat<unum, ess
        , fss, mbb>));
19      vpfloat<unum, ess, fss, mbb> *x //result array
20          = (vpfloat<unum, ess, fss, mbb>*)malloc((n)*sizeof(vpfloat<unum, ess
        , fss, mbb>));
21
22      //set_ess_fss_mbb_rnd(ess, fss, mbb, rnd=RTN); // implicit set of the
        ESS, FSS, and MBB status registers
23      set_wgp(wgp); // set the WGP status regster
24
25      generate_hilbert(ess, fss, mbb, wgp, A, n);
26      rnd_uniform(ess, fss, mbb, wgp, b, n);
27      gauss(ess, fss, mbb, wgp, A, b, x, n); // call the gauss kernel
28      print_result_and_errors(ess, fss, mbb, wgp, A, b, x, n);
29
30      // free allocated elements
31      free(A); free(b); free(x);
32      return;
33  }
34
35  void gauss_vp(
36              unsigned ess, unsigned fss, unsigned mbb, unsigned wgp,
37              vpfloat<unum, ess, fss, mbb> *A,
38              vpfloat<unum, ess, fss, mbb> *b,
39              vpfloat<unum, ess, fss, mbb> *x,
40              unsigned n
```

```
41              ) {
42      unsigned k, imin, i, j;
43      vpfloat<unum, ess, fss, mbb> valmin, tmp, p, sum; //allocate temporary
        VP variables
44
45      // IMPORTANT: the following operations are done invoving variables in
        memory of mbb bytes and inside the coprocessor on wgp*64 bits
46      for(k = 0; k < n-1; k++) { // for each column
47          /* Research the minimum not-null element (in absolute value
48          * in the k column with index i>=k                              */
49          valmin = A[k*n+k]; imin = k;
50          for(i = k+1; i < n; i++) { // for each line
51              if (valmin != 0) {
52                  if (my_abs(A[i*n+k]) < my_abs(valmin) && A[i*n+k] != 0) {
53                      valmin = A[i*n+k];
54                      imin = i;
55                  }
56              } else {
57                  valmin = A[i*n+k];
58                  imin = i;
59              }
60          }
61          /* Swap the elements of the matrix A and of the array b
62          * between the lines imin and k                                 */
63          for(j = 0; j < n; j++) {
64              tmp = A[imin*n+j];
65              A[imin*n+j] = A[k*n+j];
66              A[k*n+j] = tmp;
67          }
68          tmp = b[imin];
69          b[imin] = b[k];
70          b[k] = tmp;
71          // Reduce the matrix with the Gauss elimination method
72          for(i = k+1; i < n; i++) {
73              p = A[i*n+k]/A[k*n+k]; // the division is implemented in
        assembly on wgp*64 bits leveraging only the coprocessor internal
        registers
74              for(j = 0; j < n; j++)
75                  A[i*n+j] = A[i*n+j] - p*A[k*n+j]; // the MAC operation is
        done on WGP*46 bits on internal registers, the result is stored in
        memory on MBB bytes
76              b[i] = b[i] - p*b[k]; // the MAC operation is done on WGP*46
        bits on internal registers, the result is stored in memory on MBB bytes
77          }
78      }
79      // Solve the triangularized system
80      x[n-1] = b[n-1]/A[(n-1)*n+n-1]; // the division is implemented in
        assembly on wgp*64 bits leveraging only the coprocessor internal
        registers
81      for(i = n-2; i > -1; i--) {
82          sum = 0; // accumulator variable kept in the internal register, its
        content is not stored in memory
83          for(j = n-1; j > i; j--)
84              sum = sum + A[i*n+j]*x[j]; // the MAC operation is done on WGP
        *46 bits on internal registers
85          x[i] = (b[i] - sum)/A[i*n+i]; // both subtraction and division
        operations are done on WGP*46 bits on internal registers, the result is
        stored in memory on MBB bytes
86      }
87  }
```

./Chapters/Chapter6_gauss_vpfloat.c

A.3 The Gauss kernel for variable precision in C and inline assembly

This section presents an extract of the code used to perform the Gauss elimination algorithm in variable precision. With it, the results showed in Section 6.1 were obtained. Coprocessor operations are performed by using intrinsics in the code. Other C constructs were used to support all the code which does not belong to the variable precision coprocessor. Here below, there are some hints about how to read the listed code.

At line 101 there the code for the Gauss elimination solver begins. The function at line 77 does the configuration of the status registers. The Gauss solver firstly triangulates the input matrix: lines 120-235. Once that the system has been reduced, and a triangular matrix is obtained, the resolution of the system is very simple and straightforward.

For simplicity, we update variables in memory as soon as a computation step is performed. The reader can understand which operations are kept in the g-layer by looking to the consecutive operations kept within G-registers. For instance, in lines 266-274, it is possible to see that G0 implements an accumulator. It corresponds to the `sum` variable used in lines 82-85 of the algorithm listed in Section A.2. The content of this accumulator is kept in the g-layer until the loop at line 275 finishes.

```
1  #include <stdarg.h>
2  #include <stddef.h>
3
4  #include "printf.h"
5  #include "unum_rocc.h"
6  #include "asm_util.h"
7  #include "unum_basics.h"
8  #include "my_division_gbound.h"
9
10 #define NB_RAN 128
11
12 //parameters to set the type of the matrix
13 #define DHILBERT 1
14
15 #define N_HILBERT 150
16
17 double RAN[NB_RAN] = { ... }; // random values uniformly distributed between
       0 and 1
18
19 #define N N_HILBERT
20 double matrix[N*N];
21
22 double b[N];
23
24 unsigned long read_cycles(void); // read clock cycle counter: used to
       measure the code's performances
25
26 void hilbert_unum( ubound_t *H, uint64_t ess, uint64_t fss, uint64_t mbb,
       int n);
27 void choose_unum_array(ubound_t *b, uint64_t ess, uint64_t fss, uint64_t mbb
       , int n);
28 void gauss_unum(ubound_t *A, ubound_t *b, ubound_t *x, int n, uint64_t ess,
       uint64_t fss, uint64_t wgp, uint64_t mbb);
29 void MatVecMultSub_unum(int n,ubound_t *x,ubound_t *A,  ubound_t *b,
       ubound_t *R, uint64_t ess, uint64_t fss, uint64_t mbb);
30
31 double sq_root_double(double x);
32 double norm_res_unum_double(unum_t *res, int n, uint64_t mbb); //sqrt( sum((
       abs(res[i]))^2) )
33
```

```
34 ubound_t ubA  [(N*N) * SIZEOF_UBOUND(ESS_MAX, FSS_MAX)] ; // matrice A
35 ubound_t ubb  [(N  ) * SIZEOF_UBOUND(ESS_MAX, FSS_MAX)] ; // vecteur b
36 ubound_t ubx  [(N  ) * SIZEOF_UBOUND(ESS_MAX, FSS_MAX)] ; // vecteur des
       inconnues x
37 ubound_t ubres[(N  ) * SIZEOF_UBOUND(ESS_MAX, FSS_MAX)] ; // residu Ax-b
38 int main()
39 {
40     unsigned long startmain, endmain;
41     //preformance counter' values
42     unsigned long startmbb, endmbb, startub, endub;
43     //declare performance arrays
44     unsigned long arr_perf_unum[WGP_MAX+1];
45     int n ; // system size
46     uint64_t ess, fss, wgp, mbb;
47
48     printf("#START OF SIMULATION\n");
49     startmain = read_cycles(); //read counter
50
51     n = N;
52
53     ess=ESS_MAX;
54     for(fss=FSS_MAX; fss>=0; fss--){
55         for(wgp = 0; wgp < (WGP_MAX+1); wgp++){
56             for(mbb = MBB_MAX(ess,fss); mbb >= MBB_MIN(ess,fss); mbb--){
57                 startmbb = read_cycles(); //read counter
58                 my_gauss_algorithm(&startub, &endub, n, 1, mbb, wgp, ess,
       fss, ubA, ubb, ubx, ubres); // test computation with unum
59                 arr_perf_unum[wgp] = (endub-startub);
60                 endmbb = read_cycles(); //read counter
61             } //MBB
62         } //WGP
63     } // FSS
64     endmain = read_cycles(); //read counter
65     printf("#END OF SIMULATION\n");
66     return 0;
67 }
68
69 void my_gauss_algorithm(unsigned long *startcnt, unsigned long *endcnt, int
       n, uint64_t mbb, uint64_t wgp, uint64_t ess, uint64_t fss, ubound_t *A,
       ubound_t *b, ubound_t *x, ubound_t *res) //coprocessor's internal
       working precision
70 {
71     unsigned long startubin, endubin;
72     unsigned long startub, endub;
73     unsigned long startubout, endubout;
74     double norm_2_res;
75     int i, j ;
76
77     SUSR(get_next_USR(RND_TO_NEAREST,ess,fss,ess,fss,wgp,mbb)); //set
       coprocessor's status registers
78
79     //SELECT THE INTPUT MATRIX
80     startubin = read_cycles(); //read counter
81     hilbert_unum(A, ess, fss, mbb, n);
82     choose_unum_array(b,ess,fss,mbb,n);
83     endubin = read_cycles(); //read counter
84
85     //COMPUTE THE GAUSS ALGORITHM
86     startub = endubin; //read counter
87     gauss_unum(A,b,x,n,ess,fss,wgp,mbb);
88     endub  = read_cycles(); //read counter
89
90     //COMPUTE THE RESIDUAL
```

```
91      startubout = endub;
92      MatVecMultSub_unum(n,x,A,b,res,ess,fss,mbb);
93      norm_2_res = norm_res_unum_double(res, n, mbb); //sqrt( sum((abs(x[i]))
        ^2) )
94      endubout  = read_cycles(); //read counter
95
96      *startcnt = startub;
97      *endcnt   = endub;
98  }
99
100 /* Gauss elimination method */
101 void gauss_unum(ubound_t *A, ubound_t *b, ubound_t *x, int n, uint64_t ess,
        uint64_t fss, uint64_t wgp, uint64_t mbb)
102 {
103     typedef union
104     {
105         double   f_;
106         uint64_t i_;
107     } DoubleBits;
108     DoubleBits x_val;
109
110     int tmp0, tmp1, tmp2; // used for ubounds
111     int i, j, k ;
112     int imin ;
113     ubound_t valmin[mbb];
114     ubound_t abs_a[mbb];
115     ubound_t abs_min[mbb];
116     ubound_t p[mbb];
117
118     GFLOAT_D2G(G30, 0.0);
119
120     for(k = 0 ; k < n-1 ; k++)
121     {
122         /* First of all, we look for the minimum (non-zero) element */
123         /* in absolute value in the column k with index i greater   */
124         /* than or equal to k.                                      */
125
126         //valmin = A[k*n+k] ;
127         LDU(G1, A+((k*n+k)*mbb) );
128
129         //*/
130         STUL(valmin, G1);
131         //
132         imin = k ;
133         for(i = k+1 ; i < n ; i++)
134         {
135             //if (valmin != 0)
136             LDU(G1, valmin);
137             GCMP_NEQ(tmp0, G1, G30);
138             if (tmp0 != 0)
139             {
140                 my_abs_unum(abs_a, A+((i*n+k)*mbb));
141                 my_abs_unum(abs_min, valmin);
142
143                 LDU(G1, abs_a);
144                 LDU(G2, abs_min);
145
146                 GCMP_LT(tmp1, G1, G2);
147                 LDU(G3, A+((i*n+k)*mbb));
148                 GCMP_NEQ(tmp2, G3, G30);
149
150                 if ((tmp1!=0) && (tmp2 != 0))
151                 {
```

