Description and compilation of ad-hoc arithmetic operators in the context of High-Level Synthesis

Luc Forget

N° d'ordre NNT : 2023ISAL0046

THÈSE DE DOCTORAT DE L'INSA LYON
Membre de l'Université de Lyon

École Doctorale 512
Informatique et Mathématiques

Spécialité / Discipline de doctorat :
Informatique

Soutenue publiquement le 29/06/2023, par :
Luc FORGET

Description and compilation of ad-hoc arithmetic operators in the context of High-Level Synthesis.

Devant le jury composé de :

RASTELLO Fabrice	Directeur de recherches	INRIA Grenoble	Président
DERRIEN Steven	Professeur des Universités	Université de Rennes	Rapporteur
FERRANDI Fabrizio	Professeur des Universités	Politecnico Milano	Rapporteur
CHOTIN Roselyne	Maître de Conférences HDR	LIP6	Examinatrice
VOLKOVA Anastasia	Maître de Conférences	Université de Nantes	Examinatrice
DE DINECHIN Florent	Professeur des Universités	INSA de Lyon	Directeur de thèse

Département FEDORA – INSA Lyon – Écoles Doctorales

Sigle	École doctorale	Nom et coordonnées du responsable

CHIMIE	Chimie de Lyon
	`https://www.edchimie-lyon.fr`
	Sec: Renée El Melhem
	Bât: Blaise Pascal, 3ème étage
	Tél: 04 72 43 80 46
	`secretariat@edchimie-lyon.fr`
		M. Stéphane DANIELE
		C2P2-CPE LYON-UMR 5265
		Bâtiment F308 bP 2077
		43 Boulevard du 11 novembre 1918
		69616 Villeurbanne
		Tél: 04 72 44 53 60
		`directeur@edchimie-lyon.fr`

E.E.A.	Électronique, Électrotechnique, Automatique
	`https://edeea.universite-lyon.fr`
	Sec: Stéphanie Cauvin
	Bât: Direction INSA Lyon
	Tél: 04 72 43 71 70
	`secretariat.edeea@insa-lyon.fr`
		M. Philippe Delachartre
		Laboratoire Creatis
		Bâtiment Blaise Pascal
		7 avenue Jean Capelle
		69621 Villeurbanne CEDEX
		Tél: 04 72 43 88 63
		`philippe.delachartre@insa-lyon.fr`

E2M2	Évolution, Écosystème, Microbiologie, Modélisation
	`http://e2m2.universite-lyon.fr`
	Sec: Bénédicte Lanza
	Bât: Atrium, UCB Lyon 1
	Tél: 04 72 44 83 62
	`secretariat.e2m2@univ-lyon1.fr`
		Mme Sandrine Charles
		Université Claude Bernard Lyon 1
		UFR Biosciences
		43 boulevard du 11 Novembre 1918
		69622 Villeurbanne CEDEX
		Tél:
		`sandrine.charles@univ-lyon1.fr`

EDISS	Interdisciplinaire Sciences-Santé
	`http://ediss.universite-lyon.fr`
	Sec: Bénédicte Lanza
	Bât: Atrium, UCB Lyon 1
	Tél: 04 72 44 83 62
	`secretariat.ediss@univ-lyon1.fr`
		Mme Sylvie Ricard-Blum
		ICBMS - UMR 5246 CNRS - Université Lyon 1
		Bâtiment Raulin - 2ème étage Nord
		43 boulevard du 11 Novembre 1918
		69622 Villeurbanne CEDEX
		Tél: 04 72 44 82 32
		`sylvie.ricard-blum@univ-lyon1.fr`

INFOMATHS	Informatique et Mathématiques
	`http://edinfomaths.universite-lyon.fr`
	Sec: Renée El Melhem
	Bât: Bâtiment Blaise Pascal, 3e étage
	Tél: 04 72 43 80 46
	`infomaths@univ-lyon1.fr`
		M. Hamamache Kheddouci
		Université Claude Bernard Lyon 1
		Bâtiment Nautibus
		43 boulevard du 11 Novembre 1918
		69622 Villeurbanne CEDEX
		Tél: 04 72 44 83 69
		`hamamache.kheddouci@univ-lyon1.fr`

Matériaux	Matériaux de Lyon
	`http://edinfomaths.universite-lyon.fr`
	Sec: Yann De Ordenana
	Bât:
	Tél: 04 72 18 62 44
	`yann.de-ordenana@ec-lyon.fr`
		M. Stéphane Benayoun
		École Centrale de Lyon
		Laboratoire LTDS
		36 avenue Guy de Collongue
		69134 Ecully CEDEX
		Tél: 04 72 18 64 37
		`stephane.benayoun@ec-lyon.fr`

MEGA	Mécanique, Énergétique, Génie Civil, Acoustique
	`http://edmega.universite-lyon.fr`
	Sec: Stéphanie Cauvin
	Bât: Direction INSA Lyon
	Tél: 04 72 43 71 70
	`mega@insa-lyon.fr`
		M. Jocelyn Bonjour
		INSA Lyon
		Laboratoire CETHIL – Bâtiment Sadi-Carnot
		9 rue de la Physique
		69621 Villeurbanne CEDEX
		Tél: /
		`jocelyn.bonjour@insa-lyon.fr`

ScSo	Sciences Sociales: Histoire, Géo., Aménagement, Urbanisme, Archéo., Science po., Socio., Anthropo.
	`https://edsciencessociales.universite-lyon.fr`
	Sec: Mélina Faveton & J.Y. Toussaint (INSA)
	Bât:
	Tél: 04 78 69 72 79
	`melina.faveton@univ-lyon2.fr`
		M. Bruno Milly
		Université Lumière Lyon 2
		86 Rue Pasteur
		69365 Lyon CEDEX 07
		Tél:
		`christian.montes@univ-lyon2.fr`

Résumé

Les techniques de synthèse de haut niveau permettent aux programmeurs non spécialistes de générer des descriptions de circuits numériques en utilisant des langages de programmation généralistes. Cependant, les outils existants ne supportent qu'un petit nombre de formats numériques et un petit nombre d'opérateurs standards. Cette thèse présente plusieurs techniques pour rajouter le support de nouveaux formats et de nouveaux opérateurs. Dans un premier temps, l'étude se focalise sur ce qui est réalisable en se restreignant aux fonctionnalités de métaprogrammation du standard C++ supporté par les outils HLS. Une bibliothèque d'opérateurs élémentaires pour les formats IEEE-754 et posit de taille arbitraire est proposée. Elle sert de base à une étude de cas comparant le coût matériel de l'implémentation de ces deux formats. L'implémentation d'évaluateurs de fonctions mathématiques arbitraires se heurte aux limites de la première approche. Dans un second temps, l'étude se porte sur les possibilités offertes par la modification du flot de compilation HLS, avec comme objectif de supporter cette fonctionnalité. Une bibliothèque permettant au développeur de spécifier des opérateurs pour approximer des fonctions arbitraires en précision arbitraire est présentée. Deux approches pour l'interfaçage de cette bibliothèque avec les outils de HLS sont proposées, selon que l'on a ou pas accès aux sources des compilateurs HLS.

Abstract

High-level synthesis allows non-specialist software developers to generate digital circuit descriptions using high-level programming languages. However, existing tools support only few numerical formats and standard arithmetic operators. This thesis introduces many technics to support new formats and new operators. First, the study focuses on what is doable using only the C++ meta-programming constructs supported by main HLS tools. A library of elementary operations for IEEE-754 and posit formats of arbitrary sizes is introduced. It is used as a support to compare the hardware cost of implementing these formats. The implementation of evaluators of arbitrary mathematical functions reaches the limit of the meta-programming approach. A second step is then to study the possibility that are brought by modifying the HLS compile flow. Implementing arbitrary function evaluator is the objective of this study. A library allowing developers to specify operators that approximate arbitrary functions at arbitrary precision is introduced. Two methods are developed to interface this library with the HLS tools, depending on whether the HLS compiler sources can be modified.

Acknowledgement

It has been a real pleasure to work on this thesis. While there have been many ups and downs along the way[1], it has been an interesting journey, and many people have contributed making the interesting part of it more interesting, and the least enjoyable parts, well, less less enjoyable.

Among these people I would like to thank my supervisor, Florent de Dinechin, for the many insightful discussions and the support I could get from him, starting from my first internship on FloPoCo up to the end of the thesis defense.

I am also grateful to Ronan and Gauthier, who I enjoyed working with at Xilinx/AMD. In addition to providing interesting points of view on software development, they have been of great help for the development of the different backends for MArTo arbitrary function support.

I am not sure if I need to be grateful to SARS-CoV-2 without which I would probably not have met them.

I am also thankful to 小泉さん, my former supervisor at NTT Multimedia Intelligence Laboratory, for his advice about perseverance and failure relativization, which proved very valuable these last few years.

Working on this thesis wouldn't have been half as nice without the good mood of the office. So thank you to Agathe, Benoît, Diane, Lélio, Marie, Matthias, Orégane, and Pierre for being such nice colleagues!

While I am of course grateful to all my family, a special mention goes to my brother and sister for their continuous support. Another one for the logistic support I received from my parents and my family in Lyon.

He (probably) did not do it on purpose, but Mochi has also been of a great help. Indeed, real-life cats are clearly underrated compared to rubber ducks for debugging purposes.

Finally, despite their constant efforts to give me more time to work on the manuscript, I do *not* thank the SNCF.

[1] The interested reader can discover the (terrible ((Ph.D. student) universe)) in the abundant related literature on the web.

Contents

List of Figures

List of Tables

List of Listings

Introduction

Transistor density on integrated circuits has stopped to follow Moore's law predictions around year 2010. This is around the same time that clock rate ceased to be improved by new generations of central processing units (CPU). As a result, the main selling point of new generations of desktop and server CPUs has shifted towards more parallelism and more hardware-supported operations. Compute-intensive applications benefit from these two improvements. Indeed, hardware-accelerated mathematical functions and fine-grained parallelism (such as Single Instruction Multiple Data (SIMD) operations or dynamic reordering of instructions for superscalar execution) reduce the runtime of one kernel iteration, while coarse-grain parallelism (through multiple cores or simultaneous multithreading) allows running multiple independent kernel invocations simultaneously, hence reducing the total execution time.

However, the set of numerical formats and operations supported by CPUs does not necessarily fit tightly the application needs. Moreover, computation kernels might expose some parallelism at a degree that cannot be handled properly via CPUs.

In order to further accelerate applications suffering from this kind of CPU limitations, application-specific integrated circuits (ASIC) can be developed. Two main barriers limit this development. The first one consists in the fixed costs associated with custom integrated circuit production. This barrier can be lowered by using Field-Programmable Gate Arrays (FPGAs) instead of ASICs, at the price of some performance loss. The second barrier, common to ASIC and FPGAs, is the required expertise needed to design those circuits.

Lowering the requested experience to develop this kind of hardware is one of the objectives of High-Level synthesis tools. These tools allow building a circuit from a high-level behavioral description written in a classical

programming language, such as C++. This thesis focuses on these tools.

While HLS tools allow software developers to write circuits easily, their support for numerical formats and mathematical functions is limited to those provided by the source language. For instance, the only floating-point formats supported by C++-based HLS tools are the 32-bits and 64-bits IEEE-754 formats `float` and `double`. Yet, using application-specific numerical formats and operators is a way to improve the quality of results (QoR, which regroups design latency and surface) of circuit designs. For instance, as elementary operations on narrower formats operate on smaller bit vectors, their latency is reduced compared to the same operation on wider a format. Besides, they also occupy less chip area, which leaves more space for packing more operators on the chip. This eventually allows running more operations in parallel. Using for each computation the narrowest format that still allows to verify the overall application accuracy constraint is then a good way to optimize this application.

The work presented in this thesis aims at providing support for custom numerical formats and operators in an HLS context. This support takes the form of a portable library, MArTo, that a developer can exploit to define and use custom numerical types in HLS code. This library supports both IEEE-754 and posit numerical formats, which are presented along with other numerical formats, arithmetic operator formalization and FPGA architecture in chapter 1. Using the MArTo allows fast evaluation of the performance/accuracy tradeoff entailed by the usage of a given numerical for a given application. MArTo is built on top of a second library, hint, that abstract operations on arbitrary size integers. Both of these libraries use C++ meta-programming constructs that are supported by HLS tools to implement the basic arithmetic operations of the supported formats. This design and the usage of the two libraries is further detailed in chapter 2, which also present a case study comparing the implementation cost of the two supported floating-point formats, IEEE-754 and posit.

A second optimization opportunities with regard to computation is the evaluation of mathematical functions. When only a limited set of mathematical functions is provided to the user (as it is the case with the C libm library), evaluating a composed function such as $x \mapsto \cos(\sin(x))$ requires composing evaluation of library-supported functions (computing $\sin(x)$ first, before evaluating $\cos$ on the result for the example function). However, ad-hoc function evaluators can be implemented to evaluate directly arbitrary expressions, allowing reduced latency and better control on precision compared to the composition approach. While generating such evaluators is a well studied problem, especially when both input and output format of the evaluator is fixed-

point, this functionality is not supported by HLS tools.

The architecture of such evaluators require pre-computed values, such as coefficients of a polynomial approximating the function to evaluate. Obtaining these values is non-trivial, which prevent computing them in a pure meta-programmatic fashion. Instead, it is more desirable to use existing tools that can provide state-of-the-art evaluation plans for arbitrary functions. However, evaluation plan has to be determined before performing the HLS compilation step.

Two methods to modify the HLS compilation flow are described in chapter 3. The first approach does not require direct modification of the HLS compiler. This makes it quite portable, but it requires that extra compilation steps are performed by the user, and it suffers from some fundamental limitations. The second approach consists in adding built-in support for arbitrary expression evaluation directly in the compiler. This method solves the issues of the first approach, but is not portable anymore. This disadvantage is attenuated by using a compiler-independent expression compiler that can be used by multiple HLS compilers.

Evaluation of arbitrary function is only one of the many possible optimized high-level primitives that could be made available to HLS developers, while providing automatic optimized implementation generation. Some perspective on building such “Real High-Level Synthesis” systems are discussed in the conclusion of this thesis.

1 Context

The central goal of the work presented in this thesis is to enable the development and the integration with software tools of high quality application specific arithmetic hardware operators. Here, hardware refers to implementation as digital circuits. This work focuses on digital circuits at the logic level. At this abstraction level, digital circuits consist in collections of logic signals combined to produce something useful. How exactly these signals are combined, and how to produce these combinations is detailed in section 1.4. The ”something useful” that results from this combination depends on the application. In the case of this work, it consists in computing the result of some computation. This requires to have some way to interpret the logic signals of a digital circuit as a numerical value. This is the aim of numerical formats, which are presented in section 1.2. Having a representation for numbers allows to perform computation on them. This is the purpose of hardware arithmetic operators, presented in section 1.3. Finally, in the context of this thesis, hardware operators are implemented on Field-Programmable Gate Arrays (FPGAs), the architecture and programming model of which are presented in section 1.4. Before delving into these details, section 1.1 briefly presents the core concepts of logic signal and logic vector, and introduces the notations that are used in the rest of this work.

1.1 Logic signals, logic vectors

At the logic level, a logic signal is the core element that digital circuits handle. A logic signal is at anytime in one of two states, called either high and low, true or false, set and unset, or ‘1’ and ‘0’. In this work, logic signals are modeled as variables from $\mathbb{B} = \{0, 1\}$ and are named with lowercase letters.

It is sometime convenient to reason on a whole group of signal as one en-

tity. These signals can be concatenated to form logic vectors. In this work, logic vector names start with a capital letter.

Vector component access is denoted using squared bracket notation. For instance, $V[2]$ is the third component of the vector V. The width of a logic vector is its number of elements, and is expressed in bits.

In this work, vectors will often be represented as bit string. With this representation, the vector first element is the rightmost one. That is, if $V = 01$, then $V[0] = 1$ and $V[1] = 0$.

1.2 Numerical formats

Performing computation with digital circuits involves to manipulate numbers with them. As these circuits only manipulates logic vectors, it is necessary to be able to use them to represent numbers. This is the role of numerical formats, which define decoding schemes to interpret a logic vector, the *representation* as a *value*. Table 1.1 gives an example of an arbitrary numerical format for 2-bits vectors. A value is associated to each representation.

Table 1.1: Example of an arbitrary numerical format on 2-bits vectors.

Representation	Associated value
00	$-\frac{2}{3}$
01	2
10	−8
11	12

All encoded values of table 1.1 example are real numbers. This is the canonical case, but some formats might also encode non-real special values, such as infinities.

Formally, a format $\mathcal{F}$ is defined by

- its representation width $n_{\mathcal{F}}$ (the size of its representations) (2 in case of the example of table 1.1),
- its (possibly empty) special value set $\mathbb{S}_{\mathcal{F}}$. This is the set that contains the non-real values that should be representable by the format such as infinities. The format domain is then

  $$\mathbb{D}_{\mathcal{F}} = \mathbb{R} \cup \mathbb{S}_{\mathcal{F}}$$

  This set is empty in the example.
- Its decoding function $d_{\mathcal{F}} : \mathbb{B}^{n_{\mathcal{F}}} \to \mathbb{D}_{\mathcal{F}}$ that maps a representation to the represented value. In the case of the example, it corresponds to reading the value associated to the representation from the table.

- The set of representable domain values $\mathbb{F}_{\mathcal{F}}$ (the image of the representation domain through the application of the decoding function). This corresponds to the set of values appearing in the second column of table 1.1.

Due to the finite number of representations, $\mathcal{F}$ cannot represent exactly all real values. Rounding functions are used to map a representable value to any real value. Two basic directed rounding functions can be defined:

- Round towards $+\infty$ (or rounding up):

$$o^{\uparrow}_{\mathcal{F}} : \mathbb{D}_{\mathcal{F}} \to \mathbb{F}_{\mathcal{F}}$$
$$v \mapsto \begin{cases} \min i \in \mathbb{F}_{\mathcal{F}} \mid i \geq v, & \text{if } v \in \mathbb{R} \\ v & \text{otherwise} \end{cases}$$

- Round towards $-\infty$ (or rounding down):

$$o^{\downarrow}_{\mathcal{F}} : \mathbb{D}_{\mathcal{F}} \to \mathbb{F}_{\mathcal{F}}$$
$$v \mapsto \begin{cases} \max i \in \mathbb{F}_{\mathcal{F}} \mid i \leq v, & \text{if } v \in \mathbb{R} \\ v & \text{otherwise} \end{cases}$$

Figure 1.1 shows the representable elements of the format defined by table 1.1, and illustrates the value returned by the two directed rounding function on this format.

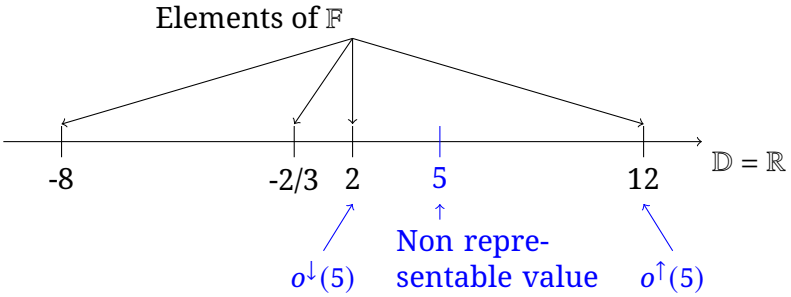


Figure 1.1: Representable values and rounding function illustration for example format of table 1.1

The $\mathcal{F}$-representable neighborhood of a real value v is defined as:

$$\text{rnb}_{\mathcal{F}}(v) = \left\{ o^{\downarrow}_{\mathcal{F}}(v), o^{\uparrow}_{\mathcal{F}}(v) \right\}$$

When v has a representation in $\mathcal{F}$, this set only contains v. As figure 1.1 shows, for the example format, $\text{rnb}(5) = \{2, 12\}$.