```
152                     //valmin = A[i*n+k] ;
153                     LDU(G4, A+((i*n+k)*mbb));
154                     STUL(valmin, G4);
155                     imin = i ;
156                 }
157             }
158             else
159             {
160                 //valmin = A[i*n+k] ;
161                 LDU(G4, A+((i*n+k)*mbb));
162                 STUL(valmin, G4);
163                 imin = i ;
164             }
165         }
166 
167         /* If the minimum element is null, we can deduce    */
168         /* that the matrix is singular. The pogram is then  */
169         /* interrupted.                                     */
170 
171         //if (valmin == 0.)
172         LDU(G1, valmin);
173         GCMP_EQ(tmp1, G30, G1);
174 
175         /* If the matrix is not singular, we invert the      */
176         /* elements of the line imax with the elements of    */
177         /* the line k. We do the same with the array b       */
178 
179         for(j = 0 ; j < n ; j++)
180         {
181             //tump1 = A[imin*n+j] ;
182             //A[imin*n+j] = A[k*n+j] ;
183             //A[k*n+j] = tump1 ;
184             LDU(G1, A+((imin*n+j)*mbb));
185             LDU(G2, A+((k*n+j)*mbb));
186             STUL(A+((k*n+j)*mbb), G1);
187             STUL(A+((imin*n+j)*mbb), G2);
188         }
189 
190         //tump2 = b[imin] ;
191         //b[imin] = b[k] ;
192         //b[k] = tump2 ;
193         LDU(G2, b+((imin)*mbb));
194         LDU(G3, b+((k)*mbb));
195         STUL(b+((k)*mbb), G2);
196         STUL(b+((imin)*mbb), G3);
197 
198         /* The matrix is reduced by the Gauss elimination */
199         /* method.                                        */
200         for(i = k+1 ; i < n ; i++)
201         {
202             //p = A[i*n+k]/A[k*n+k] ;
203             LDU(G5, A+((i*n+k)*mbb));
204             LDU(G6, A+((k*n+k)*mbb));
205 
206             /* unum division
207              * It computes G7 = G5 / G6 using the Newton-Raphson
    approximation method and using the precision specified by the wgp
    parameter
208              * It uses also G0, G1, G2, G3, G4, G5, G6, G7, G8, G9, G10
209              */
210             my_division_unum (wgp+1);
211 
212             STUL(p, G7);
```

```
213             MOV_G2G(G20, G7); //p
214
215             for(j = 0 ; j < n ; j++)
216             {
217                 //A[i*n+j] = A[i*n+j] - p*A[k*n+j] ;
218                 LDU(G22, A+((k*n+j)*mbb));
219                 LDU(G21, A+((i*n+j)*mbb));
220                 GMUL(G23, G20, G22); //p*A[k*n+j]
221                 GSUB(G24, G21, G23); //A[i*n+j] - p*A[k*n+j]
222
223                 STUL(A+((i*n+j)*mbb), G24);
224             }
225
226             //b[i] = b[i] - p*b[k] ;
227             LDU(G22, b+((k)*mbb)); //b[k]
228             LDU(G21, b+((i)*mbb)); //b[i]
229             GMUL(G23, G20, G22); //p*b[k]
230             GSUB(G24, G21, G23); //b[i] - p*b[k]
231
232             STUL(b+((i)*mbb), G24);
233
234         }
235     }
236
237     /* We check that the matrix is still not singular. */
238     /* If it is, we interrupt the program.             */
239
240     //if (A[(n-1)*n+n-1] == 0)
241     LDU(G1, A+(((n-1)*n+n-1)*mbb));
242     GCMP_EQ(tmp1, G30, G1);
243     if (tmp1 == 0.)
244     {
245         printf("\n\n\nWarning! the matrix is singular!\n\n\n") ;
246     }
247
248     /* Once the system has been reduced, a top triangular matrix is */
249     /* obtained and the resolution of the system is very simple.    */
250
251     //x[n-1] = b[n-1]/A[(n-1)*n+n-1] ;
252     LDU(G5, b+((n-1)*mbb)); //b[n-1]
253     LDU(G6, A+(((n-1)*n+n-1)*mbb)); //A[(n-1)*n+n-1]
254
255     /* g-bound division
256      * It computes G7 = G5 / G6 using the Newton-Raphson approximation
        method and using the precision specified by the wgp parameter
257      * It uses also G0, G1, G2, G3, G4, G5, G6, G7, G8, G9, G10
258      */
259     my_division_unum  (
260                           wgp+1 // working g-layer precision (0->64bits,
        1->128 bits, ..., 7->512bits of mantissa)
261                       );
262     STUL(x+((n-1)*mbb), G7); //x[n-1]
263
264     for(i = n-2 ; i > -1 ; i--)
265     {
266         GFLOAT_D2G(G0, 0.0); //sum = 0 ;
267
268         for(j = n-1 ; j > i ; j--)
269         {
270             //sum = sum + A[i*n+j]*x[j] ;
271             LDU(G1, A+((i*n+j)*mbb)); //A[i*n+j]
272             LDU(G2, x+((j)*mbb)); //x[j]
273             GMUL(G3, G1, G2);
```

```
274             GADD(G0, G0, G3);
275         }
276         //x[i] = (b[i] - sum)/A[i*n+i] ;
277         LDU(G4, b+((i)*mbb)); //b[i]
278         LDU(G6, A+((i*n+i)*mbb)); //A[i*n+i]
279         GSUB(G5, G4, G0); //b[i] - sum
280 
281         /* g-bound division
282          * It computes G7 = G5 / G6 using the Newton-Raphson approximation method and using the precision specified by the wgp parameter
283          * It uses also G0, G1, G2, G3, G4, G5, G6, G7, G8, G9, G10
284          */
285         my_division_unum  ( wgp+1 );
286 
287         STUL(x+((i)*mbb), G7); //x[i]
288     }
289 }
290 
291 /* Choose the elements of the b array */
292 void choose_unum_array(ubound_t *b, uint64_t ess, uint64_t fss, uint64_t mbb, int n)
293 {
294     typedef union
295     {
296         double   f_;
297         uint64_t i_;
298     } DoubleBits;
299     DoubleBits b_val;
300     int i ;
301 
302     for(i = 0 ; i < n ; i++)
303     {
304         //b[i]=RAN[NB_RAN-1-(i%N)];
305         b_val.f_=RAN[NB_RAN-1-(i%NB_RAN)];
306         GFLOAT_D2G(G1, b_val.i_ );
307         STUL(b+(i*mbb) , G1);
308     }
309 }
310 
311 void my_abs_unum(unum_t *res, unum_t *x){
312     typedef union
313     {
314         double   f_;
315         uint64_t i_;
316     } DoubleBits;
317     DoubleBits x_val;
318 
319     uint64_t cmp_nan;
320     int l_lt_0;
321 
322     LDU(G1, x );
323 
324     x_val.f_ = 0.0;
325     GFLOAT_D2G(G0, x_val.i_);
326     GCMP(cmp_nan, G1, G0);
327     GCMP_LL_LT(l_lt_0, G1, G0);
328 
329     if ( (cmp_nan & MASK_CMP_FLAGS_NAN) != 0 ){
330         STUL(res, G1);
331     } else {
332         if (l_lt_0==0){ //G1 > 0
333             MOV_G2G(G2,G1); // output the input value as it is
334         } else { //left ep <= 0
```

```
335             GSUB(G2, G0, G1); //negate G1
336         }
337         STUL(res, G2);
338     }
339 }
340
341 void MatVecMultSub_unum(int n, ubound_t *x, ubound_t *A, ubound_t *b,
        ubound_t *R, uint64_t ess, uint64_t fss, uint64_t mbb) {
342     // R=Ax-b
343     // on alloue la taille max
344     int li, col;
345     //ubound_t sum;
346     for (li=0; li<n; li++) {
347         GFLOAT_D2G(G0, 0.0); //sum=0.0;
348         for (col=0; col<n; col++) {
349             //sum+=A[li*n+col]*x[col];
350             LDU(G1, A+((li*n+col)*mbb)); //A[li*n+col]
351             LDU(G2, x+((col)*mbb)); //x[col]
352             GMUL(G3, G1, G2); //A[li*n+col]*x[col]
353             GADD(G0, G0, G3); //sum
354         }
355         //R[li]=sum-b[li];
356         LDU(G4, b+((li)*mbb)); //b[li]
357         GSUB(G5, G0, G4);
358         STUL(R+((li)*mbb), G5); //R[li]
359     }
360 }
361
362 double sq_root_double(double x) {
363     double rt = 1, ort = 0;
364     if (x>=0 && x==x){ //X is positive and it is not NaN
365         while(ort!=rt)
366         {
367             ort = rt;
368             rt = ((x/rt) + rt) / 2;
369         }
370         return rt;
371     } else { //result is NaN
372         return (0.0/0.0); // NaN
373     }
374 }
375 double norm_res_unum_double(unum_t *res, int n, uint64_t mbb) //sqrt( sum((
        abs(res[i]))^2) )
376 {
377     //internal variables
378     unsigned i;
379     //accumulator
380     double sum=0;
381     //loaded values
382     typedef union
383     {
384         double   f_;
385         uint64_t i_;
386     } DoubleBits;
387     DoubleBits int_var;
388
389     //MBB support
390     uint64_t prev_mbb;
391     LUSR_MBB(prev_mbb);
392     SUSR_MBB(mbb);
393
394     for (i=0; i<n; i++){
395         LDU(G0, res+(i*mbb) );
```

```
396         GFLOAT_GL2D(int_var.i_ , G0);
397         sum += (int_var.f_ * int_var.f_ );
398     }
399     SUSR_MBB(prev_mbb);
400     return(sq_root_double(sum));
401 }
402
403 void hilbert_unum (
404                     ubound_t *H, // output matrix
405                     uint64_t  ess,
406                     uint64_t  fss,
407                     uint64_t  mbb,
408                     int       n  // matrix size #rows=#columns=n
409                 )
410 {
411     typedef union
412     {
413         double   f_;
414         uint64_t i_;
415     } DoubleBits;
416     DoubleBits val;
417     double tmp;
418     int i,j;
419     for(i=0;i<n;i++){
420         for(j=0;j<n;j++){
421             tmp = (double)1.0/((i+1)+(j+1)-1);
422             val.f_ = tmp;
423             GFLOAT_D2G(G1, val.i_);
424             STUL(H+((i*n+j)*mbb), G1);
425         }
426     }
427 }
428
429 unsigned long read_cycles(void)
430 {
431     unsigned long cycles;
432     asm volatile ("rdcycle %0" : "=r" (cycles));
433     return cycles;
434 }
```

./Chapters/Chapter6_gauss_assembly.c

Appendix B

Source code Experiment 2

B.1 Complete code for Conjugate Gradient

```
int cg_vp(double *x, double *A, double *b, int m, int n,
          int precision, double tolerance) {
    vpfloat <unum, 4, get_fss(precision), get_mbb(precision)> *r_k =
        (vpfloat <unum, 4, get_mbb(precision)>*)malloc(n*sizeof(vpfloat <
    unum, 4, get_fss(precision), get_mbb(precision)>));
    copy_vector_vp_d(precision, r_k, b, n);
    vpfloat <unum, 4, get_fss(precision), get_mbb(precision)> *p_k =
        (vpfloat <unum, 4, get_fss(precision), get_mbb(precision)>*)malloc(n
    *sizeof(vpfloat <unum, 4, get_fss(precision), get_mbb(precision)>));
    copy_vector_vp_d(precision, p_k, b, n);
    vpfloat <unum, 4, get_fss(precision), get_mbb(precision)> *Ap_k =
        (vpfloat <unum, 4, get_fss(precision), get_mbb(precision)>*)malloc(n
    *sizeof(vpfloat <unum, 4, get_fss(precision), get_mbb(precision)>));
    zeros_vp(precision, Ap_k, n);
    vpfloat <unum, 4, get_fss(precision), get_mbb(precision)> alpha;
    vpfloat <unum, 4, get_fss(precision), get_mbb(precision)> *x_k =
        (vpfloat <unum, 4, get_fss(precision), get_mbb(precision)>*)malloc(n
    *sizeof(vpfloat <unum, 4, get_fss(precision), get_mbb(precision)>));
    //set_ess_fss_mbb_rnd(ess=4, get_fss(precision), get_mbb(precision), rnd
    =RTN); // implicit set of the ESS, FSS, and MBB status registers
    set_wgp(get_wgp(precision)); // set the WGP status regster

    zeros_vp(precision, x_k, n);
    vpfloat <unum, 4, get_fss(precision), get_mbb(precision)> rs_next = 0.0v
    ;
    // rs = rk'*rk
    vpfloat <unum, 4, get_fss(precision), get_mbb(precision)> rs = vdot(
    precision, n, r_k, r_k);
    int nbiter;

    for (nbiter = 1; nbiter < m*3000; ++nbiter) {
        vgemvd(precision, m, n , 1.0, A, p_k, 0.0, Ap_k);
        alpha = rs/vdot(precision, n, p_k, Ap_k);

        vaxpy(precision, n, alpha, p_k, x_k); //x = x + alpha*p
        vaxpy(precision, n, -alpha, Ap_k, r_k); //r = r - alpha *Ap
        rs_next = vdot(precision, n, r_k, r_k); // rk+1

        if (sqrt((double)rs_next) < tolerance) {
            copy_vector_d_vp(precision, x, x_k, n);
            break;
        }
        vscal(precision, n, rs_next/rs, p_k);
        vaxpy(precision, n, 1.0, r_k, p_k);
        rs_next = rs;
    }
```