This neighborhood is used to define two rounding modes that this work focus on. Rounding to nearest, as its name suggests returns the element of the neighborhood with the smallest distance to the exact value.

$$o^{\mathrm{N}}_{\mathcal{F}}(v) = \underset{x \in \mathrm{rnb}_{\mathcal{F}}(v)}{\operatorname{argmin}} |x - v|$$

In case of the illustration example, $o^{\mathrm{N}}(5) = 2$ as the distance from 5 to 2 is smaller than the distance from 5 to 12.

A tie-breaking rule determines which of the neighbor should be returned when the real value is a midpoint, i.e. when the value is equidistant to the two neighbors.

The second rounding mode is faithful rounding. It only requires that one of the neighbors is returned.

Numerical formats with arbitrary mapping such as the example of table are difficult to reason with and to use when the representation size grows. For larger formats, it is thus desirable to get some hardware-exploitable regularity in the representation-value mapping. Commonly used regular formats are fixed-point and floating-point. These specific formats are detailed in the following sections.

1.2.1 Fixed-point

Unsigned fixed-point formats

Unsigned fixed-point formats values are represented using binary positional notation. Binary positional notation is very similar to the usual decimal notation: the only difference is the radix. A number written in this system is expressed as a sum of powers of two. The fractional point in this system is used to determine which power of to corresponds to which bit. The bit immediately on its left is associated to 2^0 and the associated weight doubles each step leftward. Reciprocally, it is halved each step rightward. A bit's position p is its distance with the 2^0 weighted bit, with a positive increase in the left direction. The weight associated to such a bit is simply 2^p.

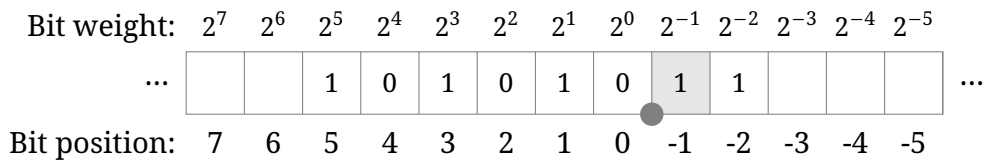


Figure 1.2: Binary positional notation of 42.75

Figure 1.2 illustrates this notation. The position of the bit is reported on

the axis below the representation, and their weight is written above them. For instance, the greyed bit is at position -1 and its weight is 2^{-1}. All non-written bits are implicit zeros, so for instance bits with position above 6 are all zeros. The represented value is the weighted sum of its bits so here

$$v = 2^5 + 2^3 + 2^1 + 2^{-1} + 2^{-2} = 42.75$$

The value v of a number written in binary positional notation with bits b_i at position i is

$$v = \sum_{i=-\infty}^{+\infty} b_i \cdot 2^i$$

However, having an infinity of representation bits is not physically possible. Practical unsigned fixed-point numeric formats are defined by their high and low bit positions, respectively m and l, such that $m \geq l$. Such a format is denoted $\mathrm{ufix}(m, l)$, and has a representation width of $n_{\mathrm{ufix}(m,l)} = m - l + 1$. Its representation is a "chunk" of the infinite positional notation, which is illustrated by figure 1.3. The decoding function is hence a restriction on the available representation bits of the previous equation, and the value v is obtained with

$$v = \sum_{i=l}^{m} 2^i R[i - l]$$

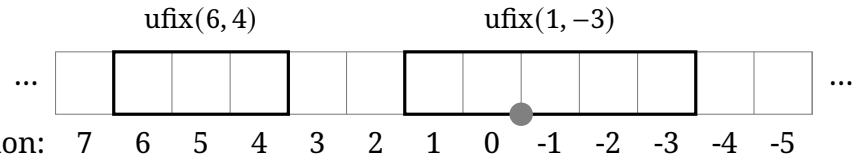


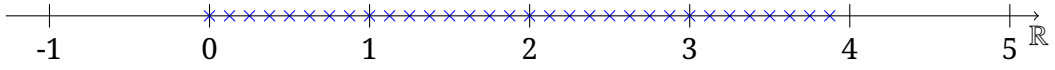
Figure 1.3: Superposition of two fixed-point representation windows with the infinite positional representation.

The weight of the rightmost bit of the representation is called the unit in the last place (ulp) of the format. This is the smallest difference that exists between two distinct representable values. Formats of the type $\mathrm{ufix}(m, 0)$ form a notable group : they allow to represent all positive integers up to $2^{m+1} - 1$. By extension, a value v from $\mathrm{ufix}(m, l)$ can be thought as a scaled integer

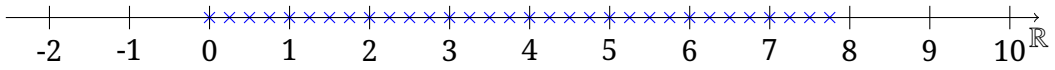
$$v = i \cdot 2^l$$

with i an integer representable on $\mathrm{ufix}(m - l, 0)$. The plots of representable values of two 5-bits fixed-point formats of Figure 1.4 illustrates this. The rep-

resentable value set of one format is a scaled version of the second format representable value set.



(a) ufix$(1, -3)$



(b) ufix$(2, -2)$

Figure 1.4: Visualization of representable values for two 5-bits unsigned fixed-point formats.

Alternative equivalent parametrizations exist for fixed-point formats. One frequently used parametrization, the $Qi.f$ notation, describes these formats with two integers i and f which are respectively the number of integer and fraction bits. For instance ufix$(4, -2)$ would be denoted $Q5.2$. While this notation is good when the decimal point is included in the described formats, it becomes quite unintuitive when it is not. For instance, the format ufix$(-3, -7)$ is named $Q-2,5$ in this formalism, the -2 integer bits conveying the fact that in addition to not having integer bits, the format also "lacks" the two leftmost fractional bits. For this reason, this work uses the ufix(m, l) notation.

Next section present extensions to this system that allow handling negative values.

Signed fixed-point formats

Two main systems coexist to add support of negative values to fixed-point formats. These two systems have the same parameters m and l as unsigned fixed-point. In these two systems, the leftmost bit is the sign bit s. The remaining low bits consist in a representation in ufix$(m-1, l)$. In the following, R_L denotes these bits and v_L the value that they represent. The decomposition of a signed fixed-point value is detailed on figure 1.5.

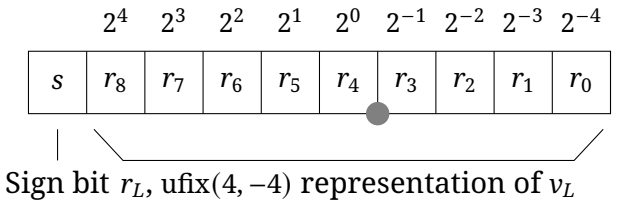


Figure 1.5: Decomposition of a signed fixed-point format with $m = 5$ and $l = -4$

The two systems differs in how the sign bit is interpreted in the decoding function. In the most natural one, the sign-magnitude encoding, it is used to determined if the encoded value is v_L (when the sign bit is '0') or its opposite $-v_L$ (when the sign bit is '1'). Formally, the represented value v_{sm} is given by the relation

$$v_{\text{sm}} = (-1)^s \cdot v_L$$

It is straightforward to see that the opposite of each representable value is also a representable value.

While it is quite intuitive, this encoding system has two main disadvantages. First, the algorithm to get the representation of the successor is quite complicated, as it depends on the sign bit. Indeed, in the general case (when both the value and its successor have the same sign), r_L should be replaced by the representation of its successor when s is unset, and by the representation of its predecessor when s is set. And a few additional corner cases have to be handled. One of them is directly related to the second disadvantage, which is that these formats have two signed representations for 0.

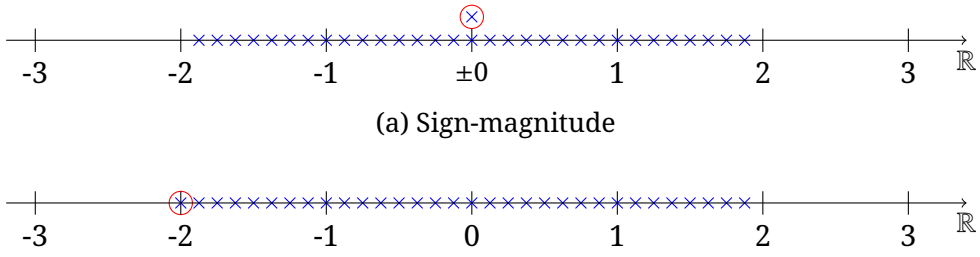
The second system, two's complement, keeps the weighted-sum principle of unsigned fixed-point format. It only differs with this format by the weight of the leftmost (sign) bit, which is -2^m instead of 2^m. The represented value v_{2c} is then

$$v_{2c} = -2^m + v_l$$

With two's complement, the successor computation is greatly simplified and there is no redundant representation for one given value. Due to that and to the fact zero is still representable in the format, opposite of representable values is not always representable itself. This asymmetry is visible on Figure 1.6b which shows the representable value for a two's complement signed fixed-point format defined with $m = 1$ and $l = -3$. Value -2 is representable, while 2 is not. By contrast, representable values of sign-magnitude fixed-point with same parameters (represented on figure 1.6a) cannot represent -2 either and replace this by a second representation for 0. In the following, $\text{sfix}(m, l)$ will denote 2's complement fixed-point format of parameters m and l.

Similarly to the unsigned case, $\text{sfix}(m, 0)$ defines a notable group of formats able to represent all integers i such that

$$-2^m \leq i < 2^m$$



(a) Sign-magnitude

(b) Two's complement

Figure 1.6: Visualization of representable for the two signed fixed-point schemes with $m = 1$ and $l = -3$. Scheme-specific representable values are circled in red.

Values v of $\mathrm{sfix}(m, l)$ can also be seen as scaled integer:

$$v = i \cdot 2^l$$

with i representable on $\mathrm{sfix}(m - l, 0)$.

Biased fixed-point formats

It is possible to add an "implicit" constant offset (i.e. independent of the representation) to the summation performed to decode a fixed-point representation. This shifts the representable values by the added amount. One example of such biased fixed-point representations is the Half-Unit Biased (HUB) encoding families [1]. A HUB representation is defined similarly to fixed-point formats by its high and low bit positions m and l. The Unit in the format name refers to the scaled integer vision of the fixed-point the represented number: HUB decoding function adds a constant offset of half of a scaled unit in the decoding process. The bias value is then 2^{l-1}.

It has the advantage that rounding a value to the nearest HUB representation only consists in truncated all bits of position lower than l.

Another application of biased fixed-point is presented along IEEE-754 encoding scheme.

Arbitrary radix fixed-point

Using binary positional notation on digital circuits that manipulate binary signal seems natural. It is however possible to represent number using different radixes.

Radix β notation decomposes a value v as a sum of weighted β-digits, each

weight being a power of β depending on the position i of the β-digit k_i.

$$v = \sum_i k_i \beta^i$$

Similarly to binary fixed-point, the representation in β radix fixed-point stores a sequence of β-digits. As the circuit can only handle logic signal, these β-digits are themselves encoded in binary.

When $\beta = 2^p$, the representation does not differ from radix 2 representation, only the interpreted meaning of representation bits differs. Indeed, in this case the contribution of each β-digit can be seen as an integer scaled by a power of two, and none of these contributions overlap. Figure 1.7 illustrates this by showing two possible interpretation of the binary-coded representation of 42.75.

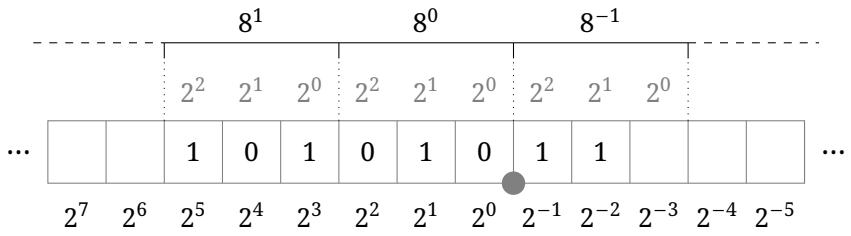


Figure 1.7: Binary and binary-coded octal positional notation of 42.75

When β is not a power of two, the binary encoding of the β-digit is wasteful. Indeed, $w = \lceil \log_2(\beta) \rceil$ bits are required to store one β-digit, but only β out of the 2^w possible combination of those bits are effectively used. This is illustrated with figure 1.8 that gives the binary-coded radix-10 representation of the value 42.75. This representation requires 15 bits, where the binary representation only requires 8 bits.

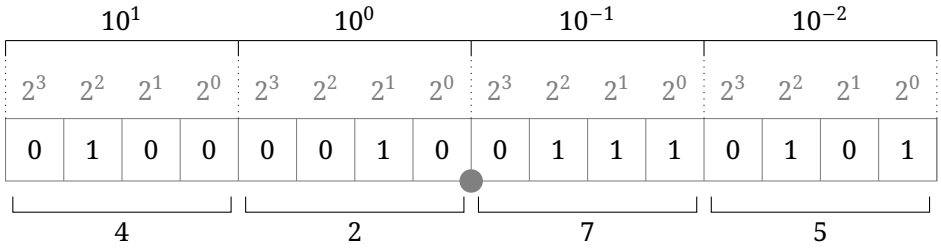


Figure 1.8: Binary-coded decimal positional notation of 42.75

So in one case the representation does not change, and in the other the representation is wasteful, so why using such a system ? If $l = 0$, there is definitely no interest to use a radix that is not a power of two, as integers are

all representable using radix 2. With $l < 0$ however, the set of exactly representable values changes. For instance 49.3 is not exactly representable in binary, but is representable in radix 10. So financial application for instance might be interested not to have representation rounding errors when they perform computation over amount of money which are expressed in decimal notation.

One way to limit the waste of representation bits is to use a radix that is very near to the power of two immediately superior to it. For instance, it might be more interesting to use radix 1000 (10^3) than radix 10[2]. Indeed, with radix 1000, 1000 out of the 1024 possible combination of the 10 bits required to store a 1000-digit effectively used. In the other hand, only 10 out of the 16 possible combination of the 4 bits required to store a radix-10 digit are used.

1.2.2 Floating point

Fixed-point formats lack dynamics: representing both high magnitude and very small numbers is not possible without using very wide (and impractical) formats. Floating-point formats solve this issue by using a representation similar to the so-called scientific notation. A value v is represented as a significand M scaled by some power E of a given radix β.

$$v = M \cdot \beta^E$$

The discussion about radix value is analog to the fixed-point case. In the present work, only binary ($\beta = 2$) floating-point formats are considered.

A basic floating-point encoding could be defined by a pair of signed fixed-point formats $(\mathcal{F}_E, \mathcal{F}_M)$, respectively the exponent and significand format. The representation is the concatenation of a representation r_E and r_M from each of the formats. The represented value v is obtained from the two fixed-point values:

$$v = d_{\mathcal{F}_M}(r_6) \cdot 2^{d_{\mathcal{F}_E}(r_E)}$$

The representation width of $\mathcal{F}_M$ specifies with how many "significant digits" the format can represent a number. This is called the precision of the floating-point format. In the other hand, the representation width of $\mathcal{F}_E$ is related to the dynamic range of the format. For two floating-point formats with identical precision, the one with higher dynamic range will be able to represent higher and lower magnitude values.

Figure 1.9 gives a possible representation for the value 42.75 in such a

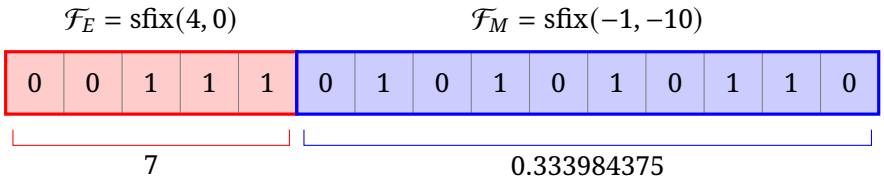


Figure 1.9: Basic floating-point representation of 42.75 as 0.333984375×2^7

format. However, this format has redundancy. Indeed, if the significand representation does not start and end with a 1, it is possible to shift it and reduce or increase the exponent to keep the same value. For instance, 42.75 can also be represented as 0100000010101011 in the format from figure 1.9. Most of the time this redundancy is undesirable, as it is "wasting" codes that could be used to represent other values. Following sections describes well-defined floating-point encoding schemes that avoid this waste.

IEEE-754 encoding scheme

Compared to fixed-point where possibility of variation is quite limited, floating-point offers a lot of freedom in their conception. In the early days of computing, this lead to each CPU manufacturer having its own specific format. This is reflected in the first version of ANSI C (C89) standard. In this version the definition of floating-point types in C is very scarce. It is only specified that C has three floating point type (`float`, `double` and `long double`), verifying the property

$$\mathbb{F}_{\texttt{float}} \subseteq \mathbb{F}_{\texttt{double}} \subseteq \mathbb{F}_{\texttt{long double}}$$

Outside from this, "the representations of floating-point types are unspecified". In addition to have different representation, the semantic of operations was also not consistent between floating-point implementations. As a result, standard compliant code could produce completely different outputs when executed on different machines.

IEEE-754 [3] is the standard that resulted from the industry effort to eliminate this heterogeneity. Its first edition was issued in 1985 and defines two fully specified binary floating-point formats (which C99 later adopted as representation for the C `float` and `double` types), and the 'extended" format family. It also defines the base operations that should be provided by compliant implementations. The 2008 revision introduced the concept of interchange formats, that are intended solely for the purpose of sharing a value, but not for performing computation. It also specified three decimal floating-point formats.

An IEEE-754 binary format representation is composed of three fields:

- The sign bit s on 1 bit,
- the biased exponent E on w_E bits,
- the fraction part F on w_F bits.

Table 1.2: Field widths for IEEE-754 standard formats.

Format	w_E	w_F
binary16	5	10
binary32	8	23
binary64	11	52
binary128	15	112
binary256	19	236

Field sizes for fully specified formats are given on table 1.2. The special value set of IEEE-754 binary encoding schemes contains four values. The first two, positive and negative infinities are the overflow markers. These are the values that are returned when an operation has a result which is bigger (resp. smaller) than the biggest (resp. smallest) representable values. The third and fourth ones are quiet and signalling Not-a-Number (NaN). They arise as a result of invalid operation, such as taking the logarithm of a negative number. The difference between signaling and quiet NaNs concerns the environment surrounding the computation, so for the rest of this work they will be considered as a unique value.

The special values are all encoded with all the exponent field bits set to one. When the fraction bits are set to zero, the represented value is infinity (with signedness depending on the sign bit). For instance, in binary16 the representation for $-\infty$ is 1111110000000000. As soon as one bit of the fraction field differs from zero, the represented value is NaN. Since the 2008 standard revision, the leftmost bit of NaN representation indicates whether it is a quiet NaN. In binary16, quiet NaN are then represented as s111111xxxxxxxxx The remaining bits (marked with `x` in the previous example) are called the NaN payload and can be used to store application-defined data. This usage is simplified with the new operations `setpayload` and `getPayload` introduced in the 2019 standard revision.

When at least one bit of E is set to 0, the represented value is a numerical value. The decoding process is very similar to what has been introduced in the previous section. A first difference is that it is using sign-magnitude encoding. The second important difference is that the fraction is normalized in order to avoid redundant value representations. That is, an implicit bit i at position 0

is prepended to the fraction, the complete explicit significand is $S_e = i.F$. In the general case, the value of i is 1.

Finally, the third main difference is that a biased exponent is stored in the representation. For a binary format of exponent width w_E, the bias value is

$$b = 2^{w_E - 1} - 1$$

The actual exponent is $E - b$. Having a biased exponent ensure that the representation ordering is the same as if they were interpreted as sign-magnitude representation of integers. Indeed, the biased exponent is an unsigned integer to which is concatenated the fraction that is also an unsigned integer. For the same exponent, the representation with the biggest fraction represent the biggest value, and when the exponent differs, the representation with the biggest exponent encodes the biggest value. When the sign differs only the sign is relevant.

This property allows efficient comparison of IEEE-754 values by using integer comparison on the representation. This also helps to compute rounding, as a rounding neighborhood is constituted by values having successive representations.

The represented value is then

$$v = (-1)^s \cdot 1.F \cdot 2^{E-b}$$

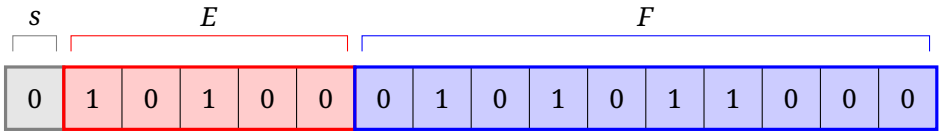


Figure 1.10: Unique IEEE-754 binary16 representation of 42.75.

Figure 1.10 gives the encoding in binary16 of the value 42.75. The value stored in the exponent is 20. As the exponent field has a width of 5, the bias value is 15, so the unbiased exponent is 5. Stored fraction is 0101011000, so the significand is 1.0101011000 (1.3359375 in decimal). The represented value is then $1.3359375 \cdot 2^5 = 42.75$.

Having always the implicit bit set to one does not allow representing 0. This is solved by adding a special case for numeric values called the subnormal range of the format. When all bits of E are set to 0, the implicit bit becomes 0. To avoid a gap between subnormal and normalized values, the exponent that corresponds to subnormal significand is the same as the minimal normal exponent. So the real value decoding process involves computing the actual

significand

$$M = i.F$$

and the unbiased exponent

$$E_u = E - b + (1 - i)$$

with

$$i = \begin{cases} 0 & \text{if } E = 0 \\ 1 & \text{otherwise} \end{cases}$$

to get the encoded value

$$v = (-1)^s \cdot 2^{E_u} \cdot M \quad .$$

In addition to allowing the representation of zero, the subnormal mechanism also brings interesting properties to the format. One of which is the following equivalence:

Property 1 *Given f an IEEE-754 binary numerical format, and o a valid IEEE-754 rounding mode*

$$\forall (x, y) \in \mathbb{F}_f^2,\ o(x - y) = 0 \Leftrightarrow x = y$$

Property 1 means that the difference between two representable values cannot round to zero unless the two values are equal. This property can be exploited to optimize code (for instance replacing left member of equivalence by right member, which replaces computing one subtraction and one comparison by only one comparison). Figure 1.11 gives an illustration of why subnormal mechanism is required to get this property to work. Difference between two very near number that have exponent near to the minimal exponent can result in a value that is smaller than $2^{E_{\min}}$, with $E_{\min}$ the minimal normal exponent. Without subnormal mechanism, these values are not exactly representable, so depending on the rounding mode, the result could be zero even if the two values are not equal.

Table 1.3 summarizes the different decoding ranges for IEEE-754 binary16 format.

Operations

The IEEE-754 standard defines five rounding modes for elementary operations. The directed rounding modes towards $+\infty$ and $-\infty$ corresponds to the

Table 1.3: Interpretation of IEEE-754 binary16 encoding.

Representation			Value	Denomination
1	11111	1111111111	Quiet NaN	Special values
1	11111	...		
1	11111	1000000000		
1	11111	0111111111	Signaling NaN	
1	11111	...		
1	11111	0000000001		
1	11111	0000000000	$-\infty$	
1	11110	1111111111	$-1.F \cdot 2^{E-b}$	Negative normal range
1	...	...		
1	00001	0000000000		
1	00000	1111111111	$-0.F \cdot 2^{E_{\min}}$	Negative subnormal range
1	00000	...		
1	00000	0000000001		
1	00000	0000000000	-0	
0	00000	0000000000	$+0$	Positive subnormal range
0	00000	0000000001	$0.F \cdot 2^{E_{\min}}$	
0	00000	...		
0	00000	1111111111		
0	00001	0000000000	$1.F \cdot 2^{E-b}$	Positive normal range
0	...	...		
0	11110	1111111111		
0	11111	0000000000	$+\infty$	Special values
0	11111	0000000001	Signaling NaN	
0	11111	...		
0	11111	0111111111		
0	11111	1000000000	Quiet NaN	
0	11111	...		
0	11111	1111111111		

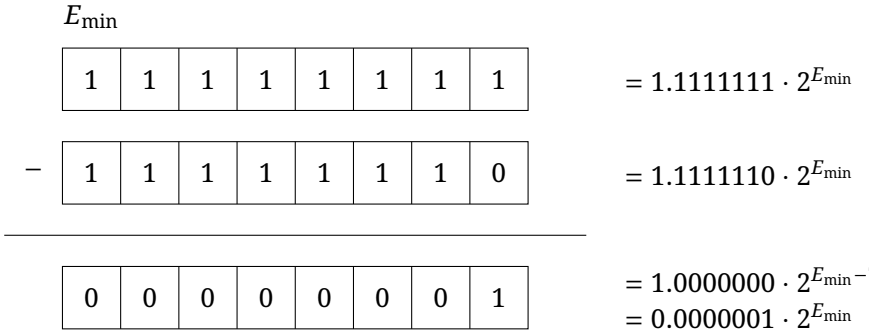


Figure 1.11: Subtraction of two floating-point values of minimal normal exponent $E_{\min}$.

one defined in section 1.3, with the notable exception that values above the maximal representable numerical value are rounded to $+\infty$ (resp. $-\infty$). A third rounding mode, towards 0, is defined as follows:

$$o_f^0(v) = \begin{cases} o_f^{\uparrow}(v) & \text{if } v < 0 \\ o_f^{\downarrow}(v) & \text{otherwise} \end{cases}$$

Round-to-nearest is also supported with two tie-breaking rules:

- Ties-to-even rule specifies that the neighbor with representation ending with a 0 bit should be returned. This allows an unbiased rounding error, as there are as many midpoints above their ”nearest even” than midpoints below.
- Ties-to-away returns the neighbor with higher magnitude. This rule has ”polarized” rounding error: negative midpoints are always rounded to lower values, where positive midpoint are always rounded to higher values. It is however a bit less expensive to compute, as only one extra bit after the last fraction bit of the result has to be known. In comparison, computing correct ties-to-even also requires knowing if the value is exactly a midpoint or not.

Compared to the fixed-point case, the distance between two successive representable values is not constant. Reasoning in terms of absolute error with floating-point is then quite complicated, so it makes more sense to use relative error. For a real value v, and a rounding function o, the relative error is

$$\epsilon = \frac{o(v) - v}{|v|}$$

Inside a *binade*, the set of all representable normal values that share the same exponent, the distance between successive values is constant. For

round-to-nearest, the biggest absolute error is obtained for midpoints. On a binade of exponent E in the normalized range, distance between representable values and midpoints is

$$\delta_N = 2^{w_F-1} \cdot 2^E$$

The minimal representable value inside the binade is $m = 2^E$. The relative error committed on the binade is then bounded by

$$\begin{aligned} |\epsilon_{\max}| &= \frac{m + \delta_N - m}{m + \delta_N} \\ &= \frac{2^{E-w_F-1)}}{2^E + 2^{E-(w_F+1)}} \\ &< \frac{2^{E-(w_F+1)}}{2^E} = 2^{-w_F-1} \end{aligned}$$

This maximal relative error does not depend on the binade. For directed rounding, the maximum absolute error inside a binade is twice the maximum absolute error of round-to-nearest, so the bound on relative error is also doubled. The same reasoning applies to each subnormal binade, but considering the effective fraction width of the binade (which takes into account the leading zero bits in the denormalized fraction). Figure 1.12 shows the relative error bound per binade when rounding to nearest IEEE-754 binary16 representation. The relative error is important for very small values (reaching 1 for values that are rounded to zero), and decreases until the normal range is reached, on which the error relative error bound is binade-independent.

IEEE-754 binary32 and binary64 fulfill the role of the all-purpose numeric formats that are used when dealing with non-integral values of unknown scales. Even the high number of redundant representation for NaN has found usage, such as compact type information and value packing in dynamically typed language interpreter, with the technique known as NaN-boxing [4]. However, this redundancy is more annoying with narrower formats, where having more representable values would be preferable. Machine-learning is an example of application class that does not need an important precision but still requires values with an important dynamic range. Multiple encoding scheme have been developed recently to meet the requirement of efficient narrow floating-point representation.

DLFloat[5] is such a format. Its 16-bits representations regular decoding process is those of an IEEE-754 format with $w_F = 9$ and $w_E = 6$. To reduce the overhead of special values, NaN and infinities are fused so the special value set of DLFloat only contains one element, NaN-or-Infinity. This element has

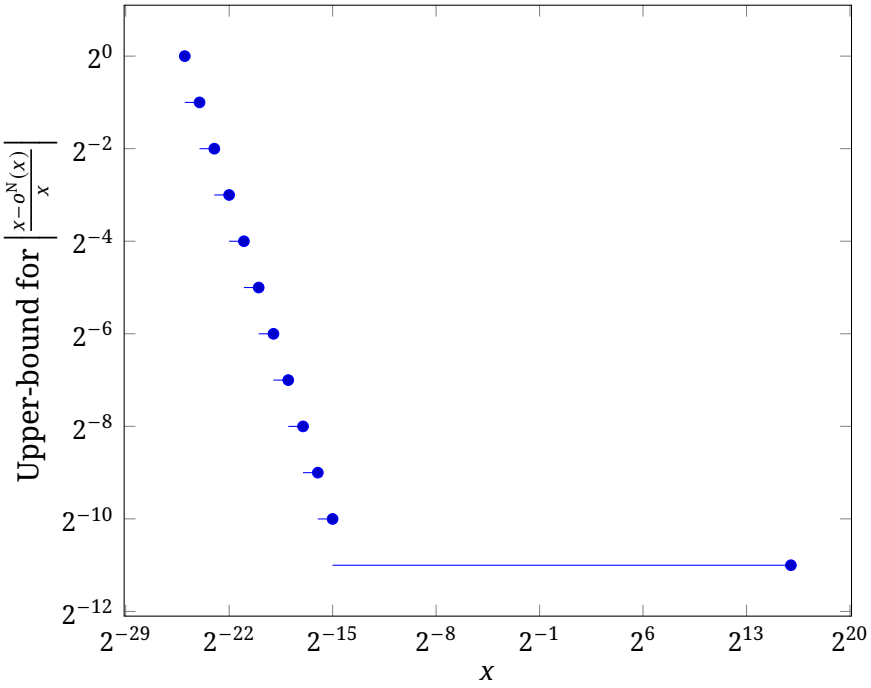


Figure 1.12: Bound on relative-error magnitude for IEEE-754 binary16 representable range.

two redundant representations, that are when the fraction and exponent bits are all set to one (the value of the sign bit is disregarded). All the other representation for NaN and infinity in the regular IEEE-754 scheme are interpreted as normal values, which increase the exponent range of one. The choice of having two redundant representation of one special value instead of having two distinct representation for infinity and NaN is done to simplify the special case handling. For the same objective, the detailed rule of IEEE-754 specifying which zero should be returned is ignored, and the two representations of zero are perfectly equivalent. Finally, subnormal value support is dropped, and the normal range is extended by one instead, which makes zero an irregular value regarding the encoding scheme.

Simpler NaN overhead reduction consists in taking a bigger exponent width for the same total representation width. As special value always reserve one exponent value, increasing the number of available exponents decrease the proportion of redundant NaN representations. This is the approach that BFloat16 [6] (a 16-bit format having an exponent width of 8-bits, similarly to IEEE-754 binary32) or MSFP8 and MSFP9 [7] (8 and 9-bits formats having the same exponent width of 5 bits than IEEE-754 binary16).

Table 1.4 illustrates the trade-off that the three 16-bits formats binary16, BFloat and DLFloat offers in terms of representable range, precision, and pro-

Table 1.4: Precision - dynamic - special-value representation trade-off for mainstream 16 -bit IEEE-like floating-point formats.

Format	w_E	w_F	Precision	# binades	Special value repr.
Binary16	5	10	2^{-10}	40	3.125 %
DLFloat	6	9	2^{-9}	64	$\approx$ 0.06 %
BFloat16	8	7	2^{-7}	161	$\approx$ 0.39 %

portion of representation dedicated to special value encoding. Figure 1.13 gives a visual representation of the precision trade-off offered by these formats.

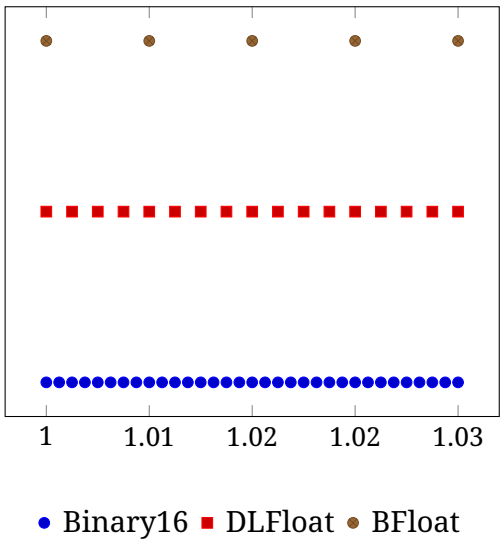
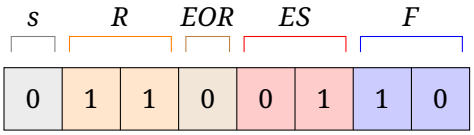


Figure 1.13: Representable values for mainstream 16bit IEEE-like formats on $[1, 1 + 2^{-5}]$

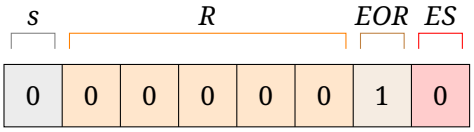
Posit encoding scheme

While previous schemes are all variation over the IEEE-754 binary formats, completely different schemes also exist. This is the case of Posit encoding [8], [9]. Posit encodes floating-point value using a mixed thermometer/positional encoding of the exponent. A Posit format is parametrized by two natural integers, its width n and its exponent shift field width w_{ES}. Regular positive Posit values have their sign bit s, the leftmost representation bit, set to 0. A sequence of R identical bits constitute the regime of the representation, with

$$1 \leq R \leq n-1 \quad .$$



(a) Posit<8,2> encoding of 48



(b) Posit<8,2> representation of 2^{-20}.

Figure 1.14: Posit<8,2> representations decomposition.

When $R < n-1$, the bit following the range acts as an end-of-regime marker. Then up to w_{ES} bits constitutes exponent shift, and all the remaining bits constitutes the fraction bits F. The represented value is decoded using the following relation:

$$v = 1.F \cdot 2^{k2^{w_{ES}}} \cdot 2^{ES}$$

With k being computed out of the regime:

$$k = \begin{cases} R-1 & \text{if } R[0] = 1 \\ -R & \text{if } R[0] = 0 \end{cases}$$

Figure 1.14 presents the decomposition of two positive Posit<8,2> representations. In the case of figure 1.14a, the range is constituted of 2 bits with the value 1, so the encoded value is

$$[1.10]_2 \cdot 2^{1 \cdot 2^2} \cdot 2^1 = 1.5 \cdot 2^5 = 48$$

The regime can extend up to a point where it "pushes out of representation" F and ES bits. In this case, pushed out bits are considered having a value of 0. This is the case on figure 1.14b. Here all the fraction bits and the first ES bit are pushed out of the representation by the regime. The regime is constituted of 5 bits with value 0 so the encoded value is

$$1.0 \cdot 2^{-5 \cdot 2^2} \cdot 2^0 = 1 \cdot 2^{-20}$$

Only exception to this scheme are 0, which is encoded with all bit set to 0, and NaR for Not a Real, the result of invalid operation, that is encoded with the sign bit set to 1 and all other bits set to 0.

The opposite of each representable posit value is also a representable

value. Except for 0, the opposite is obtained by taking the two's complement of the representation.

Posit arithmetic is saturating: operation results that are greater in magnitude that the greatest representable posit value are rounded to this greatest value. Symmetrically, posit operations having a non-zero result with magnitude lower than the lowest non-zero representable magnitude are rounded away from zero.

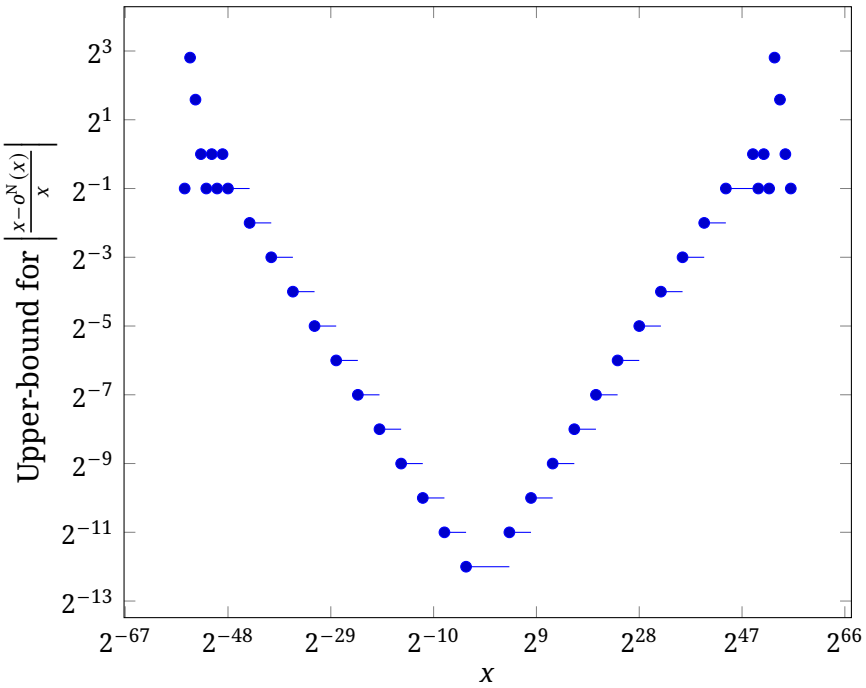


Figure 1.15: Bound on relative-error magnitude for posit<16,2> representable range.

In the most recent version of the posit standard, w_{ES} which was before defined as $\log_2(N)$ for standard posit formats has been fixed to 2 for all the formats. Due to the varying length of the fraction encoding, the relative error of posit value is more binade-dependent than for IEEE binary encoding. Figure 1.15 gives the bound on the relative error for representable posit values of the standard posit<16, 2> format. It can be seen that the format gives a good relative precision for values around 1 (for posit<16,2> the relative precision is strictly better than IEEE binary 16 between 2^-4 and 2^3), but degrades when going to high magnitude exponent, similarly to what happens in the IEEE subnormal range. All exponent in the representable binade range cannot be properly represented (which corresponds to having the regime pushing the ES bits outside the representation), so in extreme cases the relative error can even be larger than 1.

1.2.3 Logarithmic Number System

All formats introduced until now are only able to represent rational values. This is not necessarily the case, as can be seen with Logarithmic Number System [10], [11] (LNS). LNS keeps the idea from floating-point of encoding a number as a value scaled by a power of a given radix β. However, the significand is always 1, and the exponent is allowed to be non-integral. In radix β LNS, the representation of a value v is thus a fixed-point number r, such that

$$v = \beta^r$$

With this restricted definition, it is not possible to represent negative values. A sign bit is prepended to the representation to allow a sign-magnitude encoding of negative values. This still does not allow the representation of 0, so a special encoding is required for it if having it in the representable value set is a necessity. For instance, the minimal exponent value can be dedicated to it.