```
40
41      free(r_k); free(p_k); free(x_k); // free allocated elements
42
43      return nbiter;
44  }
```

Listing B.1: complete code for conjugate gradient

B.2 C code implementation of the Conjugate Gradient

```
1   #include <stdarg.h>
2   #include <stddef.h>
3
4   #include "printf.h"
5   #include "unum_rocc.h"
6   #include "asm_util.h"
7   #include "unum_basics.h"
8   #include "my_division_gbound.h"
9
10  #define THRESHOLD_START 1.0e-3
11  #define C_MAX_ITERATIONS 9999
12
13  #define INPUT_IS_MATRIX_MARKET
14
15  #define NB_RAN 128
16  double RAN[NB_RAN] = { ... }; // random values uniformly distributed between
        0 and 1
17
18  double error = 0.0;
19  double total_error = 0.0;
20
21  #ifdef __cplusplus
22  extern "C" {
23  #endif
24
25  #define NB_SAMPLES 237
26  double matrix[NB_SAMPLES][NB_SAMPLES];
27  double b[NB_SAMPLES];
28
29  void fill_matrix_market_NOS1(double matrix[237][237], unsigned size);
30  unsigned long read_cycles(void);
31  void generate_vector(double *b ,int n);
32  double sq_root_double(double x);
33  double norm_res_unum_double(unum_t *res, int n, uint64_t mbb); //sqrt( sum((
        abs(res[i]))^2) )
34  void matrix_div_scalar(double *matrix_result, double* matrix, double
        denominator, unsigned size);
35  void transpose_matrix(double *matrixPrime, double *matrix, unsigned size);
36  void randomSDPMatrix(double *matrix, unsigned size);
37  void zeros(double *matrix, unsigned dim1, unsigned dim2 = 1);
38  void zeros(unum_t *matrix, unsigned comp_mbb, unsigned dim1, unsigned dim2 =
        1);
39  void copy_matrix_vector(unum_t *new_matrix, unum_t *matrix, unsigned
        comp_mbb, unsigned dim1, unsigned dim2 = 1);
40  template<typename Tdst, typename Tsrc>
41  void copy_matrix_vector(Tdst *new_matrix, Tsrc *matrix, unsigned dim1,
        unsigned dim2 = 1);
42  void copy_matrix_vector_double(double *new_matrix, double *matrix, unsigned
        dim1, unsigned dim2 = 1);
```

```
43 void copy_matrix_vector_double(double *new_matrix, unum_t *matrix, unsigned
       comp_mbb, unsigned dim1, unsigned dim2 = 1);
44 void vector_addition(unum_t *res, unum_t *a, unum_t *b, unsigned comp_mbb,
       unsigned size);
45 void vector_subtraction(unum_t *res, unum_t *a, unum_t *b, unsigned comp_mbb
       , unsigned size);
46 void vector_subtraction_double(unum_t *res, double *a, unum_t *b, unsigned
       comp_mbb, unsigned size);
47 void matrix_multiplication(double *matrix_result, double *matrix, double *
       matrixPrime, unsigned dim1, unsigned dim2, unsigned dim3);
48 void matrix_multiplication(unum_t *matrix_result, unum_t *matrix, unum_t *
       matrixPrime, unsigned comp_mbb, unsigned dim1, unsigned dim2, unsigned
       dim3);
49 void matrix_vector_multiplication(double *vector_result, double* matrix,
       double *vector, unsigned size);
50 void matrix_vector_multiplication(unum_t *vector_result, unum_t* matrix,
       unum_t *vector, unsigned comp_mbb, unsigned size);
51 void matrix_vector_multiplication(unum_t *vector_result, double* matrix,
       unum_t *vector, unsigned comp_mbb, unsigned size) {
52 template<typename T>
53 void matrix_vector_multiplication(T *vector_result, T* matrix, T *vector,
       unsigned size) {
54 void matrix_vector_mult_scalar(double *matrix_result, double* matrix, double
        multipler, unsigned dim1, unsigned dim2);
55 void matrix_vector_mult_scalar(unum_t *matrix_result, unum_t* matrix, unum_t
        *multipler, unsigned comp_mbb, unsigned dim1, unsigned dim2);
56 void matrix_vector_mult_scalar(unum_t *matrix_result, double* matrix, unum_t
        *multipler, unsigned comp_mbb, unsigned dim1, unsigned dim2);
57 void matrix_vector_mult_scalar(double *matrix_result, double* matrix, unum_t
        *multipler, unsigned comp_mbb, unsigned dim1, unsigned dim2);
58 typedef struct {
59     double error;
60     unsigned long latency_cc;
61     unsigned iterations;
62     unsigned ess;
63     unsigned fss;
64     unsigned wgp;
65 } out_experiment_t;
66 out_experiment_t cg_unum(double *A, double *b, unsigned size, unsigned
       comp_wgp, unsigned comp_ess, unsigned comp_fss, unsigned comp_mbb,
       double threshold, unsigned max_iterations) {
67 out_experiment_t cg_unum_random_matrix(unsigned size, unsigned comp_wgp,
       unsigned comp_ess, unsigned comp_fss, unsigned comp_mbb, double
       threshold, unsigned max_iterations);
68 out_experiment_t cg_unum_tb(unsigned ess, unsigned mbb, double threshold,
       unsigned max_iterations);
69
70 double matrixPrime[NB_SAMPLES][NB_SAMPLES];
71 double matrix_mult[NB_SAMPLES][NB_SAMPLES];
72
73 int main() {
74     unsigned tb_ess=ESS_MAX;
75     out_experiment_t out_exp;
76     double error;
77     double threshold; // = (double)THRESHOLD_START;
78     printf("ENTERING THE MAIN!!!!!!!\n");
79     for (threshold = (double)THRESHOLD_START; threshold>=1e-10; threshold=
       threshold/10.0 ){
80         out_exp.iterations = 0;
81         tb_ess=3;
82         //unsigned tb_mbb=MBB_MAX(ESS_MAX, FSS_MAX);
83         error=0.0;
```

```
84          for(unsigned tb_mbb=MBB_MAX(ESS_MAX, FSS_MAX); (tb_mbb>= MBB_MIN(
        ESS_MAX, 1)) && (error<threshold) && (error==error) ; tb_mbb--){ //
        downto MBB_MIN until the error is met (<th) or is NaN (error!=error)
85
86              fill_matrix_market_NOS1(matrix, NB_SAMPLES);
87              generate_vector(b, NB_SAMPLES);
88              out_exp=cg_unum_tb(tb_ess, tb_mbb, threshold, C_MAX_ITERATIONS);
89              error = out_exp.error;
90          } // for loop MBB
91      } // for loop threshold
92      printf("FINISHING THE MAIN!!!!!!!\n");
93      return 0;
94  }
95
96  out_experiment_t cg_unum_tb(unsigned ess, unsigned mbb, double threshold,
        unsigned max_iterations) {
97      unsigned fss, remaining_bits, wgp;
98      out_experiment_t out_exp;
99
100     // compute FSS min accoring to MBB
101     // get parameters starting from MBB
102     //      size     - ESS - s - u - e - fss)
103     if ( ( (mbb*8) - ess - 1 - 1 - 1 - 1  ) <= 2 ){
104         fss = 1;
105     }
106     //           size     - ESS - s - u - e - fss)
107     else if ( ( (mbb*8) - ess - 1 - 1 - 1 - 2  ) <= 4 ){
108         fss = 2;
109     }
110     //           size     - ESS - s - u - e - fss)
111     else if ( ( (mbb*8) - ess - 1 - 1 - 1 - 3  ) <= 8 ){
112         fss = 3;
113     }
114     //           size     - ESS - s - u - e - fss)
115     else if ( ( (mbb*8) - ess - 1 - 1 - 1 - 4  ) <= 16 ){
116         fss = 4;
117     }
118     //           size     - ESS - s - u - e - fss)
119     else if ( ( (mbb*8) - ess - 1 - 1 - 1 - 5  ) <= 32 ){
120         fss = 5;
121     }
122     //           size     - ESS - s - u - e - fss)
123     else if ( ( (mbb*8) - ess - 1 - 1 - 1 - 6  ) <= 64 ){
124         fss = 6;
125     }
126     //           size     - ESS - s - u - e - fss)
127     else if ( ( (mbb*8) - ess - 1 - 1 - 1 - 7  ) <= 128 ){
128         fss = 7;
129     }
130     //           size     - ESS - s - u - e - fss)
131     else if ( ( (mbb*8) - ess - 1 - 1 - 1 - 8  ) <= 256 ){
132         fss = 8;
133     }
134     //               size     - ESS - s - u - e - fss)
135     else { //if ( ( (mbb*8) - ess - 1 - 1 - 1 - 9  ) <= 512 )
136         fss = 9;
137     }
138
139     // Compute minimal WGP
140     remaining_bits=(mbb*8) -ess -1 -1 -1 -fss;
141     if (remaining_bits<(1*64)){
142         wgp=0;
143     }else if (remaining_bits>=(1*64) && remaining_bits<(2*64)){
```

```
144         wgp=1;
145     }else if (remaining_bits>=(2*64) && remaining_bits<(3*64)){
146         wgp=2;
147     }else if (remaining_bits>=(3*64) && remaining_bits<(4*64)){
148         wgp=3;
149     }else if (remaining_bits>=(4*64) && remaining_bits<(5*64)){
150         wgp=4;
151     }else if (remaining_bits>=(5*64) && remaining_bits<(6*64)){
152         wgp=5;
153     }else if (remaining_bits>=(6*64) && remaining_bits<(7*64)){
154         wgp=6;
155     }else if (remaining_bits>=(7*64)){
156         wgp=7;
157     }
158
159     SUSR(get_next_USR(RND_TO_NEAREST, ess,fss,ess,fss, wgp, mbb));
160
161     out_exp=cg_unum_random_matrix(NB_SAMPLES, wgp, ess, fss, mbb, threshold,
          max_iterations);
162
163     return out_exp;
164 }
165
166 out_experiment_t cg_unum_random_matrix(unsigned size, unsigned comp_wgp,
        unsigned comp_ess, unsigned comp_fss, unsigned comp_mbb, double
        threshold, unsigned max_iterations) {
167
168     unsigned j = 0;
169     total_error = 0.0;
170     out_experiment_t output;
171
172     double m[NB_SAMPLES][NB_SAMPLES];
173     copy_matrix_vector((double*)m, (double*)matrix, NB_SAMPLES, NB_SAMPLES);
174     randomSDPMatrix((double*)m, NB_SAMPLES);
175     output= cg_unum((double*)m, b, NB_SAMPLES, comp_wgp, comp_ess, comp_fss,
          comp_mbb, threshold, max_iterations);
176     return output;
177 }
178
179 out_experiment_t cg_unum(double *A, double *b, unsigned size, unsigned
        comp_wgp, unsigned comp_ess, unsigned comp_fss, unsigned comp_mbb,
        double threshold, unsigned max_iterations) {
180     // error computing
181     unsigned iteration;
182     double norm_err=1.0;
183     double x[NB_SAMPLES];
184
185     unum_t bprime[NB_SAMPLES * SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
186     unum_t r_k[NB_SAMPLES * SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
187     unum_t p_k[NB_SAMPLES * SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
188     unum_t tmp[NB_SAMPLES * SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
189     unum_t alpha_beta_k[SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
190     unum_t x_k[NB_SAMPLES * SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
191     unum_t x_next[NB_SAMPLES * SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
192     unum_t r_next[NB_SAMPLES * SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
193     unum_t p_next[NB_SAMPLES * SIZEOF_UNUM(ESS_MAX, FSS_MAX)];
194
195
196     out_experiment_t out_exp;
197     unsigned long start = read_cycles();
198
199     zeros(x_k, comp_mbb, NB_SAMPLES);
200
```