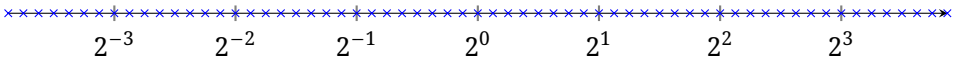


Figure 1.16: Representable positive values of an LNS system with representation of the exponent being a $\mathrm{sfix}(2,-3)$.

Figure 1.16 show the representable values for an LNS encoding scheme having the exponent defined as an $\mathrm{sfix}(2,-3)$. The values form a regular grid on a logarithmic scale. One example of representable irrational value for binary LNS is $\sqrt{2}$, represented as $2^{\frac{1}{2}}$. LNS has however the drawback of having almost no representable successive integers. For instance, binary LNS can represent 0, 1 and 2, but next representable integers are power of 2.

The rationale behind LNS is that it simplifies some elementary operations computations. Indeed, the representation of the product of two values is just the sum of their representations, and the power function evaluation is reduced to the computation of a product. However, this comes to the price of a complex addition.

Indeed, for v_0 and v_1 two values such that $v_0 > v_1$, represented in LNS by their logarithm l_0 and l_1, the computation of the representation of $v_s = v_0 + v_1$ is evaluated as follow:

$$\begin{aligned}
\log_2(v_s) &= \log_2(v_0 + v_1) \\
&= \log_2\left(2^{l_0} + 2^{l_1}\right) \\
&= \log_2\left(2^{l_1} \cdot \left(1 + 2^{l_0 - l_1}\right)\right) \quad = l_1 + s(l_0 - l_1)
\end{aligned}$$

with

$$s(x) = \log_2(1 + 2^x) \quad .$$

Contrary to fixed-point addition and product, no evaluation algorithm taking only the representation bits as input exists to evaluate s. Its evaluation requires dedicated function approximation operators, which increases the cost of computing it. The problem is similar for subtraction, and amplifies when trying to compute exact sum of products.

1.3 Hardware arithmetic operators

The motivation of developing arithmetic digital circuits is to be able to perform computation. For instance, one can be interested in computing the euclidean distance between two points (x_0, y_0) and (x_1, y_1). This consists in evaluating the function defined by

$$d : (x_0, y_0, x_1, y_1) \in \mathbb{R}^4 \mapsto \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

Having numerical formats allows to represent the function inputs using bit vector. The purpose of hardware arithmetic operators is to compute the result of a given function applied on their inputs. Such an operator is obviously parametrized by the function it approximates, f. In addition, the format of its inputs is given by its operand format list of, a n-tuple of numeric formats. The size of the operand list matches the arity of the objective function. In addition, the destination format df is the numerical format according to which the operator output is encoded.

As an example, an operator can be defined to approximate the distance function d, with all its inputs represented as 4-bits unsigned integers (corresponding to $\mathrm{ufix}(3, 0)$), and having its output encoded as an $\mathrm{sfix}(4, -2)$. This operator would return `00101.00` (5) when applied to (4,0,1,4). A rounding relation $o \subseteq \mathbb{D}_{\mathrm{df}} \times \mathbb{F}_{\mathrm{df}}$ specifying which values the operator is allowed to return when the result is not exactly representable is the last operator parameter. For instance, using the nearest neighboring as rounding relation for the example operator specifies that for the inputs (0, 0, 1, 1) it can only return 1.25 or 1.5, the representable values that surround the (not representable) exact value $\sqrt{2}$.

The formal semantics of the operator is that given an input tuple v, it returns the representation of a value $\tilde{r}(v)$ such that $(f(v), \tilde{r}(v)) \in o$.

For a given input value tuple v, the difference between the operator result and exact result is $\epsilon(v)$, the approximation error of the operator for this input.

$$\epsilon(v) = f(v) - \tilde{r}(v)$$

This work mostly focuses on the worst case absolute error of the operator

$$\epsilon = \max_{v} |\epsilon(v)|$$

Once an arithmetic operator is fully specified, it can be implemented on real hardware. Implementing an operator consists in finding a succession of elementary operations that will transform inputs representations to output representation. These elementary operations depend on the type of circuit chosen for the implementation. For instance on CPU it consists in what is available through the instruction set. This thesis is focused on arithmetic on FPGAs, the next section describes this kind of device.

1.4 Field-Programmable Gate Arrays

The process of developing integrated-circuits comports some very high fixed costs related to the manufacturing process. In order to avoid paying these costs for dysfunctional circuits, lots of efforts is made to ensure circuit design correctness before starting the manufacturing process. Functional correctness can be partially tested through simulation on CPUs. However, it can be very slow for big design depending on the granularity of the simulation. Besides, this does not always allow to test the circuit system integration. This is to allow the fast simulation of arbitrary IC designs that Field-Programmable Gate Arrays (FPGAs) have been originally developed. Nowadays, they are also used as computation accelerators or used as network stream filters.

1.4.1 FPGA architecture

At a coarse grain FPGAs consist (as their name suggests) in a collection of reconfigurable logic blocks. Configuring the FPGA “programs” the logic blocks of the device to make it emulate a specific circuit.

Look-Up Tables

At a finer grain, FPGAs can be seen as a grid of interconnected programmable boolean function evaluator. The logic element at the core of boolean function evaluation are Look-Up Tables (LUT). They consists in very small memory blocks. For instance, on Xilinx UltraScale architectures [12], LUTs are 2-bits memory addressed by a 5-bits word. These LUTs can be used to evaluate any

boolean function of five inputs, with at most two outputs. Extra logic at the LUT output enable to choose one of the two outputs based on a sixth input. This setting allows the LUT to compute any boolean function of 6 inputs.

The competing Intel Stratix 10 architecture [13] is based on 4 inputs LUTs that are grouped by 4 in what the architecture call an Adaptative Logic Module (ALM). In its widest configuration, an ALM can also be used as a 6-bits LUT of one output.

Table 1.5 show the (compressed) content of a 6-bits LUT that correspond to a 4:1 multiplexer. The output o of the function is the value of the input signal i_S, with S the selection signal.

Table 1.5: LUT content for evaluating a 4:1 multiplexer.

S		i_3	i_2	i_1	i_0	o
0	0	x	x	x	0	0
0	0	x	x	x	1	1
0	1	x	x	0	x	0
0	1	x	x	1	x	1
1	0	x	0	x	x	0
1	0	x	1	x	x	1
1	1	0	x	x	x	0
1	1	1	x	x	x	1

Each bit of a vector returned by a function can be seen as being the result of an independent function. As a result, in the general case, it requires n times more LUTs to compute a function returning a n-bits vector than a function returning a single bit.

Function of more input bits can be constructed by splitting their truth table in multiple LUTs, each one adressed by the same input bits. The LUT result to keep is selected using a multiplexer controlled by the remaining input bits. The multiplexing can be built using LUTs parametrized with the truth table of table 1.5, or using fabric built-in multiplexers.

Routing

The role of the routing infrastructure is to allow connecting resulting signals to LUT inputs, multiplexers or other built-in resources. Routing basically consists in a wire grid surrounding logic elements. Multiplexers programmed during FPGA configuration select the endpoints for each routing wire. That way, arbitrary connection pattern can be implemented.

Number of LUTs available in recent high-end FPGAs exceed 10^6. Having a dense connection graph between each LUT input and outputs is then im-

possible. Besides, physical distance between LUTs has also to be taken into account, as the time needed for a signal to propagate along a wire that cross the entire die is way more important than to go from one LUT to its immediate neighbor. That is why routing is hierarchical, with low latency and dense connection between neighbor LUTs, and slower and sparser connection between distant LUTs.

LUTs are grouped to form coarser blocks (called respectively Logic Array Block and Configurable Logic Block in stratix 10 and UltraScale architectures). These blocks contain resources to combine efficiently their LUT outputs in useful ways. For instance, an UltraScale logic CLB slice contains all the multiplexers required to group its 8 LUTs into 4 functions of 7 bits, 2 function of 8 bits or one function of 9 bits.

In the general case, connecting a LUT output to the input of another LUT requires passing through routing, which is at least one order of magnitude slower than the LUT output computation latency. This is quite detrimental to computations involving dependency chains. This issue arises for instance when building adders.

Specialized primitives

A natural way to implement an adder of two w-bits integers is to use w LUTs configured as (1-bit) full adders. The corresponding truth table is reported in table 1.6. The input bits are connected to i_0 and i_1. The output carry of the n-th full-adder is used as input for the $(n+1)$-th adder. The general scheme is reported on figure 1.17. With this setting, the carry propagation requires $w-1$ "routing" steps, which dominates the overall latency of the computation.

For this specific case, UltraScale and Stratix 10 both offer fast carry lines. These lines allow fast propagation of a carry to the neighbor LUT. Schematic for UltraScale architecture reported on figure 1.18 illustrates the functioning of fast carry chain. Each LUT is configured as an half-adder (performing the addition of two bits, which corresponds to the truth table of table 1.6 keeping only the lines where $c_{in} = 0$). In the UltraScale case, the o result goes to the LUT output $O6$ and the c_{out} output goes to $O5$. The carry propagation is done after LUT output. The fast c_{out} result of LUT A will be xor-ed with the o result of LUTB, computing the correct $O1$ value. And the o value from LUTB controls the MUXCY multiplexer to send the correct carry signal to COUT. If the LUTB o value is '0', then the input carry cannot propagate, so the LUTB c_{out} is sent to COUT. When the value is '1' then the LUTB c_{out} has necessary the value '0' and a carry will be propagated only if c_{in} value is '1', that is why in this case the c_{in} signal is propagated to COUT. As the neighbor LUTs are near to each other

Table 1.6: Full-adder truth table.

i_0	i_1	c_{in}	o	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

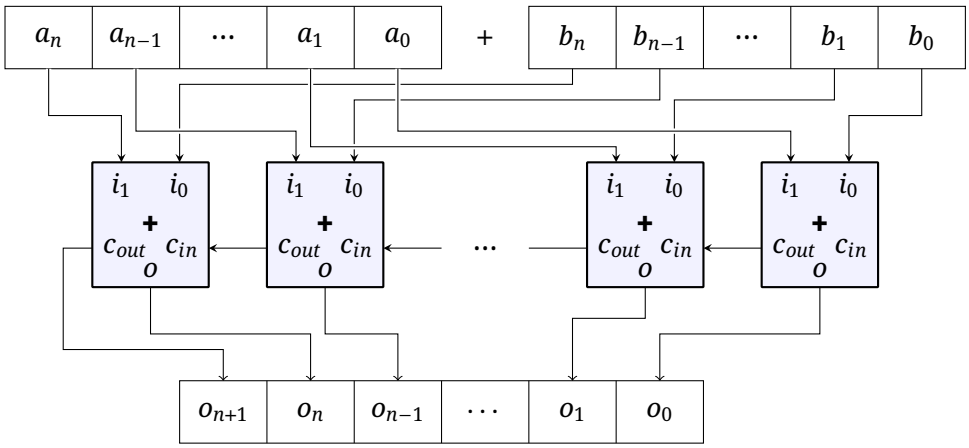


Figure 1.17: Wide adder implementation by Full Adder chaining.

and the logic is simpler than full routing logic, fast carry propagation is faster than routing (and on UltraScale devices at least, has even smaller latency than LUT read).

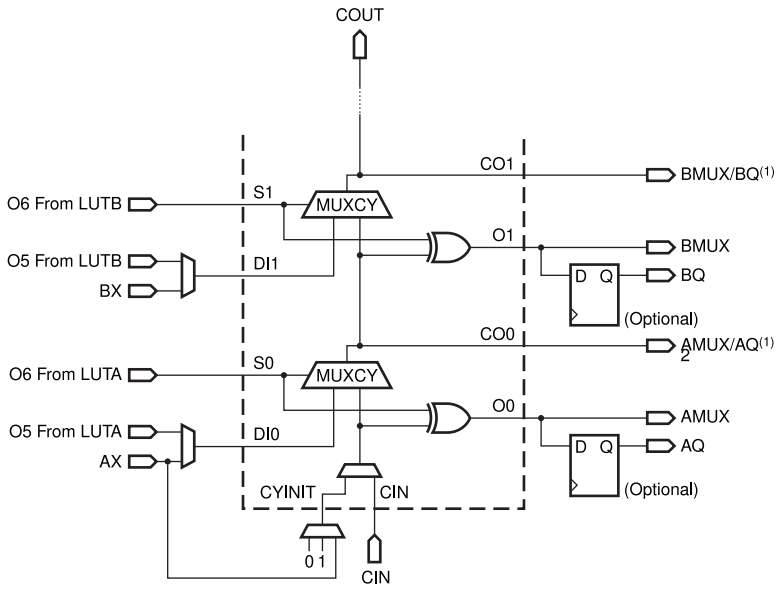


Figure 1.18: Part of the schematic of the UltraScale fast carry chain Source: [12].

Fast-carry is an example of primitive added in the FPGA fabric to allow fast implementation of frequent operations (wide addition in this case). DSPs are another example of specialized blocks that provides efficient implementation of basic operations. They provide for instance fast small product computation (27x18 bits products in the case of UltraScale DSPs [14], 27x27 for Stratix 10). Other provided operations includes pre-adding or bit-pattern comparison. Stratix 10 also offers hard operators for IEEE-754 binary32 floating point arithmetic.

In addition to computation primitives, memory primitives are also provided to avoid wasting routing to build large memories out of LUTs. Specialized LUT blocks to build distributed RAM or Block RAM are example of such primitives, with different size/latency trade-off.

1.4.2 Computing with FPGAs

While their original purpose was emulating arbitrary ASIC design, FPGAs are used nowadays as computation accelerators[15], [16]. They are well suited for processing data streams. The stream inputs can be read from FPGA interfaces or created out of a block of data retrieved from global memory. On CPU, the intermediate results of a computation are stored on general purpose

registers, and spilled back to memory if not enough registers are available. In opposition, FPGAs can be seen as computation graph processor. One FPGA configuration corresponding to one computation graph. As long as the input of a primitive (LUT or DSP for instance) does not change, the intermediate result computed by this primitive is preserved locally. By using the registers present at logic block outputs, it is possible to build arbitrary deep computation pipeline. The critical path between two registers determines the clock frequency at which the FPGA design can run. The latency to process one element is the product of the clock period and the pipeline depth. If the stream is dense enough to keep the pipeline full, one stream element can be processed at each clock cycle, ensuring good throughput.

In addition, to reduce the latency, spatial parallelism can be exploited. On a single computing graph, all independent computations can run in parallel. Besides, multiple computation graphs can also coexist on the same configuration. They can be efficiently synchronized in a dataflow fashion: FIFOs are inserted at the interface between two computation graphs, and a graph computation starts only when all its input queues have elements ready to be processed.

1.4.3 From computation graph to FPGA configuration

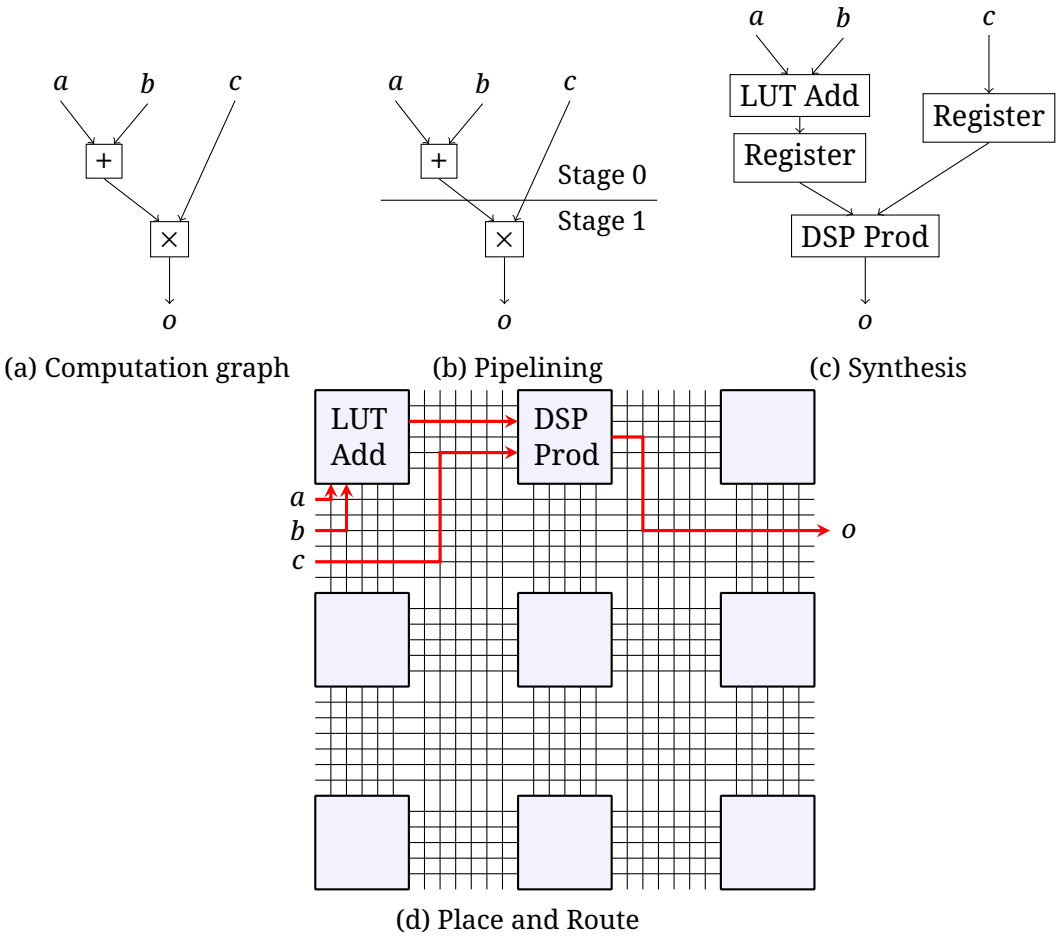
The process of getting an FPGA configuration from a computation graph involves multiple steps. The first step, pipelining, requires specifying how to split the computation graph in multiple pipeline stages. This split will determine which computation nodes should be registered. The result of this stage is a register transfer level description of the computation graph.

The second stage of this compilation process is synthesis, which performs logic optimization before mapping arbitrary computation to FPGA primitives (e.g. LUTs and DSPs). The resulting is a logical netlist describing each required FPGA primitive, the connection graph between them and their configuration.

The final stages, place and route, determine the physical location for each of the primitive of the synthesis netlist and the routing configuration required to get the specified connectivity. At the end of this stages a physical netlist fully specifying the FPGA configuration is produced. This netlist can be used to produce an FPGA bitstream: the binary file that contains the device configuration.

Resources and timing analysis are performed on the Synthesis and Routed netlists to ensure that the computation graph fits on the device and determine the clock frequency at which the configuration can run.

Figure 1.19 schematizes the different compilation stages for a simple com-



(a) Computation graph (b) Pipelining (c) Synthesis

(d) Place and Route

Figure 1.19: Schematic representation of the compilation stages from computation graph to logical FPGA primitive netlist.

putation graph computing $o = (a + b) \times c$. Choices made at an early stage have impact on the next stages. For instance, the register insertion point specified at the pipelining stage forces to $a + b$ as an intermediate result. This hinders the usage of the DSP pre-adder, which would have otherwise allowed to fuse all the computation in one DSP Primitive. The task of pipelining is then quite complicated, as it should be aware of how the computation will be mapped to hardware primitives.