```
201     matrix_vector_multiplication((unum_t*)tmp, A, (unum_t*)x_k, comp_mbb,
        size);
202     vector_subtraction_double((unum_t*)r_k, b, (unum_t*)tmp, comp_mbb, size)
        ;
203
204     copy_matrix_vector(p_k, r_k, comp_mbb, size);
205
206     for (iteration = 0; ((iteration < max_iterations) && (norm_err >
        threshold)); ++iteration) {
207         matrix_vector_multiplication((unum_t*)tmp, A, (unum_t*)p_k, comp_mbb
        , size);
208
209         matrix_multiplication((unum_t*)tmp, (unum_t*)p_k, (unum_t*)tmp,
        comp_mbb, 1, size, 1);
210
211         LDU(G5, (unum_t*)tmp);
212         STUL(alpha_beta_k, G5);
213         matrix_multiplication((unum_t*)tmp, (unum_t*)r_k, (unum_t*)r_k,
        comp_mbb, 1, size, 1);
214
215 //      alpha_beta_k = tmp[0][0]/alpha_beta_k;
216         LDU(G5, (unum_t*)tmp);
217         LDU(G6, alpha_beta_k);
218         STUL(alpha_beta_k, G7);
219
220         matrix_vector_mult_scalar((unum_t*)tmp, (unum_t*)p_k, (unum_t*)
        alpha_beta_k, comp_mbb, size, 1);
221         vector_addition((unum_t*)x_next, (unum_t*)x_k, (unum_t*)tmp,
        comp_mbb, size);
222
223         matrix_vector_multiplication((unum_t*)tmp, A, (unum_t*)p_k, comp_mbb
        , size);
224         matrix_vector_mult_scalar((unum_t*)tmp, (unum_t*)tmp, (unum_t*)
        alpha_beta_k, comp_mbb, size, 1);
225
226         vector_subtraction((unum_t*)r_next, (unum_t*)r_k, (unum_t*)tmp,
        comp_mbb, size);
227
228         matrix_multiplication((unum_t*)tmp, (unum_t*)(unum_t*)r_k, r_k,
        comp_mbb, 1, size, 1);
229 //      alpha_beta_k = tmp[0][0];
230         LDU(G5, (unum_t*)tmp);
231         STUL(alpha_beta_k, G5);
232         matrix_multiplication((unum_t*)tmp, (unum_t*)(unum_t*)r_next, r_next
        , comp_mbb, 1, size, 1);
233 //      alpha_beta_k = tmp[0][0]/alpha_beta_k;
234         LDU(G5, (unum_t*)tmp);
235         LDU(G6, alpha_beta_k);
236         STUL(alpha_beta_k, G7);
237         matrix_vector_mult_scalar((unum_t*)tmp, (unum_t*)p_k, (unum_t*)
        alpha_beta_k, comp_mbb, size, 1);
238         vector_addition((unum_t*)p_next, (unum_t*)r_next, (unum_t*)tmp,
        comp_mbb, size);
239
240         copy_matrix_vector(p_k, p_next, comp_mbb, size);
241         copy_matrix_vector(r_k, r_next, comp_mbb, size);
242         copy_matrix_vector(x_k, x_next, comp_mbb, size);
243
244         // compute computational error
245         set_coprocessor_wgp(WGP_MAX);
246         matrix_vector_multiplication(bprime, A, x_k, comp_mbb, size);
247         vector_subtraction_double(bprime, b, bprime, comp_mbb, size);
```

```
248         norm_err = norm_res_unum_double(bprime, size, comp_mbb); //sqrt( sum
    ((abs(res[i]))^2) )
249         set_coprocessor_wgp(comp_wgp);
250     }
251
252     copy_matrix_vector_double(x, x_k, comp_mbb, size);
253
254     unsigned long end = read_cycles();
255     unsigned long diff = end - start;
256
257     out_exp.error      = norm_err;
258     out_exp.latency_cc = diff;
259     out_exp.iterations = iteration;
260     out_exp.ess        = comp_ess;
261     out_exp.fss        = comp_fss;
262     out_exp.wgp        = comp_wgp;
263
264     return out_exp;
265 }
266
267 void matrix_vector_multiplication(double *vector_result, double* matrix,
    double *vector, unsigned size) {
268     for(unsigned i = 0; i < size; ++i) {
269         double sum = 0.0;
270         for (unsigned j = 0; j < size; ++j) {
271             sum += matrix[i*size + j]*vector[j];
272         }
273         vector_result[i] = sum;
274     }
275 }
276
277 void matrix_vector_multiplication(unum_t *vector_result, unum_t* matrix,
    unum_t *vector, unsigned comp_mbb, unsigned size) {
278     typedef union
279     {
280         double   f_;
281         uint64_t i_;
282     } DoubleBits;
283     DoubleBits x_val;
284     x_val.f_ = 0.0;
285     for(unsigned i = 0; i < size; ++i) {
286 //        double sum = 0.0;
287         GFLOAT_D2G(G0, x_val.i_);
288         for (unsigned j = 0; j < size; ++j) {
289 //            sum += matrix[i*size + j]*vector[j];
290             LDU(G1, matrix+((i*size + j)*comp_mbb));
291             LDU(G2, vector+(j*comp_mbb));
292             GMUL(G3, G1, G2);
293             GADD(G0, G0, G3);
294         }
295 //        vector_result[i] = sum;
296         STUL(vector_result + (i*comp_mbb), G0);
297     }
298 }
299
300 void matrix_vector_multiplication(unum_t *vector_result, double* matrix,
    unum_t *vector, unsigned comp_mbb, unsigned size) {
301     typedef union
302     {
303         double   f_;
304         uint64_t i_;
305     } DoubleBits;
306     DoubleBits x_val;
```

```
307     DoubleBits y_val;
308     y_val.f_ = 0.0;
309 
310     for(unsigned i = 0; i < size; ++i) {
311 //        double sum = 0.0;
312         GFLOAT_D2G(G0, y_val.i_);
313         for (unsigned j = 0; j < size; ++j) {
314 //            sum += matrix[i*size + j]*vector[j];
315             x_val.f_ = matrix[i*size + j];
316             GFLOAT_D2G(G1, x_val.i_);
317             LDU(G2, vector+((j)*comp_mbb));
318             GMUL(G3, G1, G2);
319             GADD(G0, G0, G3);
320         }
321 //        vector_result[i] = sum;
322         STUL(vector_result + (i*comp_mbb), G0);
323     }
324 }
325 
326 template<typename T>
327 void matrix_vector_multiplication(T *vector_result, T* matrix, T *vector,
    unsigned size) {
328     for(unsigned i = 0; i < size; ++i) {
329         T sum = 0.0;
330         for (unsigned j = 0; j < size; ++j) {
331             sum += matrix[i*size + j]*vector[j];
332         }
333         vector_result[i] = sum;
334     }
335 }
336 
337 void vector_subtraction_double(unum_t *res, double *a, unum_t *b, unsigned
    comp_mbb, unsigned size) {
338     typedef union
339     {
340         double   f_;
341         uint64_t i_;
342     } DoubleBits;
343     DoubleBits x_val;
344 
345     for (unsigned i = 0; i < size; ++i) {
346         x_val.f_= a[i];
347         GFLOAT_D2G(G0, x_val.i_);
348         LDU(G1, b+(i*comp_mbb));
349         GSUB(G2, G0, G1);
350         STUL(res + (i*comp_mbb), G2);
351     }
352 }
353 
354 void matrix_multiplication(double *matrix_result,
355                            double *matrix,
356                            double *matrixPrime,
357                            unsigned dim1,
358                            unsigned dim2,
359                            unsigned dim3) {
360     double sum = 0.0;
361     for (unsigned i = 0; i < dim1; ++i) {
362         for (unsigned j = 0; j < dim3; ++j) {
363             for (unsigned k = 0; k < dim2; ++k) {
364                 sum = sum + matrix[i*dim2 + k]* matrixPrime[k*dim3 + j];
365             }
366             matrix_result[i*dim3 + j] = sum;
367             sum = 0.0;
```

```
368         }
369     }
370 }
371
372 void matrix_multiplication(unum_t *matrix_result,
373                            unum_t *matrix,
374                            unum_t *matrixPrime,
375                            unsigned comp_mbb,
376                            unsigned dim1,
377                            unsigned dim2,
378                            unsigned dim3) {
379     typedef union
380     {
381         double   f_;
382         uint64_t i_;
383     } DoubleBits;
384     DoubleBits x_val;
385
386     //sum = 0.0;
387     x_val.f_ = 0.0;
388     GFLOAT_D2G(G0, x_val.i_);
389     for (unsigned i = 0; i < dim1; ++i) {
390         for (unsigned j = 0; j < dim3; ++j) {
391             for (unsigned k = 0; k < dim2; ++k) {
392                 //sum = sum + matrix[i*dim2 + k]* matrixPrime[k*dim3 + j];
393                 LDU(G1, matrix+((i*dim2 + k)*comp_mbb));
394                 LDU(G2, matrixPrime+((k*dim3 + j)*comp_mbb));
395                 GMUL(G3, G1, G2);
396                 GADD(G0, G0, G3);
397             }
398             //matrix_result[i*dim3 + j] = sum;
399             STUL(matrix_result + ((i*dim3 + j)*comp_mbb), G0);
400             //sum = 0.0;
401             GFLOAT_D2G(G0, x_val.i_);
402         }
403     }
404 }
405
406 void matrix_vector_mult_scalar(double *matrix_result, double* matrix, double
      multipler, unsigned dim1, unsigned dim2) {
407     for(unsigned i = 0; i < dim1; ++i) {
408         for (unsigned j = 0; j < dim2; ++j) {
409             matrix_result[i*dim2 + j] = matrix[i*dim2 + j]*multipler;
410         }
411     }
412 }
413
414 void matrix_vector_mult_scalar(unum_t *matrix_result, unum_t* matrix, unum_t
      *multipler, unsigned comp_mbb, unsigned dim1, unsigned dim2) {
415     LDU(G2, multipler);
416     for(unsigned i = 0; i < dim1; ++i) {
417         for (unsigned j = 0; j < dim2; ++j) {
418             //matrix_result[i*dim2 + j] = matrix[i*dim2 + j]*multipler;
419             LDU(G1, matrix+((i*dim2 + j)*comp_mbb));
420             GMUL(G0, G1, G2);
421             STUL(matrix_result + ((i*dim2 + j)*comp_mbb), G0);
422         }
423     }
424 }
425
426 void matrix_vector_mult_scalar(unum_t *matrix_result, double* matrix, unum_t
      *multipler, unsigned comp_mbb, unsigned dim1, unsigned dim2) {
427     typedef union
```