1.4.4 Computation graph descriptions

Hardware Description Languages

On the traditional FPGA configuration workflow, the pipelining step is done manually. The pipelined computation graph is given to the tool chain in a Hardware Description Language (HDL) such as Verilog [17] or VHDL [18]. List-

ing 1.1 gives an example of a Verilog description of a circuit corresponding to the pipelined computation graph of figure 1.19b. A register is manually inserted to create a pipelined stage. As it can be seen on this example, traditional HDL description is at such a low-level that the programmer has to specify that registers are assigned on clock signal positive edges. Besides, the distinction between blocking and non-blocking assignment can be confusing. In addition, despite these languages being standardized, their support in commercial tools is incomplete, each toolchain supporting a different subset of it. Finally, traditional HDL are well suited to give a hardware description, but offer only very limited possibility to manipulate a description. For instance, it is not easy to get a generic ”protocol” behavior, that would specify a generic interface, and add this behavior easily to existing components. This limits drastically the possibility of developing standalone generic reusable hardware description in these languages. The reusability and genericity is then obtained by developing generators in other languages that produce HDL description on demand.

Listing 1.1: Verilog description of a circuit computing $(a + b) \cdot c$ for 12-bits inputs.

```
module PreAddProd(
    input         clock,
    input         reset,
    input  [11:0] A,
    input  [11:0] B,
    input  [11:0] C,
    output [24:0] result
);
    reg [12:0] preadd_res;
    assign io_result = preadd_res * C;
    always @(posedge clock) begin
        if (reset) begin
        preadd_res <= 13'h0;
        end else begin
        preadd_res <= A + B;
        end
    end
endmodule
```

Hardware Construction Languages

All these constraints hinder FPGAs adoption. To relax them, new generation of languages, hardware construction languages (HCL) have been developed. Usually, they consist in domain specific languages embedded in a general purpose host programming language. Developing hardware with these tools consists in writing a software application that builds a hardware description with the embedded language at runtime. Access to the host language allows the development of integrated component libraries, and higher level libraries that combine these components. Functions are provided to convert these representations to traditional HDLs. The most significant example of such tools is Chisel [19], that uses Scala as host language. Listing 1.2 gives a description of the same circuit using chisel. The `clock` and `reset` signals are completely hidden in this description. Inserting a register is done using a dedicated class (here a register with initial value), and the custom assignment operator (`:=` symbol) is overloaded in a way that makes sense for register (update on clock rising edge and reset on reset). It is possible to get a handle on the clock signal if required, but the goal is to have primitives with good default behavior that avoid having to manipulate it directly. Line 13 shows how to print the Verilog corresponding to a `PreAddProd`.

Listing 1.2: Chisel description of a circuit computing $(a+b)\cdot c$ for 12-bits inputs.

```
1  class PreAddProd extends Module {
2      val io = IO(new Bundle {
3          val A = Input(UInt(12.W))
4          val B = Input(UInt(12.W))
5          val C = Input(UInt(12.W))
6          val result = Output(UInt(25.W))
7      })
8
9      val preadd_res = RegInit(0.U(13.W))
10     preadd_res := io.A + io.B
11     io.result := preadd_res * io.C
12 }
13 println(getVerilog(new PreAddProd))
```

High-Level Synthesis

While improving the usability by reducing the amount of boilerplate to describe components, and offering genericity, these recent HCL stay at a very

low abstraction level. In order to broaden the audience even more, High Level Synthesis (HLS) tools have been developed. The aim of these tools is allowing programmers to give behavioral description of circuits as programs written in high level programming languages. While Modern HDL uses modern languages to generate circuit description, in HLS the program is the circuit description. Both AMD and Intel offer an HLS solution to program their FPGA, respectively Vitis HLS [20] and Quartus HLS. Non-vendor alternatives exist both in academia, such as Bambu [21] or in the industry, such as LegUp (which originated from an academic project but is now supported by Microchip).

Most of the tool use C++ as their source language. As with HDL, different subsets of the language are supported by the different tools. A first point of divergence is the target C++ standard version, which defines which syntactic constructs are supported. Apart from this, the main divergences is the level of support of the standard library constructs, especially regarding multi-thread and memory allocation related library functions. The work presented in this thesis mostly uses Vitis HLS, which officially supports C++14, without dynamic allocations and no standard library containers.

Listing 1.3: HLS C++ description of a circuit computing $(a + b) \cdot c$.

```
1 int preAddProd(int A, int B, int C) {
2   return (A + B) * C;
3 }
```

Listing 1.3 shows an example of a pure C++ HLS code describing the same computation as on the HDL/HCL examples. This code does not contain any pipeline related information. It is the role of the compiler to find a good pipelining for the implementation in consideration with the compilation objective (clock frequency, area or latency minimization).

Describing the computation graph that should be implemented on FPGAs directly as a language function allows its integration in more sophisticated libraries. One example of this is SYCL [22], a C++ library that allows to program heterogeneous devices in a single-source fashion. Listing 1.4 gives an example of a SYCL application. The code of the lambda parameter of the `submit` method call will be compiled for FPGA. Invocation of this FPGA code from the CPU application and data synchronization between host and device memory is taken care of by the SYCL runtime. While the buffer programming model presented here is not always suitable for efficient FPGAs implementation, Intel has proposed an extension for a pipe based alternative programming model that is more fit to FPGAs [23].

Listing 1.4: Simultaneous definition and call of an FPGA kernel computing $(a+b)\cdot c$.

```
int main() {
    // Declare buffer of one element
    sycl::buffer<int> abuff{{1}}, bbuff{{1}}, cbuff{{1}}, rbuff{{0}};
    // Write values inside the buffer on host (CPU) side
    {   abuff.get_access()[0] = 5;
        bbuff.get_access()[0] = 24;
        cbuff.get_access()[0] = 17;  }
    // Create a task queue that will run on FPGAs
    sycl::queue Q{sycl::fpga_selector{}};
    {
        Q.submit([&](sycl::handler cgh){
            // Will be executed on FPGA
            auto a = abuff.get_access(cgh)[0];
            auto b = bbuff.get_access(cgh)[0];
            auto c = cbuff.get_access(cgh)[0];
            rbuff.get_access(cgh)[0] = (a+b) * c;
        });
    }
    {   // Will print the result of (24+5) * 17
        std::cout << rbuff.get_access()[0] << "\n";
    }
    return EXIT_SUCCESS;
}
```

It is possible for the user to specify constraints on the implementation directly from the code, by using custom pragmas or attributes. For instance, the latency of the function, the pipeline initiation interval, global limit on primitive usage or specific mapping of intermediate computation to defined primitives is controllable by pragmas. Some tools also allow the manual insertion of register using a special library type or a built-in function. This is however not the case anymore in recent Vitis HLS versions.

While HCL targets hardware developers wanting an improved productivity over traditional HDL flows, HLS tools target software engineers. The approach is more iterative and holistic. Full algorithm describing the intent of the circuit is first written. After functional verification, tuning can be performed by using pragmas. A big advantage of using a compiled software language for describing the behavior of the circuit is the possibility of very fast functional verification (by compiling the circuit code as software and running it on CPUs, with access to regular software debugging tools). Besides, co-

simulation can be performed to check the correctness of the generated hardware.

1.5 High-Level Synthesis arithmetic support

C++ having initially been designed with CPUs as sole target, current standard version offers only basic integer type of power of 2 multiple of 8 bits up to 64 bits. While the introduction of the arbitrary width `BitInt` type of C23 [24] will probably find its counterpart in next C++ standard, current system provides their own library types to manipulate arbitrary width integer. For instance, the equivalent 12-bit circuit in Vitis HLS would be defined as in listing 1.5.

Careful choice of numerical formats for computation is a way to reduce area and latency by avoiding computing overprecise results. To this extent, HLS tools offer library types to use custom numeric formats, in addition to languages built-in types. The first class of these types are arbitrary width integers. The corresponding types are `ac_int` and `ap_int` for Quartus HLS and Vitis HLS respectively. These types are parametrized by a width and a boolean indicating whether they are signed. Signed integers negative values are represented in 2's complement. Parametrized fixed-point types are also provided.

Listing 1.5: HLS C++ description of a circuit computing $(a+b)\cdot c$ with 12-bits inputs.

```
#include <ap_int.h>

using uint12_t = ap_uint<12>;

ap_uint<25> preAddProd(uint12_t A, uint12_t B, uint12_t C) {
  return (A + B) * C;
}
```

For all these types, only basic arithmetic operations (addition, subtraction, product, division and modulo) are supported. In particular there is no support for usual transcendental functions evaluation. To get this functionality, either the user write its own arithmetic operator (which is a bit tedious if required for each computation and breaks completely the accessibility purpose of HLS) or interface with an externally provided operator.

1.5.1 HDL arithmetic core generators

Externally provided operators can be provided in the form of HDL description. Most of the time, these descriptions are produced by arithmetic cores generators. These tools take a description of the operator they should produce (either in text form or by providing a GUI), the hardware target for which to produce it, and output an HDL description of an implementation of the operator optimized for this target. An example of such generator is FloPoCo [25]. Listing 1.6 gives an example of the command line to produce a Fused Multiply Add operator (computing $o = a \cdot b + c$ with a single rounding) using FloPoCo.

```
flopoco target=kintex7 IEEEFMA WE=8 WF=23
```

Listing 1.6: Flopoco command line to produce an IEEE-754 binary32 Fused Multiply Add operator for Kintex 7 architecture.

HDL files produced by a generator can then be integrated into an HLS program. The way to implement this integration is not standardized and depends on the toolchain. In general, the integration of an exogenous HDL IP inside an HLS program impacts the synthesis QoR. The external IP is already pipelined and can require that its inputs are registered. This has the effect that the IP is seen as a logic and pipelining optimization barrier for the HLS compiler, potentially increasing the latency and surface compared to an integrated HLS IP.

1.5.2 Toward on-demand HLS arithmetic operator implementation ?

To summarize, it is possible to program FPGAs at different level of abstraction.

- *HDL* allow to have fine-grain control over how things are implemented, but they are very verbose even for frequent operations. Their specification supports only very limited genericity.
- *HCL* provide a neat interface to generate HDL, allowing better expressivity and less verbosity while allowing almost the same fine-grain control than HLS, and offering more genericity by providing access to a general purpose programming language.
- *HLS* provides construct of a higher level of abstraction, enhancing the productivity of component writing. Describing a component as a host language program allows integration in a broader software environment, however in the current state HLS lacks some possibility of fine-grain tuning of im-

plementation. One of such limitations is the lack of support of application-specific arithmetic primitives.

Can we have HLS with custom arithmetic operator support and fine grain control over their implementation ? This is the question to which this thesis aims at giving a (partial) answer. Chapter 2 will present a first approach to solve this issue with pure C++ library code. To support more complex use-cases, chapter 3 will present alternative solutions that require compiler support.

2

A portable HLS-enabled library for custom numerical formats

This chapter introduces MArTo [26], a proof-of-concept HLS library for basic operation support on custom numeric formats. This library implements product, subtraction and division operation for arbitrary fixed-point, IEEE-754 and posit formats. The library is written in pure C++, in order to be portable on various HLS toolchains.

The support of the elementary operations is based on arbitrary width integers arithmetic. Main HLS tools provide specific libraries for using this arithmetic. In order to be portable across toolchains, MArTo operators are built using a zero-cost abstraction layer for these libraries. This layer comes into the form of another library: hint [27]. It is introduced in section 2.1.

2.1 Hint, a portable abstraction layer for arbitrary width integer arithmetic

Arbitrary-sized integers are extremely useful when designing custom operators: for instance, IEEE-754 operators have 11-bit exponents and 52-bit fractions at the inputs and outputs. The adder data-path comports a 53-bit explicit fraction and a 56-bit data after effective addition. For multipliers, the intermediate mantissa product has a width of 106 bits. Being able to manipulates such integers is then an important feature for writing HLS-enabled arithmetic operators.

2.1.1 Integers and HLS

The support of arbitrary-sized bit vectors is not standardized among HLS tools. The nearest to a common standard is the `ac_int` templatised C++ library

developed at Mentor Graphics [28]. It is supported by the industrial tools Intel HLS and CatapultC, and the academic tool GAUT. However, the HLS tool with the most traction in reconfigurable computing is probably Xilinx Vitis HLS, and it uses a proprietary library called `ap_int`.

While `ap_int` and `ac_int` provide almost functionally equivalent support for basic arbitrary-sized signed and unsigned integers, their interfaces differ, and they do not have equivalent support for operations such as leading zero count.

Other tools only support widths up to 64-bits: the academic tool Augh [29] defines 64 new non-standard base types `int1` to `int64`. Two other tools, LegUP and Panda/Bambu only support the standard C integer types so code must be written with only 8-, 16-, 32-, and 64-bit integers. If only standard-width types are available, a 17-bit integer must be emulated in the code using 32-bits numbers and bit-masks. These masked operations will hopefully be transformed to 17-bits integer operations during some optimization step. However, if this happened late in the process, the HLS schedule can be produced considering 32-bits operations, giving a pessimistic scheduling. Besides, this adds a lot of clutter in user code, and reduce its readability.

The necessity for the support of arbitrary width integer arithmetic slowly percolates in C/C++ standard, as it can be seen with the recent inclusion of the `BitInt(n)` types in C23. However, the semantics for these new types is underspecified. For instance, the result of overflowing operations on signed integer is “undefined behavior” in C/C++. This means that each implementation is free to return what it wants when such case arises.

Strong semantics

C++ is strongly typed, which means that each value has a type. Being able to handle bit-precise integer types is a great way to control the bus/register size of the corresponding hardware. While it is straightforward to control the type of declared variables, it is more complicated to define the exact meaning of compound expressions involving implicit intermediate types, and implicit type conversions.

For instance, the type of intermediate result `(a+b)` in the expression `(a+b)+c` is well specified in C/C++, but this specification is sometime non-intuitive and error-prone. If both `a` and `b` are unsigned 8-bit integers (declared with type `uint8_t`), the intermediate result `(a+b)` is implicitly promoted to the `int` type, that is implementation-defined but should have a width of at least 16 bits. If not optimized out by the compiler, this implicit conversion entails a waste of at least 7 bits. These rules have been established

to limit the risk of accidental overflow, but they do not actually catch all cases. An example is given in listing 2.1 with operands having types of different signedness. In this example, C/C++ conversion rules specifies that `b` should be converted to `uint32_t` before performing the computation. That is why no sign extension is performed when the result is affected to a wider type.

```
#include <iostream>
#include <cstdint>

int main() {
  uint32_t a = 1;
  int32_t b = -1;
  int64_t res = a*b;
  // print 4294967295 (= 2^(32) - 1)
  std::cout << res <<'\n';
  return 0;
}
```

Listing 2.1: Example of a counter-intuitive C intermediate result type.

Mainstream tools such as VivadoHLS or CatapultC tend to choose the implicit intermediate types in a way that ensures that no information is lost. For instance, if both `a` and `b` are 32-bit integers, the implicit type of `(a+b)` should be a 33-bit integer to hold the possible carry out, unless the result of `(a+b)+c` is itself finally stored in a 32-bit integer, in which case all the arithmetic may happen modulo 2^{32}. But this should nevertheless be specified.

Things are a bit trickier with shifts: left shifts may or may not lose the shifted out bits. Right shifts always loose the shifted-out bits, but in the signed case they may perform a sign extension, where the size of the intermediate format will matter. An example is the program from listing 2.2.

With gcc 11.2 for x86_64 architecture, execution of the code from listing 2.2 prints 255 when compiled with the `-O0` flag and 0 when compiled with the `-O2` flag. This can be explained by different intermediate types for the intermediate result (a 64-bit int in the -O0 case, a 32-bit type in the -02 case). This actually conforms to the C++ standard, according which shifting a value of a number of bits bigger than the format width is undefined behavior. Such corner cases (C++ standard is full of undefined behavior) advocate for a library with consistent and completely specified semantics, in order to avoid users that are not aware of them to fall in this kind of trap.

```cpp
#include <iostream>
#include <cstdint>
int main() {
  int32_t a,b,s;
  a = 255;
  s = 33;
  b = (a<<s) >>s;
  std::cout << b << "\n";
}
```

Listing 2.2: Undefined behavior on shifts.

2.1.2 Core arithmetic primitives for floating-point operators

Most floating-point operators (be it IEEE-754, posit-like, or other) rely on the following basic components:

- **Arbitrary-precision addition, subtraction, multiplication**. Multiplication can be implemented out of addition, but on reconfigurable targets it can also be built by assembling DSP blocks in a clever way. Therefore, multiplication should be a primitive, and its implementation is best left to back-end tools that know the target. Division and square root can be implemented either out of addition and tabulation, or out of multiplications. Whether or not this algorithmic choice should be left to the back-end tools is out of the scope of this thesis.
- **Arbitrary precision shifters**. There are standard operators in C/C++ for shift operations: << and >>. As we have already observed, it doesn't mean that their behavior is completely defined by the standard. In a processor, we usually have shift instructions that input the shift value and an integer, and output an integer of the same size (with possible loss of information). The C shift operators expose these instructions. Now if we are generating hardware, it is interesting to generalize as depicted in Figure 2.1: a shifter may be defined by an input width w_i, a maximum shift distance d, and an output width w_o. The shift input will be an integer on $\lceil \log_2 d \rceil$ bits. The shifter can be errorless (no shifted-out bits) if $w_o \geq w_i + d$. Besides, the "fill" bit of the shifter can also be specified (for instance to use the sign bit on a right shifter). Figure 2.9 gives an *in-situ* illustration of shifter usage.
- **Arbitrary precision leading-bit counters**. These operators count the number of successive leftmost bits sharing the same value. Leading-zero counters (LZC) and Leading-One counters (LOC) return a non-zero count only if the bits value is what they count (respectively '0' and '1'). For instance, LZC

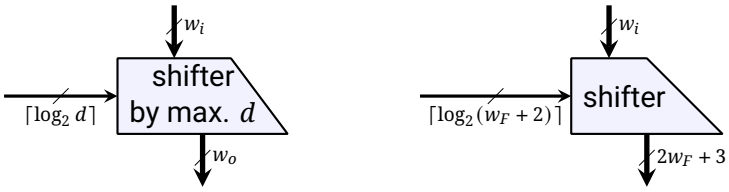


Figure 2.1: A generic shifter (left) instantiated (right) in a floating-point adder with w_F fraction bits

of the signal `0001010` is 3, while LOC of the same value is 0. Leading-Zero or One counter (LZOC) as an extra input bit to choose what to count at runtime. This allows for instance to build leading-bit counters, that behaves like a LZC if the leftmost bit is 0, else like a LOC. The two kind of counters are presented on figure 2.2.

(a) Leading-zero counter (b) Leading zero or one counter

Figure 2.2: Generic leading-bit counters.

To reduce the data-path width and improve the delay, it is often useful to merge these operators:

- **Addition with carry-in** is in principle no more expensive than plain addition. Writing $a+b+1$, or $a+b+c$ where one of the three variables is a single bit, should translate to a single adder.
- A **shifter-sticky** is a shifter whose output size is the same as the input size. Therefore, bits may be shifted out. The logical OR of shifted-out bits, historically called sticky-bit, is computed in parallel with the shift result. When computing a floating-point sum, fraction alignment is performed, by right-shifting fraction of the operand with lower exponent. The sticky bit keeps trace of those shifted-out bits to allow correct rounding. Compared to the naive alternative of computing a shift on a right-padded input before computing an OR-reduction on the output, performing the steps in parallel allows to reduce the operator latency and avoid the cost of the MUXes to shift bits that have no other aim than to be OR-reduced.

  An *in-situ* example of shifter-sticky usage is visible on figure 2.7, page 68.

- A normalizer is a **combined leading-zero counter and shifter**. It is a leading-zero counter that, at the same time, shifts the input so that the leading bit of the output is the first non-zero bit. It outputs the normalized result, along with the number of zero bits that were counted. Posits make use of a similar combined leading-bit-counter and shifter, as illustrated on figure 2.5, page 65.