```
428     {
429         double   f_;
430         uint64_t i_;
431     } DoubleBits;
432     DoubleBits x_val;
433 
434     LDU(G2, multipler);
435     for(unsigned i = 0; i < dim1; ++i) {
436         for (unsigned j = 0; j < dim2; ++j) {
437             //matrix_result[i*dim2 + j] = matrix[i*dim2 + j]*multipler;
438             x_val.f_ = matrix[i*dim2 + j];
439             GFLOAT_D2G(G1, x_val.i_);
440             GMUL(G0, G1, G2);
441             STUL(matrix_result + ((i*dim2 + j)*comp_mbb), G0);
442         }
443     }
444 }
445 
446 void matrix_vector_mult_scalar(double *matrix_result, double* matrix, unum_t
      *multipler, unsigned comp_mbb, unsigned dim1, unsigned dim2) {
447     typedef union
448     {
449         double   f_;
450         uint64_t i_;
451     } DoubleBits;
452     DoubleBits x_val;
453 
454     LDU(G2, multipler);
455     for(unsigned i = 0; i < dim1; ++i) {
456         for (unsigned j = 0; j < dim2; ++j) {
457 //             matrix_result[i*dim2 + j] = matrix[i*dim2 + j]*multipler;
458             x_val.f_ = matrix[i*dim2 + j];
459             GFLOAT_D2G(G1, x_val.i_);
460             GMUL(G0, G1, G2);
461             GFLOAT_GL2D(x_val.i_ , G0);
462             matrix_result[i*dim2 + j] = x_val.f_;
463         }
464     }
465 }
466 
467 void vector_addition(unum_t *res, unum_t *a, unum_t *b, unsigned comp_mbb,
     unsigned size) {
468     for (unsigned i = 0; i < size; ++i) {
469 //         res[i] = a[i] + b[i];
470         LDU(G0, a+(i*comp_mbb));
471         LDU(G1, b+(i*comp_mbb));
472         GADD(G2, G0, G1);
473         STUL(res + (i*comp_mbb), G2);
474     }
475 }
476 
477 void vector_subtraction(unum_t *res, unum_t *a, unum_t *b, unsigned comp_mbb
     , unsigned size) {
478 
479     for (unsigned i = 0; i < size; ++i) {
480         LDU(G0, a+(i*comp_mbb));
481         LDU(G1, b+(i*comp_mbb));
482         GSUB(G2, G0, G1);
483         STUL(res + (i*comp_mbb), G2);
484     }
485 }
486 
```

```
487 void copy_matrix_vector(unum_t *new_matrix, unum_t *matrix, unsigned
        comp_mbb, unsigned dim1, unsigned dim2 = 1) {
488     for (unsigned i = 0; i < dim1; ++i)
489         for (unsigned j = 0; j < dim2; ++j) {
490 //              new_matrix[i*dim2 + j] = Tdst(matrix[i*dim2 + j]);
491             LDU(G0, matrix+((i*dim2 + j)*comp_mbb));
492             STUL(new_matrix+((i*dim2 + j)*comp_mbb), G0);
493         }
494 }
495
496 template<typename Tdst, typename Tsrc>
497 void copy_matrix_vector(Tdst *new_matrix, Tsrc *matrix, unsigned dim1,
        unsigned dim2 = 1) {
498     for (unsigned i = 0; i < dim1; ++i)
499         for (unsigned j = 0; j < dim2; ++j)
500             new_matrix[i*dim2 + j] = Tdst(matrix[i*dim2 + j]);
501 }
502
503 double norm_res_unum_double(unum_t *res, int n, uint64_t mbb) //sqrt( sum((
        abs(res[i]))^2) )
504 {
505     //internal variables
506     unsigned i;
507     //accumulator
508     double sum=0;
509     //loaded values
510     typedef union
511     {
512         double   f_;
513         uint64_t i_;
514     } DoubleBits;
515     DoubleBits int_var;
516
517     for (i=0; i<n; i++){
518         LDU(G0, res+(i*mbb) );
519         GFLOAT_GL2D(int_var.i_ , G0);
520         sum += (int_var.f_ * int_var.f_ );
521         //printf("\t\tsum(%u)=%le | sqr=%le | elem=%le\n", i, sum, (int_var.
        f_ * int_var.f_), int_var.f_ );
522     }
523     return(sq_root_double(sum));
524 }
525
526 void copy_matrix_vector_double(double *new_matrix, double *matrix, unsigned
        dim1, unsigned dim2 = 1) {
527
528     for (unsigned i = 0; i < dim1; ++i)
529         for (unsigned j = 0; j < dim2; ++j) {
530             new_matrix[i*dim2 + j] = matrix[i*dim2 + j];
531         }
532 }
533
534 void copy_matrix_vector_double(double *new_matrix, unum_t *matrix, unsigned
        comp_mbb, unsigned dim1, unsigned dim2 = 1) {
535     typedef union
536     {
537         double   f_;
538         uint64_t i_;
539     } DoubleBits;
540     DoubleBits x_val;
541
542     for (unsigned i = 0; i < dim1; ++i)
543         for (unsigned j = 0; j < dim2; ++j) {
```

```
544             //new_matrix[i*dim2 + j] = Tdst(matrix[i*dim2 + j]);
545             LDU(G0, matrix+((i*dim2 + j)*comp_mbb));
546             GFLOAT_GL2D(x_val.i_, G0);
547             new_matrix[i*dim2 + j] = x_val.f_;
548         }
549 }
550
551 /* Choice of the elements of the array B */
552 void generate_vector(double *b ,int n) {
553     for(unsigned i = 0 ; i < n ; i++) {
554         b[i]=RAN[NB_RAN-1-(i%NB_RAN)];
555     }
556 }
557
558 double sq_root_double(double x) {
559     double rt = 1, ort = 0;
560     if (x>=0 && x==x){ //X is positive and it is not NaN
561         if (x!=(double)(1.0 / 0.0)){
562             while(ort!=rt) {
563                 ort = rt;
564                 rt = ((x/rt) + rt) / 2;
565             }
566             return rt;
567         }else{
568             return x;
569         }
570     } else { //result is NaN
571         return (0.0/0.0); // NaN
572     }
573 }
574
575 void matrix_div_scalar(double *matrix_result, double* matrix, double
    denominator, unsigned size) {
576     for(unsigned i = 0; i < size; ++i) {
577         for (unsigned j = 0; j < size; ++j) {
578             matrix_result[i*size + j] = matrix[i*size + j]/denominator;
579         }
580     }
581 }
582
583 void transpose_matrix(double *matrixPrime, double *matrix, unsigned size) {
584     for (unsigned i = 0; i < size; ++i) {
585         for (unsigned j = 0; j < size; ++j) {
586             matrixPrime[i*size + j] = matrix[j*size + i];
587         }
588     }
589 }
590
591 void randomSDPMatrix(double *matrix, unsigned size) {
592     transpose_matrix((double*)matrixPrime, matrix, size);
593     matrix_multiplication((double*)matrix_mult, (double*)matrix, (double*)
    matrixPrime, size, size, size);
594     matrix_div_scalar((double*)matrix, (double*)matrix_mult, double(size),
    size);
595 }
596
597 void zeros(double *matrix, unsigned dim1, unsigned dim2 = 1) {
598     for (unsigned i = 0; i < dim1; ++i)
599         for (unsigned j = 0; j < dim2; ++j) {
600            matrix[i*dim2 + j] = 0.0;
601         }
602 }
603
```