All the tools support C++ primitives (addition, multiplication, shifts), although the actual behavior does not necessarily follow C++ semantics. Sometimes, they are implemented as highly optimized IP cores. Sometimes, they are implemented as libraries.

The most notable missing basic operation is the leading bit count. Among the fused operations, only the add with carry-in is sometimes supported.

2.1.3 Type safety for arbitrary-precision integers in HLS

Current HLS integer libraries perform very little compile-time sanity checks. This section describes the Hint library variable declaration and its elementary methods along with their semantics and checks. These basic methods are used to build more complex operators. Code from listing 2.3 is an illustration of code using the `hint` library.

```
1  template<template<unsigned int, bool> class Wrapper>
2  // Takes a 12-bit vector as input,
3  // Consider it as the concatenation of two unsigned integer
4  // Compute the LZC of their sum
5  Wrapper<3, false> computation(Wrapper<12, false> input) {
6    auto high = input.template slice<11, 6>();
7    auto low = input.template slice<5, 0>();
8    auto sum = high + low; // 7 bits, unsigned
9    return lzc(sum); // 3 bits, unsigned
10 }
```

Listing 2.3: Example of `hint` code computing the leading-zero count of the sum of two 6-bits integers provided concatenated in a 12-bits vector.

Variable declaration

The Hint library defines templated wrappers around underlying backends. A wrapper is a template class parametrized by an unsigned int describing the width of the represented integer type and a boolean defining whether it

is signed. The library offers wrappers for the main arbitrary precision integer library: `ap_int` and `ac_int` for HLS, and GMP for software libraries. These wrapper template classes can be used as a template template parameter to build backend agnostic functions. The templated `computation` function of listing 2.3 illustrates this. It can for instance be instantiated with parameter of type `VivadoWrapper<12, false>` (a wrapper around `ap_uint<12>`), or `IntelWrapper<12, false>` (that wraps an `ac_int<12, false>`). Using C++20 concepts could simplify the syntax, but at the moment of writing this version of the standard is not supported by HLS toolchains.

Arithmetic operation result type and value assignment

The resulting type of any arithmetic operation is always *the smallest type that ensures that no overflows can occur*. For instance, the addition of two 6-bit integers at line 8 of listing 2.3 results in an 7 bit integer. This is similar to the choice made by industrial toolchains libraries. In these libraries however, assignment to a variable of mismatched size is a cause of silent truncation or padding. This can result in counter-intuitive behavior. For instance, Vitis HLS gives access to a built-in method `__builtin_clz` that takes an `int` as parameter and returns its leading-zero count. This function can be used on an `ap_uint<24>` without any error signaled by the compiler. However, as it is only defined for an int, the value will be silently extended to a 32-bit integer and the resulting leading-zero count will be too high of 8 compared to the expected result.

The hint library diverges from this approach by only allowing assignment of matching expression types. Without explicit truncation or padding, a compilation error fires when attempting to assign non-matching expression to a variable. In listing 2.3 for instance, if the return type of the function has been declared as `Wrapper<5, false>`, the return statement of line 9 would have been invalid, as the LZC on a 7-bit vector is a 3-bit unsigned value.

When porting to `hint` some arithmetical operators initially developed with `ap_int`, this strict semantics allowed the discovery of some place where internal computations were silent padding occured. While this issues did not affect the operator functionality, resizing correctly the relevant intermediate signals allowed to get a better default pipelining.

Slicing

Slicing consists in extracting a contiguous sub-vector from a bit vector. For instance, the slice `1101` can be extracted from the bits (3 to 6) of the vector

`01101101`. This operation is pervasive in arithmetic operators. It is for instance used to separate the different fields from an IEEE-754 representation.

Slicing methods for `ac_int` and `ap_int` have different restrictions, and may therefore not be interchangeable. In `ac_int`, a bit slice of size `S` starting as bit weight `lsb` of the variable `var` is written `var.slc<S>(lsb)`. Conversely, the `ap_int` slicing method is `var.range(msb,lsb)` where `msb` and `lsb` are the weights of the MSB (most significant bit) and LSB (least significant bit) from which to slice `var`.

The difference is that the value of `S` must be known at compile time (more specifically, it should be a `constexpr` value). Therefore, a slice whose size varies in a loop will compile using `ap_int` but not using `ac_int`. However, the synthesized hardware bus size is fixed for the whole loop block (it makes no sense to "add dynamically" wires during the loop iterations[1]). So the choice made by `ac_int` seems more judicious as it allows compile-time check that the size of the slice is not bigger than the input value.

The approach followed in hint is even more restrictive, as the position of both limits of the slice needs to be `constexpr` values, in order to allow bound checks. The syntax is `var.slice<msb,lsb>()` where `msb` and `lsb` are the positions of the MSB and LSB of the slice. The usage of this method is visible on line 6 and 7 of listing 2.3.

2.1.4 Others operations

The Hint API can be extended with any other methods with the same spirit that all types must be checked at compile time. The provided operations are:

- concatenation, which join the bits from two variable, resulting in an integer having for width the sum of the two operand widths.
- bitwise operations such as and, or, xor that from two identical W width variables returns a W bits variable containing the corresponding bitwise operation.
- and/or reductions that returns a single bit result,
- a signal inverter,
- an operator that reverse the bit order of the variable (the MSB of the input becomes the LSB of the output for instance),
- a padding operator,

[1]It can make sense if the iteration space is known at compile time, and the slice indices depends only on it. In this case, the loop can be spatially unrolled with each iteration instance having bus of the required size. The loop unrolling can be required by a specific pragma in Vitis HLS. However, the intent for this kind of constructs is more clearly expressed with a proper template unrolling.

- a bit replicator,
- a comparison operators that only compares identical width and sign Hint variables,
- a multiplexer operator that takes a control bit and two identical width variables,
- explicit modular arithmetic operations,
- add with carry operation.

A concrete example of `hint` library usage is presented in listing A.2, which present the code of an IEEE-754 adder.

2.1.5 Software design of backend common interface

There are several ways to implement a common interface to the many backend arbitrary width integer libraries. Polymorphism through virtual inheritance is not an option as the library aims at adding no overhead compared to native library usage, so static polymorphism should be used. This section presents two possible approaches and their limitations.

Shared implicit interface

A first approach is to have independent classes for each backend. This class should implement the same shared interface. While being the more direct way to implement the library, this approach has two major drawbacks. The first one is that the static checks have to be duplicated in each backend class, thus being more prone to semantic divergence. The second drawback is that the common interface is not enforced by the compiler.

Curiously recurring template pattern

An alternative solution that solves these issue is by using the Curiously Recurring Template Pattern [30] C++ idiom. This idiom enables static polymorphism by providing a template class that defines the public library interface as non-virtual methods. Backend classes inherit from the base class specialized for themselves. Interface (non-virtual) method delegates the computation to (also non virtual) derived class private implementation methods via casting `this` to a pointer on derived class. This allows the enforcement of a shared internal interface among backends (the base CRTP class cannot be specialized for a given backend if it does not provide the internal interface methods). In addition, the static checks can be factorized in the CRTP public interface code.

This method uses no runtime polymorphism, so dispatching to the right backend method is done at compile time without the need of virtual tables.

This approach worked correctly in software simulation for both Vitis HLS and Intel HLS. However, neither of the tools managed to pipeline correctly the code produced by this approach. Investigation tends to point the responsibility of an incomplete code inlining by the HLS compilers. Due to this issue, the first approach as been chosen for hint.

Cleaner interface with concepts and functional library

Two opportunities to improve `hint` interface have been identified. First, pure functions wrapping hint objects template methods. An example of such a wrapper is given in listing 2.4 for the slice method. This could improve greatly the readability of `hint` code by removing the need of prefixing template method usage with the `template` keyword in generic operator code.

```
1 template<unsigned int MSB,
2          unsigned int LSB,
3          unsigned int size,
4          bool isSigned,
5          template<unsigned int, bool> class Wrapper>
6 auto slice(Wrapper<size, isSigned> input) {
7   return input.template slice<MSB, LSB>();
8 }
```

Listing 2.4: Pure function wrapping call to the `slice` template method.

The second improvement comes from the concept mechanism introduced in C++20. Using this mechanism would allow expressing concisely the type requirement, and dropping the non-intuitive template template parameter. For instance the `slice` functional wrapper of listing 2.4 could be rewritten in the form presented in listing 2.5.

```
1 template<unsigned int MSB,
2          unsigned int LSB>
3 auto slice(HintWrapper input) {
4   return input.template slice<MSB, LSB>();
5 }
```

Listing 2.5: Pure function wrapping call to the `slice` template method with cleaner interface using C++20 concepts.

With these two improvement, the toy operator of listing 2.3 could be expressed as in listing 2.6.

```
auto computation(UnsignedHintWrapper<12> input) {
  auto high = slice<11, 6>(input);
  auto low = slice<5, 0>(input);
  auto sum = high + low;
  return lzc(sum);
}
```

Listing 2.6: Rewriting of example from listing 2.3 with proposed interface improvement.

Concepts have not been used due to the lack of support of C++20 standard by the targeted HLS toolchains. Using functional wrappers to improve the readability could have been done earlier, but the idea has only been identified recently, which explains why it is not implemented.

2.1.6 Evaluation

Usage of `hint` in real applications (floating-point operators) will be presented in section 2.2. Current section evaluates the individual building blocks provided by the library.

The evaluation is divided in two parts. The first consists in ensuring that the library does not add overhead compared to native library usage. The second part evaluates combined operators, ensuring that they indeed reduce resources consumption and/or latency compared to their sequential counterparts.

All the results presented section are given after place-and-route. At the time of evaluation, Vivado HLS (the ancestor of Vitis HLS) 2018.1, the latest available version, was quite unstable, with numerous crashes and silent production of incorrect hardware at rare occasions. That is why the more stable Vivado HLS 2016.3 version was used at this time. Most of the results presented in this section are obtained with this version. Experiments based on this library and using more recent tool version are presented in section 2.3.1. These experiment results seem to indicate that conclusions obtained with this old tool version still hold with recent tool. Nevertheless, in order to get recent numbers, the experiment on LZC comparison for Xilinx toolchain has been re-run using Vitis HLS 2022.2, the most recent version at time of writing. The FPGA architecture targeted for experiments on Xilinx HLS toolchain is Kintex 7. For the Intel toolchain, IntelHLS 19.1 is used to target Arria 10.

Table 2.1: Synthesis of LZC Arria 10 (achieved clock target of 240MHz)

	N	ALMs	FFs	MLABs	cycles
`ac_int` LZC	26	32.5	32	0	1
	55	86.5	91	1	5
	256	465.5	710	1	8
`hint` LZC	26	32.5	32	0	1
	55	86.5	91	1	5
	256	465.5	710	1	8

Table 2.2: Synthesis of LZC and shifters on Kintex 7 (achieved target delay of 3ns). The number are obtained with Vitis 2022.2.

Implementation	N	LUTs	FFs	SRLs	cycles
`ap_int` LZC	26	26	33	1	2
	55	71	105	1	4
	256	278	546	3	9
`hint` LZC	26	26	33	1	2
	55	71	105	1	4
	256	277	546	3	9

Overhead evaluation

The overhead evaluation is performed on the implementation of a leading zero counter (LZC). The lzc algorithm chosen in this work has been implemented using `ac_int`, `ap_int` and Hint. The synthesis results are given in Tables 2.1 and 2.2 for Intel and Xilinx respectively. The width (N) of the inputs corresponds to real world examples. Indeed, 26 and 55 bits are the widths of the LZC needed in single and double precision floating-point adders, while a 256-bits LZC is needed when dealing with posit32 exact product accumulators.

The comparison between the native type implementation and the hint type implementation shows no overhead when using Hint.

Combined operators

A combined shifter + sticky has been implemented. This operator computes a right shift, and a 1-bit signal, the sticky bit, indicating if any of the shifted-out

Table 2.3: Synthesis of shifters+sticky on Arria 10 (achieved clock target of 240MHz)

Implementation	N	ALMs	FFs	MLABs
`ac_int` (shifter then sticky)	27	134	63	2
	56	277	212	3
`hint`	27	82	40	0
	56	179.5	128	0

Table 2.4: Synthesis of shifters+sticky on Kintex 7 (achieved target of 3ns)

Implementation	N	LUTs	FFs	Cycles
`ap_int` (shifter then sticky)	27	113	110	3
	56	309	234	4
`hint`	27	84	65	3
	56	203	133	3

bits had the value 0. It is useful when computing the addition of two floating point numbers. The shifter is used to align the fraction, and the sticky bit is required to compute correctly some rounding modes.

While a naive operation would first right-pad the input, shift the padded input and finally OR-reduce the slice that corresponds to shifted out bits. This doubles the size of the vector to shift, and is then quite costly. Instead, the or-reduction can be performed inline at each shifter stage.

The synthesis results of the shifter+sticky are presented in Tables 2.3 and 2.4. The width (N) of the operators comes once again from what is required in an IEEE-754 adder for binary32 (27 bits) and binary64 (56 bits). As both tables show, this optimization saves a considerable amount of logic on both targets. Table 2.3 does not report the number of cycles required for said operators. Indeed, IntelHLS was giving untrustworthy latency results. However, the circuits were co-simulated to ensure that they produced the correct mathematical results using ModelSim.

A second useful primitive is the normalizer, that combines a LZC and a left shifter. The semantic of this operator is to shift the input by its leading zero count, in order to bring its first ‘1’ bit to the leftmost position. This is for instance useful to normalize the result of IEEE-754 products involving one subnormal number. This can be done naively by first computing the LZC,

Table 2.5: Synthesis of normalizers Arria 10 (achieved clock target of 240MHz)

Implementation	N	ALMs	FFs	MLABs	cycles
`hint` LZC + native shift	28	93	106	0	2
	57	218.5	213	0	2
	64	296.5	340	0	3
	256	1487	1238	13	7
	279	1603	1322	14	7
`hint` fused Shift and LZC (normalizer)	28	88.5	72	0	1
	57	212	209	0	2
	64	279.5	308	0	3
	256	1388	1960	0	6
	279	1455.5	1544	0	4

Table 2.6: Synthesis of normalizers on Kintex 7 (achieved target delay of 3ns)

Implementation	N	LUTs	FFs	SRLs	cycles
`hint` LZC + native shift	28	96	81	0	8
	57	222	144	0	9
	64	264	142	0	8
	256	1532	1045	0	11
	279	1691	1131	0	12
`hint` fused shift + LZC (normalizer)	28	102	122	0	4
	57	254	297	0	5
	64	292	275	0	6
	256	1164	1568	0	8
	279	1958	2265	0	10

and then shifting by this amount. However, the LZC implementation is not far from computing successive shifts with masking, so it is possible to build an operator that performs the two operations simultaneously to reduce the computation latency compared to the naive implementation.

To evaluate such an operator, a combined hint lzc+shift is compared to a hint LZC followed by a native shift (>>). Tables 2.5 and 2.6 provide synthesis results of these implementations for both Intel and Xilinx FPGAs. In addition to sizes previously presented for the LZC.

For both Intel and Xilinx, the latency of the normalizer is improved compared to a serial LZC followed by shifter implementation. In some cases, the tools are even able to reduce the area of the operator.

2.2 Custom floating-point format library

The hint portability layer allowed to build MArTo [26], a portable HLS library for custom-size IEEE-754 and posit arithmetic types support. MArTo allows the programmer to define custom types by specializing the template classes `IEEENumber` and `PositNumber` that are parametrized by the format dimension (W_E and W_F for IEEE-754 formats, W and w_{ES} for posit formats) and the `hint` backend used for the underlying integer operations. Once a type is defined, it can be used like a native numerical type.

2.2.1 Elementary operation support

On its current state, MArTo supports for basic operation is limited to addition, subtraction and product. The five IEEE-754 rouding modes are supported for `IEEENumber` operations. An example of code using the library is shown on listing 2.7. It is easy to replace the numerical format used in this code by changing the definition of `my_fp_type`.

```
1 using my_fp_type = IEEENumber<9, 14, hint::VivadoWrapper>;
2 auto myFunction(my_fp_type a, my_fp_type b, my_fp_type c) {
3     return (a*b) + c;
4 }
```

Listing 2.7: MArTo code example.

For `IEEENumber`, the default rounding mode when using overloaded +, - and * is round to nearest, ties to even. Example from listing 2.8 shows how to use a non-default rounding mode. In this example, the input type can only be an

instance of `IEEENumber`, as `PositNumber` does not support alternative rounding modes.

```
using my_ieee_type = IEEENumber<9, 14, hint::VivadoWrapper>;
//computes o↓(oN(a * b) + c)
auto myFunction(my_ieee_type a, my_ieee_type b, my_ieee_type c) {
    auto prod = a * b;// Default rounding to-nearest ties to even
    return ieee_add(prod, c, IEEERounding::roundDown);
}
```

Listing 2.8: MArTo code example.

In addition to elementary operations, MArTo supports exact accumulation of products in wide fixed-point accumulators. This operation is detailed in the next section.

2.2.2 Exact fixed-point accumulation of floating-points products

Sum of product (or vector dot-product) is a pervasive operation in scientific computation. This is for instance the operation at the core of matrix product. This sum of product can be computed by using elementary addition and product. However, a rounding error is made at each elementary operations, and those errors accumulate. When the number of components of the vector is high, the resulting error can be important. This is even worse if this sum of product is part of the loop of an iterative algorithm.

An obvious solution to reduce the overall error in this case is to reduce the elementary rounding error by using floating-point formats with higher precision. However, this increases the memory required to store the elements, and the cost of elementary operation.

An alternative solution is to offer primitives that allows to compute such sum of product with only one final rounding. It is possible to provide all-in-one sum-of-product accumulation operations for fixed number of inputs, such as the Fused Multiply Add (FMA) of IEEE-754 that computes

$$\mathrm{FMA}(a, b, c) = a \cdot b + c$$

with only one rounding.

For arbitrary number of input, a solution initially proposed by Kulisch [31] is to have fixed-point number wide enough to store exactly any floating-point product of the source format playing the role of accumulator. Each product to sum is computed without rounding (resulting in a floating point number with a fraction width twice bigger than original format, and exponent field

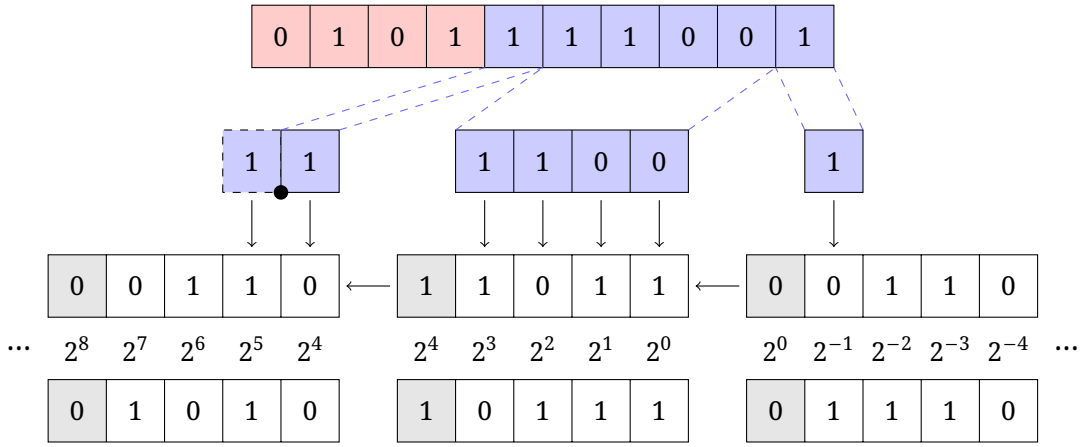


Figure 2.3: Example of product accumulation on the carry-save banks of a Kulisch-like accumulator.

having one bit more). The resulting fraction is added in the accumulator at the corresponding position (determined by the product exponent). When all products have been added, the accumulator stores the exact sum of the input products. The final operation is to round it back to the original format.