```
604 void zeros(unum_t *matrix, unsigned comp_mbb, unsigned dim1, unsigned dim2 =
        1) {
605     typedef union {
606         double   f_;
607         uint64_t i_;
608     } DoubleBits;
609     DoubleBits x_val;
610     x_val.f_ = 0.0;
611     for (unsigned i = 0; i < dim1; ++i)
612         for (unsigned j = 0; j < dim2; ++j) {
613 //            matrix[i*dim2 + j] = 0.0;
614            GFLOAT_D2G(G0, x_val.i_);
615            STUL(matrix + ((i*dim2 + j)*comp_mbb), G0);
616         }
617 }
618
619 unsigned long read_cycles(void)
620 {
621     unsigned long cycles;
622     asm volatile ("rdcycle %0" : "=r" (cycles));
623     return cycles;
624 }
625
626 void fill_matrix_market_NOS1(double matrix[237][237], unsigned size){
627     unsigned i,j;
628     if(size==237){
629         for(unsigned i=0; i<237; i++){
630             for(unsigned j=0; j<237; j++){
631                 matrix[i][j]=0.0;
632             }
633         }
634
635         // Fill the matrix with the not-null elements
636         matrix[1-1][1-1] =  1.6000000000000e+05;
637         matrix[1-1][1-1] =  1.6000000000000e+05;
638         ...
639
640     }else{
641         printf("\n\n\n\t\t!!!!!!!!!!!!!!!!!!!!!!!!!!!!!ERROR: matrix input
        size is wrong!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n\n\n");
642     }
643     return;
644 }
645
646 #ifdef __cplusplus
647 }
648 #endif
```

./Chapters/Chapter6_cg.c

Appendix C

Exploring other variable-precision formats

The UNUM [2] and posits [3] are the two main Variable-Precision (VP) Floating-Point (FP) formats available in state of the art. This thesis shows that the UNUM format is not perfect, and it proposes a modified version of the UNUM format to compensate for the deficiencies encountered during experiments (Section 7.2.3). Compared to UNUM, the posit format provides a better exponent encoding for small exponent values, but its footprint increases linearly among the exponent values (Section 3.1.4).

This chapter shows that the VP FP formats available in state of the art are not perfect, and it shows that other VP FP formats not yet explored still exists. Section C.1 describes a modified version of the UNUM format. Section C.2 proposes a modified version of the posit format. Section C.3 depicts a new family of VP FP formats. Finally, Section C.4 makes a comparison between all these new formats.

C.1 A modified UNUM format

This section describes a modified version of the UNUM format that we plan to support in the new release of the VP unit.

There are some cases where the modified UNUM format used in the VP unit (BMF) has less precision than double due to the presence of its descriptor fields (`es-1`, `fs-1`, and `u`). These fields use some bits to encode the mantissa while gaining computation precision. Even if data in memory are stored using this format, the experiments of Chapter 6 show that the results output precisions, compared to the IEEE 754 64-bit double format, do not differ. Thus, for at least this set of applications, the UNUM memory format does not degrade the output result accuracy, even if there are some cases where the representation of data in memory has less precision than double. As a consequence, a format that has more mantissa bits can get only benefits.

The BMF format uses MBB-byte memory slots for its encoding (Section 4.3). This "limitation" on the original UNUM format [2] makes the `fs-1` UNUM field useless. The bits used to encode this field could be used to increase the mantissa precision. Furthermore, experiments show that there is no need to have interval arithmetic support directly in the format: we used interval arithmetic only to evaluate the computational error at the end of each application kernel. Thus, the u-bit (`u`) field useless as well.

We propose to support the UNUM-like VP FP format depicted in Figure C.1. In this format, the size is programmable with a byte-granularity. The exponent encoding supported is the same as conventional UNUMs, and the `f` field takes all the remaining bits in the slot of MBB bytes. Unlike the conventional UNUM format, and the BMF format described in Section 4.3, this format does not have the `fs-1` and `u` fields since it has a fixed memory footprint the declaration of the variable, and does not support interval arithmetic.

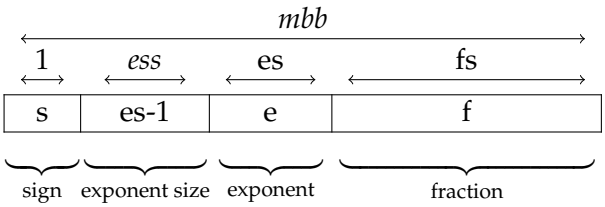


Figure C.1: The modified UNUM floating-point format

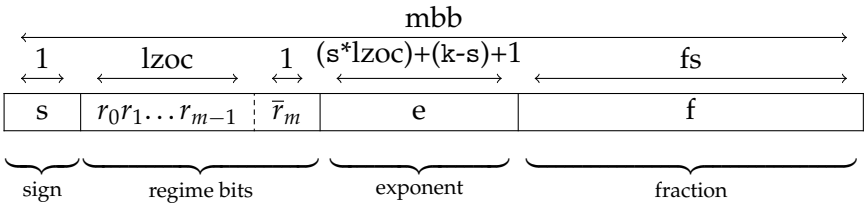


Figure C.2: The modified posit floating-point format

The exponent encoding is the same as the original UNUM format (Section 3.1). Subnormal numbers are supported. Distinguished values such as zero, infinity, and not-a-number have distinguished encodings. Zero is encoded with all the fields set to 0. Negative zero is not allowed. The encoding for infinity uses the `es-1` and `e` fields with their bits all set to 1, and the `f` fields with the less significant bit unset and the others set to 1. In the case that the `f` field has only one bit, it is set to 0. The `s` field differentiates the encodings for positive and negative infinity. Not-a-number is encoded with the `es-1`, the `e`, and the `f` fields with all their bits set to 1. The `s` field differentiates two not-a-number encodings (i.e., signaling and quiet not-a-number). However, we do not have any application for which these two encodings can provide any benefit.

The user can support interval arithmetic operations with this scalar format through microcode. Since this format does not have the u-bit, we cannot support the concepts of almost-infinity or almost-zero (details in Section 3.1.3). However, we did not find any applications for which a dedicated encoding for these concepts can make any difference in the output result. The operations that receive in input an almost-infinity produce the same output of the operations that get infinity in the input. Furthermore, operations that receive an almost-zero in input suffer from cancellation as well as operations that get zero in the input.

C.2 A modified posit format

The main difference of the posit FP format [3] comparing to the UNUM one is its fixed size and exponent encoding. As explained in Section 3.1, comparing to UNUM, the posit exponent encoding can have a more compact exponent around zero at the cost of having an exponent footprint that grows linearly (not logarithmically like in UNUM) with the exponent value. Figure C.2 shows the modified posit FP format with an exponent memory footprint that is more compact around zero, like posit, and that its footprint grows logarithmically with the exponent value, like UNUM.

This format is similar to the posit one (Figure 1.3, Section 3.1.4). It differs on its length and a slightly different exponent encoding. It has a length defined with a byte granularity. Its exponent encoding uses the regime bits, like in the posit format [3], to encode the exponent field length. Please note that, in the posit format, the exponent field has fixed length, and the regime bits encode the exponent offset from the value zero.

In this format, the `k` and the `s` parameters specify the minimal exponent field length and the exponent increment gap, respectively. Their minimal value is 1. Like the `es` parameter in

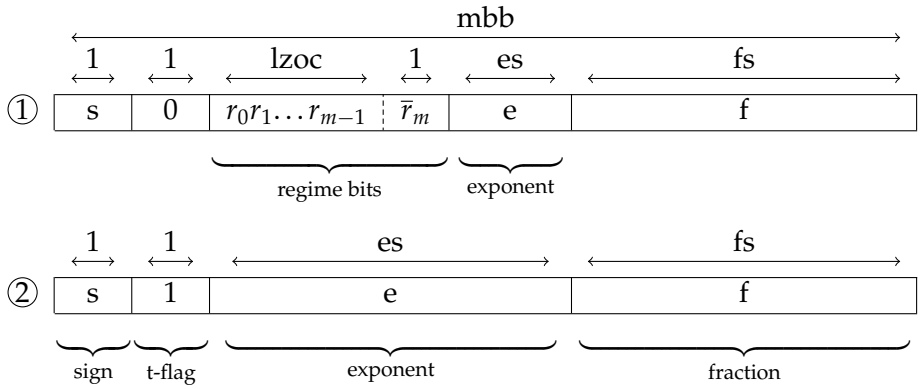


Figure C.3: The not-continuous posit floating-point format

posit, the higher the k and s values, the lower the bit-length gain around zero and the bit-length overhead at the maximum exponent value. The equation below shows the equations for the exponent encoding that we propose for k=1 and s=1.

$$exp = \begin{cases} -(\sum_{i=2}^{i=lzoc} 2^i) - \bar{e} - 1, & \text{if } r_0 = 0 \text{ (negative values) and } lzoc > 1 \\ -\bar{e} - 1, & \text{if } r_0 = 0 \text{ (negative values) and } lzoc = 1 \\ +e, & \text{if } r_0 = 1 \text{ (positive values) and } lzoc = 1 \\ +(\sum_{i=2}^{i=lzoc} 2^i) + e, & \text{if } r_0 = 1 \text{ (positive values) and } lzoc > 1 \end{cases}$$

Table C.1 shows the comparison between different posit and modified posit exponent encodings. This comparison is made for exponent values between -32 and +31 ⓐ; thus, the baseline encoding is a 6-bit 2's complement encoding ⓑ. Columns ⓒ and ⓓ represent the exponent encodings of the posit format with the configuration parameters es=2 and es=3, respectively. Columns ⓔ, ⓕ, ⓖ, and ⓗ represent the exponent encodings of the modified posit exponent encoding with a different configuration of the k and s parameters. The last bit of the regime bit field blue to help the reader to distinguish the exponent subfields.

As is possible to see, the exponent encoding of the modified-posit format guarantees exponent representations nearby the exponent value zero with the same bit-length as posit, while having a logarithmic bit-length increase with the exponent value, like UNUM. The k and s parameters change the way how the exponent values are encoded. The higher these values are, the higher is the exponent bit-length nearby the exponent value zero, the lower is the exponent bit-length for exponent values high in magnitude.

In some cases, posit has a more compact exponent encoding than the modified posit one. The exponent encoding can be more compact because, unlike posit, the modified posit exponent encodings grows with a granularity of two bits. On the contrary, the modified posit encoding has a larger dynamic than the one possible to encode in the posit format.

C.3 A new family of not-continuous variable-precision formats

State of the art for Variable-Precision (VP) Floating-Point (FP) formats focused on finding new exponent encodings to gain mantissa precision for exponent values around zero. All these exponent encodings are defined on *continuous* functions that map every exponent value with its binary encoding [2], [3]. They compact exponent encodings for values around zero, at the cost of having larger exponent encodings for values far from zero (Section 3.1). This section proposes a new category of VP FP formats with a *not-continuous* exponent encoding to reduce the average exponent footprint in the format.

Integer ⓐ	2's complement ⓑ	posit es=2 ⓒ	posit es=3 ⓓ	mod. posit k=1, s=1 ⓔ	mod. posit k=2, s=1 ⓕ	mod. posit k=1, s=2 ⓖ	mod. posit k=2, s=2 ⓗ
-32	100000	00000000100	00001000	0000111100	000111000	0001110100	00101000
-31	100001	00000000101	00001001	0000111101	000111001	0001110101	00101001
-30	100010	00000000110	00001010	0000111110	000111010	0001110110	00101010
-29	100011	00000000111	00001011	0000111111	000111011	0001110111	00101011
-28	100100	0000000100	00001100	00010000	000111100	0001111000	00101100
-27	100101	0000000101	00001101	00010001	000111101	0001111001	00101101
-26	100110	0000000110	00001110	00010010	000111110	0001111010	00101110
-25	100111	0000000111	00001111	00010011	000111111	0001111011	00101111
-24	101000	000000100	0001000	00010100	0010000	0001111100	00110000
-23	101001	000000101	0001001	00010101	0010001	0001111101	00110001
-22	101010	000000110	0001010	00010110	0010010	0001111110	00110010
-21	101011	000000111	0001011	00010111	0010011	0001111111	00110011
-20	101100	00000100	0001100	00011000	0010100	0010000	00110100
-19	101101	00000101	0001101	00011001	0010101	0010001	00110101
-18	101110	00000110	0001110	00011010	0010110	0010010	00110110
-17	101111	00000111	0001111	00011011	0010111	0010011	00110111
-16	110000	0000100	001000	00011100	0011000	0010100	00111000
-15	110001	0000101	001001	00011101	0011001	0010101	00111001
-14	110010	0000110	001010	00011110	0011010	0010110	00111010
-13	110011	0000111	001011	00011111	0011011	0010111	00111011
-12	110100	000100	001100	001000	0011100	0011000	00111100
-11	110101	000101	001101	001001	0011101	0011001	00111101
-10	110110	000110	001110	001010	0011110	0011010	00111110
-9	110111	000111	001111	001011	0011111	0011011	00111111
-8	111000	00100	01000	001100	01000	0011100	01000
-7	111001	00101	01001	001101	01001	0011101	01001
-6	111010	00110	01010	001110	01010	0011110	01010