In a basic implementation where the accumulator is stored as one bit vector, addition of a product is an expensive operation due to the delay required for carry propagation from the lowest to the highest bit. In order to reduce this latency, the accumulator can be split in multiple banks in carry-save fashion. With this setting, there is an overlap bit between two adjacent banks. When a product is added to the accumulator, the carry bit from the lower bank is also added, and the resulting carry bit is stored in current bank carry-out bit waiting next summation to propagate. Figure 2.3 gives an example of one accumulation step. On this figure, the carry saved for each bank is highlighted with a gray background. At each accumulation step, bank-wise additions can be performed in parallel, as the carry bit is stored locally. In this setting, the cost of complete carry propagation is only paid when rounding the accumulator back to the floating-point format.

Posit standard requires the support of accumulator operations. In the posit world, these accumulators are called quires. The 2019 version of the IEEE-754 standard recommend support for the `dot` operation, which computes an approximation of the dot product of two floating-point vectors. However, it leaves the accuracy implementation defined.

MArTo defines several types to represent these wide accumulators in plain integer or carry save formats, overload the operator to allow convenient syntax for accumulation of custom type products, and provides the rounding operator to convert these accumulators to the elementary format they accumu-

late products of.

2.2.3 Operator implementations in MArTo

Implementation of IEEE-754 elementary operations

MArTo IEEE-754 elementary operation implementations are quite standard, so they will not be described in great details. The adder has a single-path architecture (which had better QoR on tested targets than dual path implementation at the time of development). The only ”subtlety” is the use of fused operators. For instance, the adder uses a combined shifter + sticky operator for fraction alignment. Architectural choices have been inspired by the corresponding floating-point architectures produced by FloPoCo [25]. The interested reader is invited to check [32] to find detailed discussion on possible IEEE-754 operator implementation. To get an idea of the overall design, the generic adder code is reported in listing A.2. The fused shifter and or reduction is visible on line 95 of this listing. This code is also interesting to look at in order to see the relative amount of work of the different computation “steps”. Indeed, while they occupy an important number of line of code, special case handling and rounding logic actually consists in a few or/and reduction and boolean combination of these reduction results. These operations are not very expensive in regard to the large shifts required to align the fraction and renormalize the result when required.

On the other hand, there is less literature about Posit operator. That is the reason why their MArTo implementation is further detailed in the following sections.

Posit smallest floating-point superset

The general idea behind posit operator implementation is to convert the posit encoding to a fixed-width fields floating-point format, and perform the operation on this representation.

The sizes of posit value exponent and fraction depend on the regime size. However, the hardware bit-widths cannot change dynamically, they have to be designed for the worst case widths. This worst case therefore corresponds to the smallest fixed-precision floating-point format that is a superset of posit numbers. To match posit encoding, this floating-point format only includes normal numbers with fractions expressed in two’s complement.

The parameters for the smallest fixed precision floating-point superset are derived as follows. The size W_F of a two’s complement fraction is the maximum precision of a posit value, obtained for the minimum length $l = 2$ of the

Table 2.7: Parameters of standard posit formats.

Standard posit			smallest FP superset		internal formats	
N	W_{ES}	$E_{\max}$	W_E	W_F	W_{pif}	W_{upif}
8	0	6	4	5	12	14
16	1	28	6	12	21	23
32	2	120	8	27	38	40
64	3	496	10	58	71	73

regime, therefore

$$W_F = N - (3 + W_{ES})$$

The maximal exponent is obtained when the regime length is $N-1$. In this case, all the e_s and F bits are pushed out by the regime. Hence, the maximum exponent value is

$$E_{\max} = (N-2)2^{W_{ES}} \tag{2.1}$$

As the opposite exponent can also be reached, the number of bits needed to store the exponent is

$$\begin{aligned} W_E &= 1 + \lceil \log_2 \left((N-2)2^{W_{ES}} \right) \rceil \\ &= 1 + W_{ES} + \lceil \log_2 (N-2) \rceil \end{aligned} \tag{2.2}$$

The W_{ES} parameter allows trading between the range of the format and its maximal precision.

Table 2.7 gives, for each of the standard posit formats, the exponent and fraction sizes of the smallest floating-point superset.

The hardware-friendly posit Intermediate Format

With this smallest posit FP superset, it becomes possible to define a new intermediate encoding for numbers in this set that is more adapted to hardware arithmetic operations, in the sense that all its fields have a fixed width. In this work, this encoding scheme is denoted posit Intermediate Format or PIF.

Figure 2.4 highlights the role of this format in an end-to-end posit arithmetic operator. Posits are first decoded to PIF. As PIF can represent all posit values, this operation is exact. Then, the arithmetic operation is performed on PIF data. Finally, the result is encoded back to posit. Since rounding (to an exponent-dependent position) must be performed in this encoding step, the output format of the PIF operation must be an Unrounded PIF, which is a PIF extended with all the additional information needed for standard-compliant rounding, detailed below in Section 2.2.3.

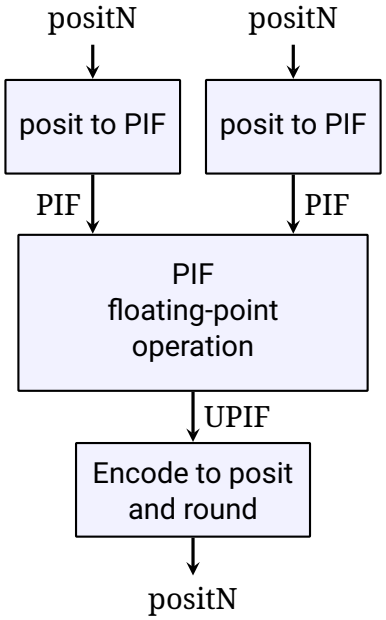


Figure 2.4: Architecture of a posit operator in a Posit Arithmetic Unit that uses posit registers and posit-to-posit operators.

We believe the 3-step approach of Figure 2.4 is inevitable for stand-alone posit operators (except for very small formats where a simple tabulation may be used). It is followed (more or less explicitly) by leading hardware posit implementations [33]–[35].

The PIF should be designed with two objectives in mind:

- Posit to PIF conversion should be as simple as possible,
- Arithmetic operations should be efficiently computed on this representation.

Because of the second objective, PIF is a simple normalized floating-point representation that uses the parameters of Table 2.7. To address the first objective, the proposed PIF encodes both the exponent and significand in two's complement (where IEEE-754 uses a biased encoding for exponent and a sign-magnitude representation for the significand). This avoids two's complement to sign-magnitude conversions (which may cost a carry propagation). It also has the side effect of slightly simplifying PIF addition of values with opposite signs.

Two's complement encoding for a normalized significand consists of a sign bit s and a fraction F on W_F bits. The two's complement significand value is then

$$v = -2s + \bar{s}.F \quad . \tag{2.3}$$

However, (2.3) does not allow the encoding of zero. The proposed PIF encoding scheme introduces an extra bit i which is one if and only if the represented value is strictly positive. The significand value becomes

$$v = -2s + i.F \quad . \tag{2.4}$$

Zero is the only posit value whose PIF representation has both s and i set to zero. This enables efficient zero detection in arithmetic operators. For non zero values, exactly one of s or i is set.

PIF also has an extra *isNaR* bit, set to one if and only if the represented value is NaR. An alternative option could have been to use a non-posit value, for instance with both s and i set to one. This would trade one bit of representation for a few gates of encoding/decoding logic.

To summarize, the PIF encoding scheme is composed of the following fields:

- a *isNaR* flag,
- the sign bit s,
- the exponent E stored in two's complement on W_E bits,
- the weight one bit i,
- the fraction bits F on W_F bits.

The encoded value is

$$v = \begin{cases} NaR & \text{if } isNaR \text{ is one} \\ (-2 + i + 0.F) \times 2^E & \text{otherwise} \end{cases} \tag{2.5}$$

Unrounded PIF encoding of the result of basic operations

In the general case, the result of an operation on two PIF values is not exactly representable as a PIF, and must be rounded. As PIF is a floating-point format, we may use textbook techniques [36], [37] for this. For the basic operations (addition, multiplication, division and square root) the exact result can always be represented on at most $2W_{\text{pif}}$ bits, then for the purpose of rounding the extra W_{pif} bits can be condensed into only two bits:

- an extra fraction bit at the LSB, called the *round* bit;
- a *sticky* bit, set if and only if the exact value is strictly greater than what is represented by the fraction f extended with the *round* bit (but still smaller than the next representable value). In other words, a *sticky* bit of zero means that the value represented by the extended fraction is exact.

We define the UPIF (Unrounded PIF) format as a PIF with these two extra bits.

The floating-point literature often uses a third additional bit (called the guard bit), useful in the case when a 1-bit normalization of the significand may be needed. In the big picture of Figures 2.4, the PIF operator is in charge of this normalization, so no guard bit is needed in UPIF.

This work only demonstrates the use of the UPIF format on addition/subtraction and multiplication, but it is equally suitable for division and square root. Digit recurrence algorithms [36] compute a remainder along with the quotient or square root, out of which the round and sticky bits can be computed. Multiplication-based algorithms [36], [37] also can output their result in UPIF format – for instance by computing the remainder.

Table 2.7 gives the width for PIF and UPIF associated with standard posit formats.

Saturation management

Posit arithmetic does not offer an overflow detection mechanism to the user. When the exact result of an operation is bigger than the biggest representable value, this biggest representable value is returned.

This saturation could in principle be handled in a generic way in the "Encode to posit and round" block of Figure 2.4. However, as each operation leads to different overflow situations, it is more efficient to manage saturation in each PIF operator. The UPIF specification exposed previously actually assumes that saturation management has been performed by the operator, otherwise extra bits would have been needed. Another advantage is that some saturation situations may be detected in parallel with computation, thus reducing latency.

Posit to PIF decoder

The proposed posit decoder is depicted in Figure 2.5.

The "LZOC + Shift" block (LZOC stands for "leading zero/one counter") counts the range bits while discarding them, resulting in a normalized fraction.

The most significant exponent bits E_h are computed out of the range count. If the leading bit is equal to s, then $E_h = -l(= \bar{l}+1)$; else $E_h = l-1$. An optimization is to skip the first range bit when counting, effectively computing $l' = l - 1$. Indeed, if the first range bit is equal to s, $E_h = \overline{l'+1}+1 = \bar{l'}$, or $E_h = l'$ otherwise. This high bit decoding method improves the state of the art by avoiding an

adder to compute $-l$. The exponent least significant bits E_l are obtained by xoring with s the W_{ES}first bits of the aligned fraction.

The PIF exponent E is the concatenation of E_h and E_l (or is equal to E_h if the corresponding format has $W_{ES} = 0$). The decoder is slightly simplified with $W_{ES} = 0$ posit formats, as it saves the XOR gates labeled $*$ on Figure 2.5. The PIF fraction F consists of the W_F least significant bits of the aligned fraction.

An OR reduction over the $N-1$ rightmost bits of the posit input is used to detect both zero and NaR values, in conjunction with s.

The weight 0 significand bit i is computed out of s and the detection of zero value.

The most expensive parts of this architecture are the "OR reduce" over $N-1$ bits to detect NaR numbers, and the combined leading zero/one counter and shifter.

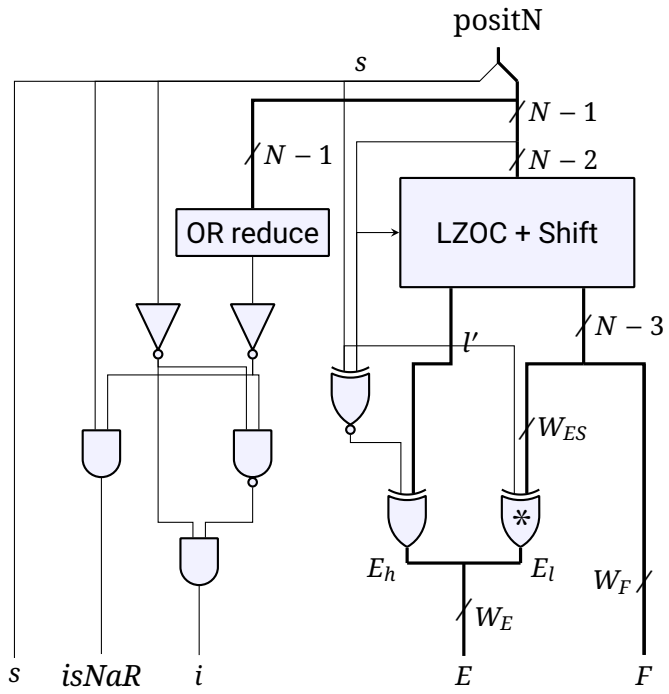


Figure 2.5: Architecture of a posit to PIF decoder.

UPIF to posit and PIF to posit

The complete UPIF to posit encoder architecture is shown in Figure 2.6.

First, the rounding bit is appended to the fraction, and the range is computed then prepended. The Shifter+Sticky component then simultaneously right-shifts this word and ORs the shifted-out bits into a unique sticky bit,

which is then ORed to the PIF sticky bit to get the final sticky bit.

Finally a round-up bit is computed (the right AND of Figure 2.6 implementing round to nearest, and the left AND the “ties to even” rule), and added to the final encoding.

The case $W_{ES} = 0$ requires additional logic (not shown in Figure 2.6) to detect and forbid a special case of rounding up that would cause the output to round to NaR or 0.

PIF floating-point operations

The architectures of the PIF adder/subtracter (Figure 2.7) and multiplier (Figure 2.8) first compute the exact result (top part of the figures) using the transposition to the PIF format of classical floating-point algorithms.

Although the adder is a single-path architecture [37], its datapath can be minimized thanks to the classical observation that large shifts in the two shifters are mutually exclusive. Indeed, the normalizing LZOC+Shift of Figure 2.7 will only perform a large shift in a cancellation situation, but such a situation may only occur when the absolute exponent difference is smaller than 1, which means that the first shift was a very small one. Conversely, when the first shifter performs a large shift, the rightmost part of the significand can be immediately compressed into a sticky bit, since we know that it will not be shifted back by the second LZOC+Shift. All this allows us to keep most intermediate signals on $W_F + 2$ to $W_F + 6$ bits, where previous work [33], [34] seem to use datapaths that are twice as large.

The posit multiplier shown in Figure 2.8 is straightforward. It performs the addition of the exponent, the signed product of the significands, then if necessary normalises the result (one-bit fraction shift and exponent update). This architecture aims at minimal area and delay. If energy efficiency is the goal, alternative architecture have been proposed that exploit the relation between exponent magnitude and precision to disable the computation of unneeded product LSBs [38].

The bottom part of Figures 2.7 and 2.8 normalize the exact result computed by the top parts to a PIF. For both operators, the exact significand must be realigned, correcting the exponent accordingly.

Hardware support for exact accumulation

Several existing machine learning accelerators [39], [40] already use variations of the exact accumulator to compute on IEEE-754 16-bit floating-point. Other application-specific uses have been suggested [41], [42]. For larger sizes, this could be a useful instance of “dark silicon” [43].

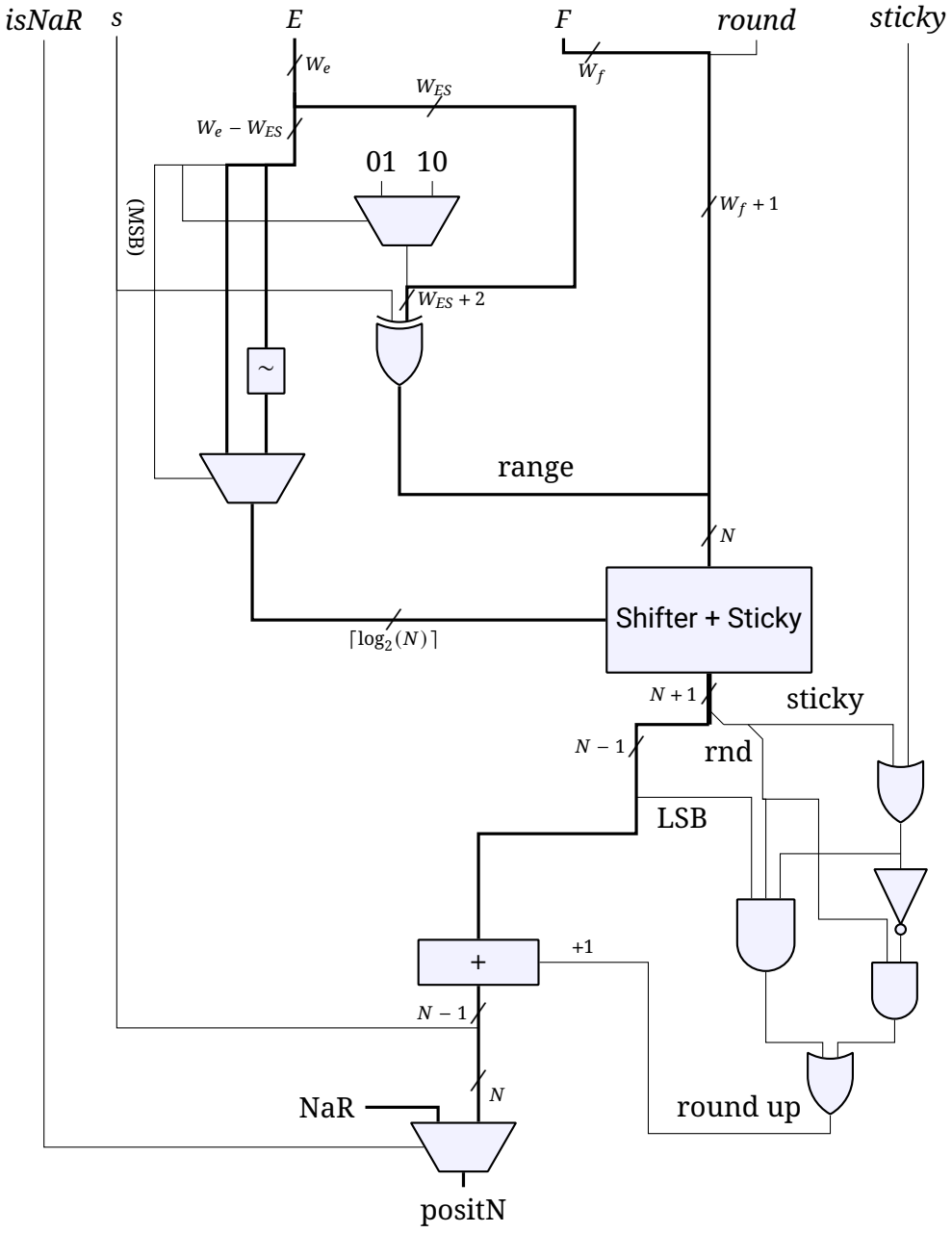


Figure 2.6: Architecture of a UPIF to posit encoder.
The PIF to posit encoder is similar, with the round and sticky logic (including the final adder) removed.