-5	111011	00111	01011	001111	01011	0011111	01011
-4	111100	0100	01100	0100	01100	0100	01100
-3	111101	0101	01101	0101	01101	0101	01101
-2	111110	0110	01110	0110	01110	0110	01110
-1	111111	0111	01111	0111	01111	0111	01111
0	000000	1000	10000	1000	10000	1000	10000
1	000001	1001	10001	1001	10001	1001	10001
2	000010	1010	10010	1010	10010	1010	10010
3	000011	1011	10011	1011	10011	1011	10011
4	000100	11000	10100	110000	10100	1100000	10100
5	000101	11001	10101	110001	10101	1100001	10101
6	000110	11010	10110	110010	10110	1100010	10110
7	000111	11011	10111	110011	10111	1100011	10111
8	001000	111000	110000	110100	1100000	1100100	11000000
9	001001	111001	110001	110101	1100001	1100101	11000001
10	001010	111010	110010	110110	1100010	1100110	11000010
11	001011	111011	110011	110111	1100011	1100111	11000011
12	001100	1111000	110100	11100000	1100100	1101000	11000100
13	001101	1111001	110101	11100001	1100101	1101001	11000101
14	001110	1111010	110110	11100010	1100110	1101010	11000110
15	001111	1111011	110111	11100011	1100111	1101011	11000111
16	010000	11111000	1110000	11100100	1101000	1101100	11001000
17	010001	11111001	1110001	11100101	1101001	1101101	11001001
18	010010	11111010	1110010	11100110	1101010	1101110	11001010
19	010011	11111011	1110011	11100111	1101011	1101111	11001011
20	010100	111111000	1110100	11101000	1101100	1110000000	11001100
21	010101	111111001	1110101	11101001	1101101	1110000001	11001101
22	010110	111111010	1110110	11101010	1101110	1110000010	11001110
23	010111	111111011	1110111	11101011	1101111	1110000011	11001111
24	011000	1111111000	11110000	11101100	111000000	1110000100	11010000
25	011001	1111111001	11110001	11101101	111000001	1110000101	11010001
26	011010	1111111010	11110010	11101110	111000010	1110000110	11010010
27	011011	1111111011	11110011	11101111	111000011	1110000111	11010011
28	011100	11111111000	11110100	1111000000	111000100	1110001000	11010100
29	011101	11111111001	11110101	1111000001	111000101	1110001001	11010101
30	011110	11111111010	11110110	1111000010	111000110	1110001010	11010110
31	011111	11111111011	11110111	1111000011	111000111	1110001011	11010111

Table C.1: Comparison between the exponent encodings of the posit and the modified posit formats, varying their configurations parameters

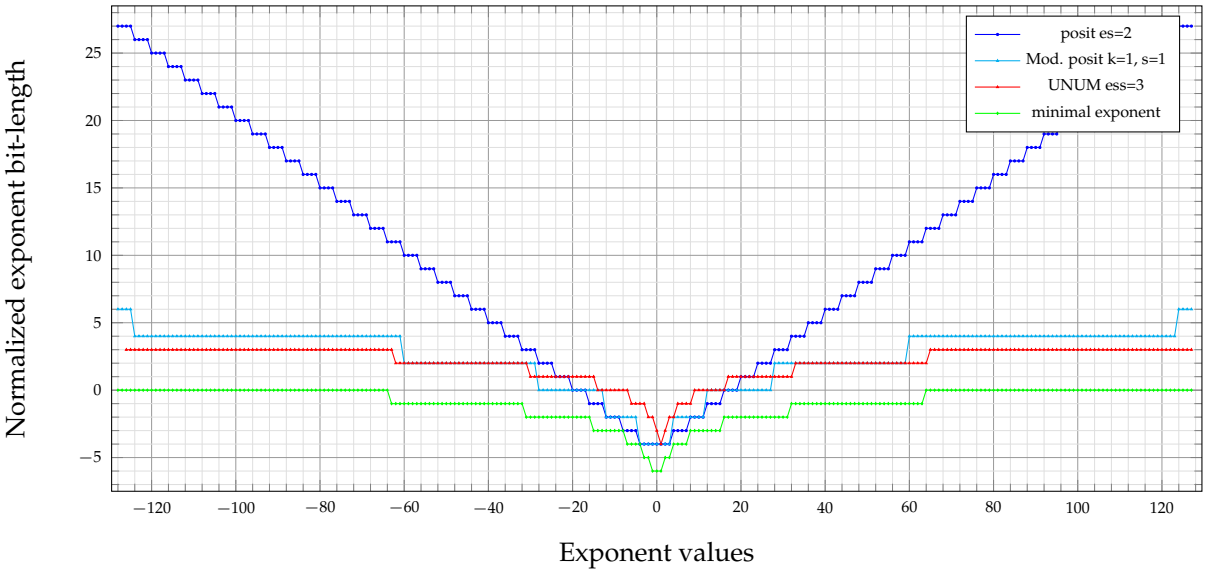


Figure C.4: Exponent bit-length comparison between the posit, the UNUM, and the modified posit formats, normalized with respect to a two's complement 8-bit exponent encoding

The idea is straightforward: use VP exponent encodings for values nearby zero and use a standard encoding, for instance, two's complement or biased encodings, for values far from zero. This configuration is worth it when the maximum exponent bit-length does not overpass the two's complement encoding. Figure C.3 shows an example of a VP FP format that follows this idea. This format takes advantage of the posit exponent encoding for exponent values near zero ①, and the two's complement encoding for values far from zero ② (for large magnitude exponent values). The *threshold* flag (t-flag) marks the transition between the two exponent encodings.

The exponent encoding of this format is 1 bit larger than the posit format around zero. For FP values nearby zero, the mantissa is 1-bit less precise than the same mantissa in the posit format. Nevertheless, it bounds the worst-case exponent field bit-length to 1 bit as soon as the t-flag is activated. This format may be useful in applications with a large exponent variability. However, its benefits have to be measured in real applications with real hardware.

C.4 Formats comparison

Figure C.4 shows a comparison between the exponent encodings used for UNUM format, posit format, and the exponent encoding that we propose in the modified posit format (Section C.2). The horizontal axis shows all the possible exponent values which span from -128 and +127. The vertical axis represents the bit-overhead of the exponent encoding according to the 8-bit encoding baseline. This comparison is normalized to an 8-bit two's complement exponent encoding, representing the exponent encoding for a custom IEEE-like format. All the points of the y-axis below zero are exponent encodings more compact than an 8-bit two's complement representation, while all the above are those requiring more bits.

The green line depicts the minimal exponent encoding, in two's complement, for each possible exponent value. This line represents the minimum theoretical footprint that an exponent encoding can have (-1, 0, and 1, can be encoded on 2 bits). The other three lines show the exponent overhead for the UNUM, posit, and the modified posit encodings, with ess=3, es=2 and k=s=1, respectively. The usage of additional bits to encode the length (or the end) of the exponent field in these encodings hinders the possibility of reaching the green line. This plot

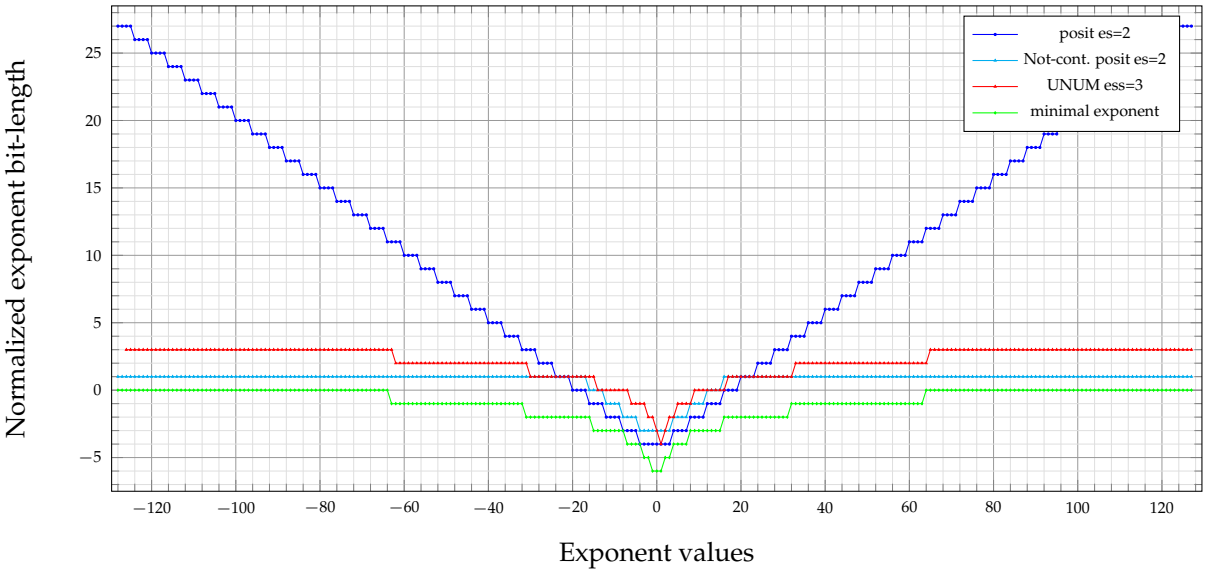


Figure C.5: Exponent bit-length comparison between the posit, the UNUM, and the not-continuous posit-like formats, normalized with respect to a two's complement 8-bit exponent encoding

depicts the three exponent encoding formats, with the configurations' ess=3, es=2, and k=s=1, because they share the same minimum encoding footprint.

Looking at the "window of values" where an advantage on bit footprint for the exponent encodings exists, the posit format is better than the UNUM one. On the contrary, for exponent values outside this window (especially for small es values in posit), the posit format has a dramatically worst encoding than the one used in the UNUM format. Having this worst encoding can be dangerous in applications that use (also few) large values to do computation. They can propagate significant cancellation errors in the computation.

The exponent encoding we propose is a tradeoff between the two exponent encodings. For exponent values nearby zero, it behaves like the posit format being more compact than the UNUM exponent encoding. For exponent values far from zero, it behaves like the UNUM format being more compact than posit.

Figure C.5 depicts a similar comparison but with the not-continuous posit-like format described in Section C.3. On the one hand, as it is possible to see, the posit format is one bit more precise than the not-continuous format for exponent values nearby zero. This offset is due to the threshold flag. On the other hand, this encoding is the most compact available in state of the art for exponent values large in magnitude.

Summary This appendix shows two examples of Variable-Precision (VP) floating-point format different from state of the art. No format is better than another one in general. Every format works better in specific exponent values ranges. This keeps the research on VP architectures, algorithms, and formats is still open for future investigations.

Bibliography

[1] "IEEE standard for floating-point arithmetic", *IEEE Std 754-2008*, Aug. 2008. DOI: 10.1109/IEEESTD.2008.4610935.

[2] J. L. Gustafson, *The end of error: Unum computing*, C. &.H. C. Science, Ed. Feb. 2015.

[3] J. L. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic", *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, 2017, ISSN: 2313-8734. DOI: 10.14529/jsfi170206.

[4] U. W. Kulisch, *Computer arithmetic and validity: Theory, implementation, and applications*. Berlin, Boston: De Gruyter, Oct. 2018.

[5] V. Eijkhout, *Introduction to high performance scientific computing*. Lulu.com, Oct. 2012, ISBN: 1257992546. DOI: 10.5555/2464807.

[6] D. Zivanovic, M. Pavlovic, M. Radulovic, H. Shin, J. Son, S. A. Mckee, P. M. Carpenter, P. Radojković, and E. Ayguadé, "Main memory in HPC: Do we need more or could we live with less?", *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 1, 3:1–3:26, Mar. 2017, ISSN: 1544-3566. DOI: 10.1145/3023362.

[7] "The rocket chip generator", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[8] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of floating-point arithmetic, 2nd edition*. Birkhäuser Boston, May 2018, p. 632, ISBN: 978-3-319-76525-9.

[9] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding", *ACM Transactions on Mathematical Software*, vol. 33, no. 2, Jun. 2007, ISSN: 0098-3500. DOI: 10.1145/1236463.1236468.

[10] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn, "The transprecision computing paradigm: Concept, design, and applications", in *Design, Automation Test in Europe Conference Exhibition*, Mar. 2018, pp. 1105–1110. DOI: 10.23919/DATE.2018.8342176.

[11] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, "Design, implementation and testing of extended and mixed precision BLAS", *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 152–205, Jun. 2002, ISSN: 0098-3500. DOI: 10.1145/567806.567808.

[12] M. Baboulin, A. Buttari, J. J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms", *Computer Physics Communications*, vol. 180, pp. 2526–2533, Dec. 2009. DOI: 10.1016/j.cpc.2008.11.005.

[13] M. Joldes, O. Marty, J. Muller, and V. Popescu, "Arithmetic algorithms for extended precision using floating-point expansions", *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1197–1210, Apr. 2016, ISSN: 0018-9340. DOI: 10.1109/TC.2015.2441714.

[14] L. Kettner and S. Näher, "Two computational geometry libraries: LEDA and CGAL", in *Handbook of Discrete and Computational Geometry, Second Edition*. Apr. 2004, pp. 1435–1463. DOI: 10.1201/9781420035315.

[15] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision arithmetic library*, version 5.0.5, 2012. [Online]. Available: https://gmplib.org/.

[16] F. Johansson, J. Davenport, M. Kauers, G. Labahn, and J. Urban, "Numerical integration in arbitrary-precision ball arithmetic", in *Lecture Notes in Computer Science*, vol. 10931, Springer International Publishing, Jul. 2018, pp. 255–263. DOI: 10.1007/978-3-319-96418-8_30.

[17] M. D. Ercegovac, "On-line arithmetic: An overview", in *28th Annual Technical Symposium on Real-Time Signal Processing VII*, vol. 0495, SPIE, Nov. 1984. DOI: 10.1117/12.944012.

[18] F. Barsi and P. Maestrini, "Error correcting properties of redundant residue number systems", *IEEE Transactions on Computers*, vol. C-22, no. 3, pp. 307–315, Mar. 1973, ISSN: 0018-9340. DOI: 10.1109/T-C.1973.223711.

[19] M. G. Arnold, T. A. Bailey, J. R. Cowles, and M. D. Winkel, "Applying features of IEEE 754 to sign/logarithm arithmetic", *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 1040–1050, Aug. 1992, ISSN: 2326-3814. DOI: 10.1109/12.156547.

[20] ——, "Arithmetic co-transformations in the real and complex logarithmic number systems", *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 777–786, Jul. 1998, ISSN: 2326-3814. DOI: 10.1109/12.709377.

[21] J. Johnson, "Rethinking floating point for deep learning", Nov. 2018. arXiv: 1811.01721 [cs.NA].

[22] C. B. Moler, "Iterative refinement in floating point", *J. ACM*, vol. 14, no. 2, pp. 316–321, Apr. 1967, ISSN: 0004-5411.

[23] N. J. Higham, *Accuracy and stability of numerical algorithms*, Second. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, Aug. 2002, ISBN: 978-0-89871-521-7. DOI: 10.1137/1.9780898718027.

[24] N. Revol, "The MPFI library: Towards IEEE 1788–2015 compliance", in *Parallel Processing and Applied Mathematics*, Springer International Publishing, Mar. 2020, pp. 353–363. DOI: 10.1007/978-3-030-43222-5_31.

[25] "IEEE standard for interval arithmetic", *IEEE Std 1788-2015*, pp. 1–97, Jun. 2015, ISSN: null. DOI: 10.1109/IEEESTD.2015.7140721.

[26] E. Carson and N. J. Higham, "Accelerating the solution of linear systems by iterative refinement in three precisions", *SIAM Journal on Scientific Computing*, vol. 40, no. 2, Jan. 2018. DOI: 10.1137/17M1140819.

[27] F. Morrison, "High precision arithmetic for scientific applications", *CoRR*, Sep. 2013. arXiv: 1309.5498.

[28] D. H. Bailey, "High-precision floating-point arithmetic in scientific computation", *Computing in Science Engineering*, vol. 7, no. 3, May 2005. DOI: 10.1109/MCSE.2005.52.

[29] J. Thijssen, *Computational physics*, 2nd ed. Jun. 2007. DOI: 10.1017/CBO9781139171397.

[30] P. Grandclément, "Introduction to spectral methods", in *Stellar Fluid Dynamics and Numerical Simulations: From the Sun to Neutron Stars*, EAS Publications Series, Sep. 2006, pp. 153–180. DOI: 10.1051/eas:2006112.

[31] C. Grossmann, H.-G. Roos, and M. Stynes, *Numerical treatment of partial differential equations*. Springer International Publishing, 2007, ISBN: 978-3-540-71582-5. DOI: 10.1007/978-3-540-71584-9.

[32] D. Gottlieb and S. A. Orszag, "Numerical analysis of spectral methods: Theory and applications", vol. 26, Jan. 1977. DOI: 10.1137/1.9781611970425.

[33] E. Hewitt and R. E. Hewitt, "The Gibbs-Wilbraham phenomenon: An episode in Fourier analysis", *Archive for History of Exact Sciences*, vol. 21, no. 2, pp. 129–160, Jun. 1979. DOI: 10.1007/BF00330404.

[34] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in c (2nd ed.): The art of scientific computing*. New York, NY, USA: Cambridge University Press, Oct. 1992, ISBN: 0-521-43108-5.

[35] S. Boldo, F. Faissole, and A. Chapoutot, "Round-off error analysis of explicit one-step numerical integration methods", in *24th IEEE Symposium on Computer Arithmetic*, London, United Kingdom, Jul. 2017. DOI: 10.1109/ARITH.2017.22.

[36] Y. Saad, J. R. Chelikowsky, and S. M. Shontz, "Numerical methods for electronic structure calculations of materials", *SIAM Review*, vol. 52, no. 1, pp. 3–54, Feb. 2010, ISSN: 0036-1445. DOI: 10.1137/060651653.

[37] R. Walker, *Electronic structure calculations on graphics processing units: From quantum chemistry to condensed matter physics*. Feb. 2016. DOI: 10.1002/9781118670712.

[38] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, "Applying the roofline model", in *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2014, pp. 76–85. DOI: 10.1109/ISPASS.2014.6844463.

[39] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen, "LAPACK: A portable linear algebra library for high-performance computers", in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Los Alamitos, CA, USA, Nov. 1990, pp. 2–11, ISBN: 0-89791-412-0. DOI: 10.1109/SUPERC.1990.129995.

[40] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "Mumps: A general purpose distributed memory sparse solver", in *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*, T. Sørevik, F. Manne, A. H. Gebremedhin, and R. Moe, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 121–130, ISBN: 978-3-540-70734-9. DOI: 10.1007/3-540-70734-4_16.

[41] J. Wilkinson, *Rounding errors in algebraic processes*. Englewood Cliffs, 1964, ISBN: 0-486-67999-3.

[42] D. H. Bailey, R. Barrio, and J. Borwein, "High-precision computation: Mathematical physics and dynamics", *Applied Mathematics and Computation*, vol. 218, no. 20, pp. 10106 –10 121, Mar. 2012, ISSN: 0096-3003. DOI: 10.1016/j.amc.2012.03.087.

[43] D. H. Bailey and J. M. Borwein, "High-precision numerical integration: Progress and challenges", *J. Symb. Comput.*, vol. 46, no. 7, pp. 741–754, Jul. 2011, ISSN: 0747-7171. DOI: 10.1016/j.jsc.2010.08.010.

[44] F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel, *The SIAM 100-digit challenge, a study in high-accuracy numerical computing*, 2004. DOI: 10.1137/1.9780898717969.

[45] A. Mushtaq, A. Noreen, K. Olaussen, and I. Overbo, "Very-high-precision solutions of a class of Schroedinger type equations", *Computer Physics Communications*, vol. 182, no. 9, pp. 1810 –1813, Sep. 2011, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2010.12.046.

[46] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage", *ACM Transactions on Mathematical Softhware*, vol. 5, no. 3, pp. 308–323, Sep. 1979, ISSN: 0098-3500. DOI: 10.1145/355841.355847.

[47] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan, "DLFloat: A 16-b floating point format designed for deep learning training and inference", in *IEEE 26th Symposium on Computer Arithmetic*, Jun. 2019, pp. 92–95. DOI: 10.1109/ARITH.2019.00023.

[48] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks", in *IEEE 26th Symposium on Computer Arithmetic*, Jun. 2019, pp. 88–91. DOI: 10.1109/ARITH.2019.00022.

[49] M. J. Schulte and E. E. Swartzlander, "A family of variable-precision interval arithmetic processors", *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 387–397, May 2000, ISSN: 0018-9340. DOI: 10.1109/12.859535.

[50] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the hardware cost of the posit number system", in *FPL 2019 - 29th International Conference on Field-Programmable Logic and Applications (FPL)*, Barcelona, Spain, Sep. 2019, pp. 1–8. DOI: 10.1109/FPL.2019.00026.

[51] R. Kirchner and U. W. Kulisch, "Accurate arithmetic for vector processors", *Journal of Parallel and Distributed Computing*, vol. 5, no. 3, pp. 250–270, Jun. 1988, ISSN: 0743-7315. DOI: 10.1016/0743-7315(88)90020-2.

[52] R. Kirchner and U. W. Kulisch, "Accurate arithmetic for vector processors", *Journal of Parallel and Distributed Computing*, vol. 5, no. 3, pp. 250–270, Jun. 1988, ISSN: 0743-7315. DOI: 10.1016/0743-7315(88)90020-2.

[53] A. Knöfel, "Fast hardware units for the computation of accurate dot products", in *Proceedings 10th IEEE Symposium on Computer Arithmetic*, Jun. 1991, pp. 70–74. DOI: 10.1109/ARITH.1991.145536.

[54] M. Muller, C. Rub, and W. Rulling, "Exact accumulation of floating-point numbers", in *Proceedings 10th IEEE Symposium on Computer Arithmetic*, Jun. 1991, pp. 64–69, ISBN: 0-8186-9151-4. DOI: 10.1109/ARITH.1991.145535.

[55] P. R. Capello and W. L. Miranker, "Systolic super summation", *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 657–677, Jun. 1988, ISSN: 2326-3814. DOI: 10.1109/12.2205.

[56] J. Kernhof, C. Baumhof, B. Hofflinger, U. W. Kulisch, S. Kwee, P. Schramm, M. Selzer, and T. Teufel, "A CMOS floating-point processing chip for verified exact vector arithmetic", in *Twientieth European Solid-State Circuits Conference*, Sep. 1994, pp. 196–199, ISBN: 2-86332-160-9.

[57] U. W. Kulisch, "The fifth floating-point operation for top-performance computers accumulation of floating-point numbers and products in fixed-point arithmetic", in *Forschungsschwerpunkts Computerarithmetik, Intervall-rechnung und Numerische Algorithmen mit Ergebnisverifikation, Universität Karls-ruhe*, Apr. 1997. DOI: 10.5445/IR/70197.

[58] D. R. Lutz and C. N. Hinds, "High-precision anchored accumulators for reproducible floating-point summation", in *IEEE 24th Symposium on Computer Arithmetic*, Jul. 2017, pp. 98–105. DOI: 10.1109/ARITH.2017.20.

[59] N. Brunie, "Modified fused multiply and add for exact low precision product accumulation", in *IEEE 24th Symposium on Computer Arithmetic*, Jul. 2017, pp. 106–113. DOI: 10.1109/ARITH.2017.29.

[60] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanovic, "A hardware accelerator for computing an exact dot product", in *IEEE 24th Symposium on Computer Arithmetic*, Jul. 2017, pp. 114–121. DOI: 10.1109/ARITH.2017.38.

[61] M. S. Cohen, T. E. Hull, and V. C. Hamacher, "CADAC: A controlled-precision decimal arithmetic unit", *IEEE Transactions on Computers*, vol. C-32, no. 4, pp. 370–377, Apr. 1983, ISSN: 0018-9340. DOI: 10.1109/TC.1983.1676238.

[62] T. E. Hull, M. S. Cohen, and C. B. Hall, "Specifications for a variable-precision arithmetic coprocessor", in *Proceedings 10th IEEE Symposium on Computer Arithmetic*, Jun. 1991, pp. 127–131. DOI: 10.1109/ARITH.1991.145548.

[63] N. Anane, H. Bessalah, M. Issad, K. Messaoudi, and M. Anane, "Hardware implementation of variable precision multiplication on FPGA", in *Design Technology of Integrated Systems in Nanoscal Era*, Apr. 2009, pp. 77–81. DOI: 10.1109/DTIS.2009.4938028.

[64] Y. Lei, Y. Dou, J. Zhou, and S. Wang, "VPFPAP: A special-purpose VLIW processor for variable-precision floating-point arithmetic", in *Field Programmable Logic and Applications*, Sep. 2011, pp. 252–257. DOI: 10.1109/FPL.2011.51.

[65] Y. Lei, Y. Dou, S. Guo, and J. Zhou, "FPGA implementation of variable-precision floating-point arithmetic", in *Advanced Parallel Processing Technologies*, Springer Berlin Heidelberg, Sep. 2011, pp. 127–141, ISBN: 978-3-642-24151-2.

[66] M. M. Ozbilen and M. Gok, "A multi-precision floating-point adder", in *Ph.D. Research in Microelectronics and Electronics*, Jun. 2008, pp. 117–120. DOI: 10.1109/RME.2008.4595739.

[67] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar, "A 1.45GHz 52-to-162GFLOPS/W variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm CMOS", in *IEEE International Solid-State Circuits Conference*, Feb. 2012, pp. 182–184. DOI: 10.1109/ISSCC.2012.6176987.

[68] S. Arish and R. K. Sharma, "Run-time reconfigurable multi-precision floating point multiplier design for high speed, low-power applications", in *Signal Processing and Integrated Networks*, Feb. 2015, pp. 902–907. DOI: 10.1109/SPIN.2015.7095315.

[69] M. Nakata, "Poster: MPACK 0.7.0: Multiple precision version of BLAS and LAPACK", in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov. 2012, pp. 1353–1353. DOI: 10.1109/SC.Companion.2012.183.

[70] Z. Xianyi, W. Qian, and Z. Chothia, "OpenBLAS", 2014. [Online]. Available: http://xianyi.github.io/OpenBLAS.

[71] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic", in *Proceedings 10th IEEE Symposium on Computer Arithmetic*, Jun. 1991, pp. 132–143. DOI: 10.1109/ARITH.1991.145549.

[72] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates", *Discrete & Computational Geometry*, vol. 18, no. 3, pp. 305–363, Oct. 1997, ISSN: 1432-0444. DOI: 10.1007/PL00009321.

[73] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Jul. 2019. DOI: 10.1109/TVLSI.2019.2926114.

[74] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: The good, the bad and the ugly", in *CoNGA'19*, ACM, Mar. 2019, 6:1–6:10, ISBN: 978-1-4503-7139-1. DOI: 10.1145/3316279.3316285.

[75] A. Rao, "The RoCC doc v2: An introduction to the rocket custom coprocessor interface", Tech. Rep.

[76] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, Jan. 2001, ISBN: 978-1-55860-671-5. DOI: 10.5555/355074.

[77] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture", in *Proceedings of the 6th International Symposium on Memory Management*, ser. ISMM '07, New York, NY, USA: ACM, Oct. 2007, pp. 103–104, ISBN: 978-1-59593-893-0. DOI: 10.1145/1296907.1296909.

[78] A. Munshi, "The OpenCL specification", in *IEEE Hot Chips 21 Symposium*, IEEE, Aug. 2009, pp. 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342.

[79] M. R. Hestenes and E. L. Stiefel, "Methods of conjugate gradients for solving linear systems", vol. 49, no. 6, Dec. 1952. DOI: 10.6028/jres.049.044.

[80] P. Vivet, E. Guthmuller, Y. Thonnart, G. Pillonnet, G. Moritz, I. Miro-Panadès, C. Fuguet, J. Durupt, C. Bernard, D. Varreau, J. Pontes, S. Thuries, D. Coriat, M. Harrand, D. Dutoit, D. Lattard, L. Arnaud, J. Charbonnier, P. Coudrain, A. Garnier, F. Berger, A. Gueugnot, A. Greiner, Q. Meunier, A. Farcy, A. Arriordaz, S. Cheramy, and F. Clermidy, "A 220GOPS 96-core processor with 6 chiplets 3D-stacked on an active interposer offering 0.6ns/mm latency, 3Tb/s/mm2 inter-chiplet interconnects and 156mW/mm2 @ 82%-peak-efficiency DC-DC converters", in *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*, Feb. 2020. DOI: 10.1109/ISSCC19947.2020.9062927.

[81] *Matrix market repository*, Accessed: 2019-11-30, May 2007. [Online]. Available: https://math.nist.gov/MatrixMarket/.

[82] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, "Matrix market: A web resource for test matrix collections", in *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, London, UK, UK: Chapman & Hall, Ltd., Jan. 1997, pp. 125–137, ISBN: 978-1-5041-2942-8. DOI: 10.1007/978-1-5041-2940-4_9.

[83] A. Waterman and K. Asanović, "The RISC-V instruction set manual, volume i: User-level ISA", Tech. Rep., version 2.2, May 2017.