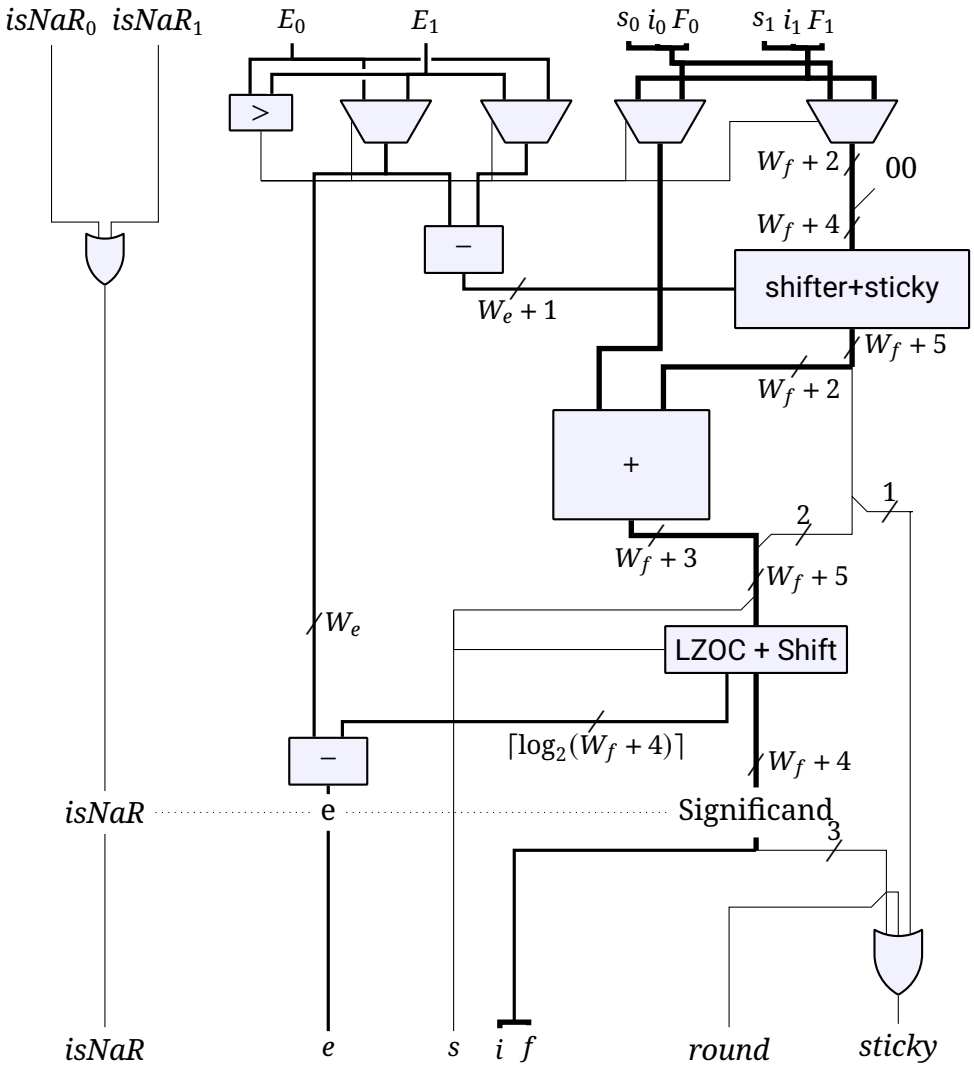


Figure 2.7: Architecture of a PIF adder. Exponent comparison block denoted with “>” also takes the operand i and s bits to detect zero values, but wires have been omitted here for clarity.

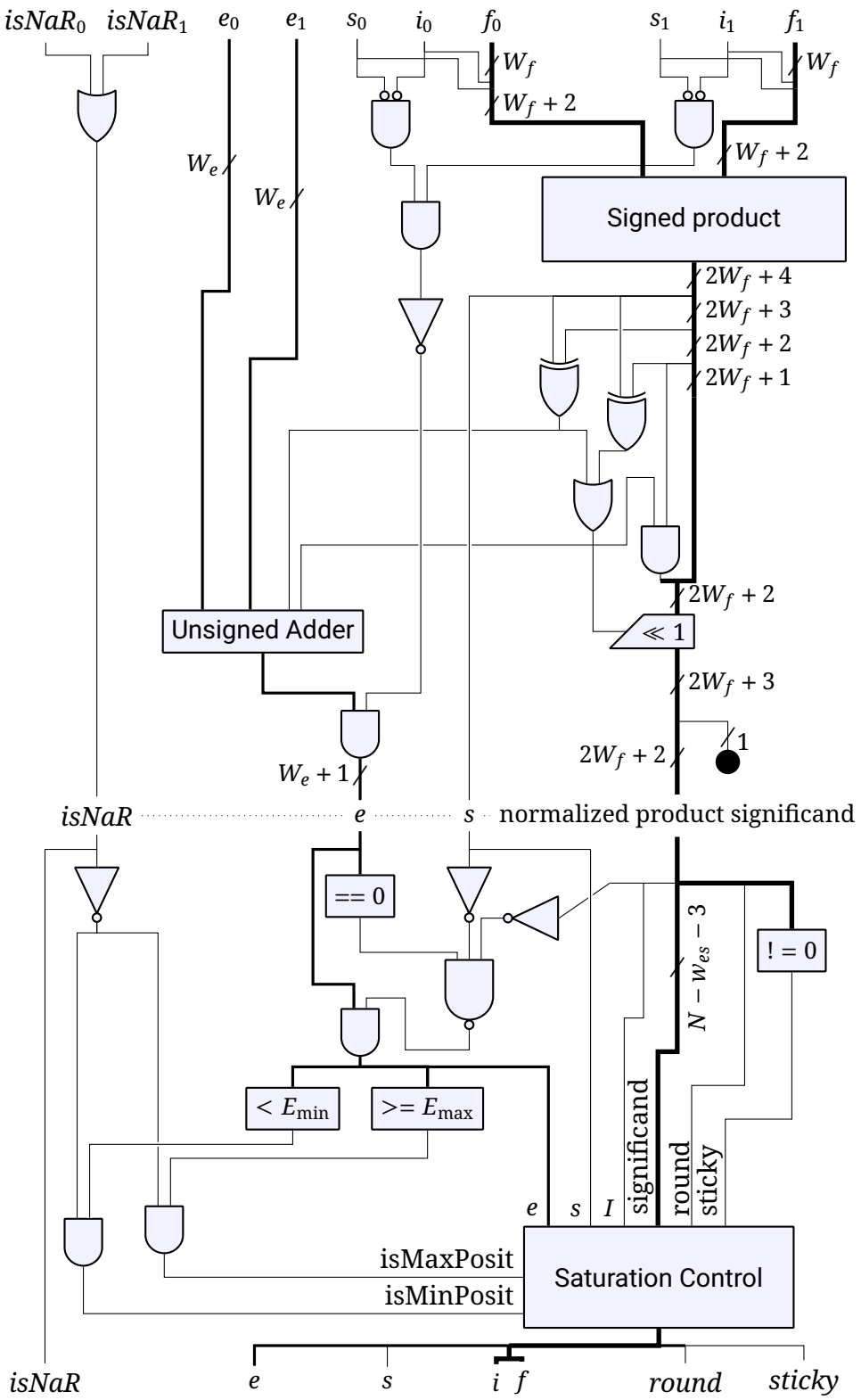


Figure 2.8: Architecture of a PIF multiplier.

Table 2.8: Quire bit-width parameters for standard 3.2 posit formats.

Posit		Quire sizes				
N	W_{ES}	C	W_Q	W_O	W_R	W_Z
8	0	6	32	12	13	6
16	1	14	128	42	57	28
32	2	30	512	150	241	120
64	3	62	2048	558	993	496

The posit standard, [8], [9] defines fused operations as *those expressible as sums and differences of the exact product of two posits; no other fused operations are allowed.* It also specifies a binary interchange format, the quire, which consists in a fixed point number of size W_Q defined by

$$W_Q = P_{\max} + C - P_{\min} + 1$$

With $P_{\max}$ and $P_{\min}$ the maximal and minimal possible exponent for a given posit format product, and C the number of extra-bits used to catch overflows. In the sequel, we discuss the cost of hardware support for accumulating number in a quire. For draft standard formats, C is set such that

$$W_Q = \frac{N^2}{2}$$

As the parameter W_Q is a storage requirement, it defines a lower bound of the area cost. Figure 2.9 shows a high-level functional description of a quire accumulation, and shows that there is a W_Q-bit addition on the critical path from the quire to itself, which would also entail a large cycle delay. A technique that relaxes this constraint is reviewed in Section 2.2.3.

The posit standards [8], [9] specifies NaR as a special quire value. This means that this special value must be tested at each new quire operation. Instead, we add to the internal quire an isNaR flag bit, set when a NaR is added to the quire, and sticky until the quire is cleared. This isNaR bit can be encoded and decoded only when transferring quire to/from memory, however we suggest that it could even replace one of the quire carry bits in the interchange format.

In the posit context, it is natural to use two's complement for managing signs in the quire. Some implementations of Kulisch's exact accumulator seem to use a sign-magnitude representation for the accumulator [44], matching the sign-magnitude representation of IEEE floating-point. However, a two's complement representation of the accumulator is more efficient even in the IEEE-754 context [39], [45].

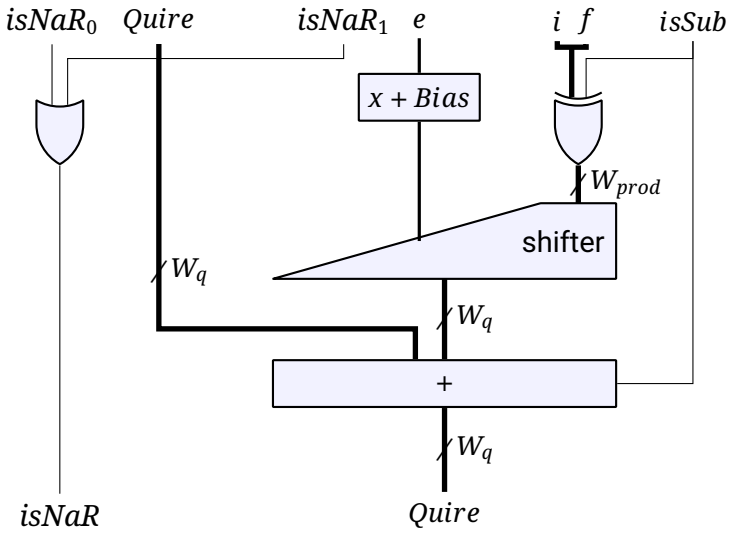


Figure 2.9: Architecture of a posit quire addition/subtraction.

Addition of products to the quire

The posit quire is able to perform exact sums and sums of products. Therefore, the input format of the quire is defined as the output of the exact multiplier from the top half of figure 2.8 (before the rounding logic).

To add a simple posit to the quire, it is first converted to PIF, then the PIF value is trivially extended to the exact product format.

The simplest implementation of the quire addition/subtraction is depicted in Figure 2.9. An exact posit product significand is shifted to the correct place to the quire format according to its exponent. A large adder then performs the addition with the previous quire value. The two's complement subtraction is performed at the cost of a XOR on the input and a carry-in to the adder, as in the posit adder/subtracter. For this the shifter must be a sign-extending one.

The simple architecture of Figure 2.9 can be used directly for small sizes (up to posit16). For larger sizes, the long carry propagation delay of the addition in this architecture will restrict the maximum frequency achievable. To address this, a cost-effective solution [41], [45] is to segment the quire into smaller words (typically standard 32-bit or 64-bit words). Carry propagation is then limited to a segment, and the carries between segments are stored in registers and propagated to the next segment during the next cycle. Another point of view is that the quire is kept in a high-radix carry-save redundant format (radix is 2^{32} or 2^{64}). If such a format is used, its conversion to a non-redundant format will incur additional overhead to complete carry propagation. A cheap way of achieving this propagation is simply to dedicate a few

cycles to the completion of the carry propagation, during which the summand input to the quire is kept at zero. The number of carry-propagation cycles is $W_q/32$ for 32-bit segments. These extra cycles are amortized if the quire is used for summing large numbers of values.

Conversion from quire to posit

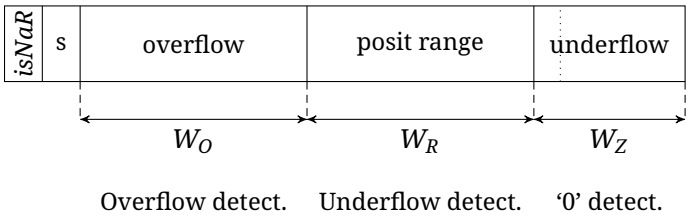


Figure 2.10: The bits of a standard quire.

The conversion of the quire value to a posit is divided in two steps. The quire is first converted to a UPIF value before the latter is encoded to a posit.

There are four distinct cases to take into account when converting the quire to the UPIF:

- If the quire holds a NaR value, the result is NaR;
- If the quire value is larger in magnitude than the maximum-magnitude posit (overflow), the latter should be returned (saturation);
- If the quire value belongs to the representable posit range, it should be converted;
- If the quire value is smaller in magnitude than the minimum-magnitude non-zero posit (underflow), the latter should be returned (saturation);

Figure 2.10 illustrates the different interesting zones of a quire.

Detection of overflow consists in comparing all the bit in the overflow zone with the sign bit. If at least one differs, the posit overflows. The width of the overflow zone W_O is given by:

$$W_O = Q_{\text{msb}} - E_{\max} = E_{\max} + C$$

For quire values inside the posit range, a normalization should be performed, which uses a LZOC + shifter of input width W_R, with

$$W_R = E_{\max} - E_{\min} + 1$$

Finally, to detect the difference between an underflow value and a real zero, a wide OR is performed on the underflow zone of width W_Z, with

$$W_Z = E_{\min} - Q_{\text{lsb}} = E_{\max}$$

For draft standard posit sizes, the simplified formulas are the following:

$$W_O = \frac{N^2}{8} + \frac{3N}{4} - 2$$
$$W_Z = \frac{N^2}{8} - \frac{N}{4}$$
$$W_R = \frac{N^2}{4} - \frac{N}{2} + 1$$

Associated numerical values reported in Table 2.8.

Verification of functionnal correctness of the operators

In order to verify that the proposed architectures are correct, the following functional tests were first run in sofware:

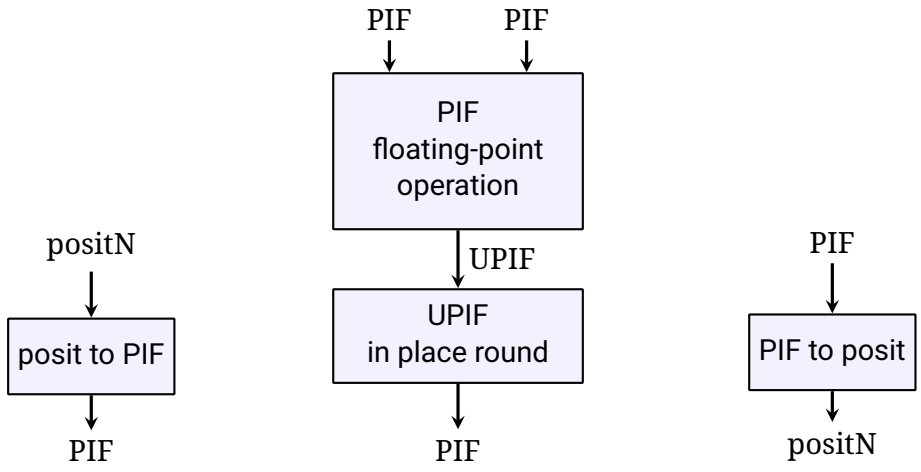
- Exhaustive test against softposit of posit8 and posit16 addition and multiplication.
- Some corner case tests of quire addition/subtraction and conversion back to posit for posit16.
- Exhaustive test for addition/product of IEEE16 against SoftFloat, for the five IEEE-754 rounding modes.

Then the VHDL file produced by the Vivado HLS compiler for a 16-bit posit adder was exhaustively tested using a VHDL simulator. Scripts and sources to reproduce this experiment are accessible from the MArTo repository.

Finally, the standard posit16 multiplier was synthesized, placed and routed for the Zynq FPGA of a Zybo board using the Vivado HLS toolchain, and the resulting FPGA circuit was exhaustively checked against SoftPosit executed on the ARM core of the Zynq. All these tests were successful.

2.3 Case study: comparing IEEE-754 and posit hardware implementation cost

Many works have compared the accuracies of posit and IEEE-754 floating-point formats [46]–[49]. However, these works didn't evaluate the performance implication of replacing IEEE-754 by posit. This is the aim of this case



(a) Memory read (b) Arithmetic operation (c) Memory write

Figure 2.11: Architecture of a PAU using posits as a memory-only encoding, with PIF registers and PIF-to-PIF operators.

study, which will evaluate two possible architecture for posit arithmetic units. The first one is the one described on figure 2.4, with posit used both as memory and internal register format. As shown on the figure however, this mostly adds a decoding step overhead to a classical floating-point operation (the PIF have no subnormal to support that simplify a bit these operators compared to IEEE-754, but the posit encoding is actually very similar to subnormal handling). As an alternative, posit can be used as memory format only, decoded on the memory path, and PIF as register format. The block-diagram of this architecture is presented on figure 2.11.

In order to avoid double-rouding and related issues, PIF in register should always represent the rounded result of the operation. This is the reason of the “UPIF inplace round” block of Figure 2.11b shows an UPIF in place round block. This block is roughly equivalent to the UPIF to posit encoder of figure 2.6, with the difference that a mask is shifted instead of the representation to. PIF to posit conversion that occurs when writing the value back to memory (as shown on figure 2.11b) is a simplification of the UPIF to posit encoder. Indeed, PIF values stores exactly representable posit values, so all the rounding logic can be removed.

2.3.1 Comparison of operator area and latency

Qualitative comparison

A first “qualitative” comparison between posit and IEEE-754 operators can be performed at the architecture level. There is no 1-to-1 match to the decoder

step in IEEE-754 arithmetic operator implementation, as the representation already has fixed-width fields. However, it can be compared with the detection of special cases. On the IEEE-754 side, one OR reduction on the exponent bits is needed to detect subnormals, another OR reduction on the significand bits is needed to detect zeros, and two similar AND reductions are needed to detect respectively NaN and infinities. The two OR reduction have a similar input width in total to the posit decoder OR reduction block. Normalization of subnormal values requires a LZC and a shifter, that operate on the fraction bits. On the posit side, the combined LZOC + shifter operates on a wider value. Besides, the fact that it is an LZOC and not an LZC inclurs internal AND reductions that matches the cost of the IEEE-754 ones. All in all, the posit decoding process cost is expected to be very similar to the handling of special cases and subnormal values in IEEE-754.

The PIF arithmetic operator implementations are very similar to IEEE-754 operators, so area and latency should be quite similar.

Handling rounding and special cases encoding in IEEE-754 requires one integer addition of the representation width, a few gates to determine if the rounding should occur according to the rounding mode, and a few bitwise masking operations to handle the generation of NaN/infinities if required. By contrast, posit rounding and encoding process which requires the same addition, but also a potentially large shift seems more costly.

As a whole, it is expected that posit operators require more resources and have a higher latency compared to IEEE-754 operators for identical widths formats. The expected overhead coming mostly from an expensive rounding and encoding process.

Comparison with state of the art

As we eventually observe that posits are larger and slower than IEEE floats, it is important to be convincing that we are not using a substandard posit implementation. For this purpose, Tables 2.9, 2.10, 2.11 and 2.12 gather the results of best-effort comparisons with the state of the art in posit hardware at time of writing. It shows that MArTo definitely improved this state of the art.

There is less pressure to show that the MArTo implementation of IEEE floats is efficient. A comparison with Xilinx implementation of IEEE floats is provided in Table 2.14. There, the line labeled Xilinx Float corresponds to IP used by Vivado HLS when using the `float` and `double` data types in the HLS C++ (hence the lack of 16-bit results). This hard IP is the industry standard when using Vivado, and can be considered a state-of-the-art implementation

Table 2.9: Comparison with [34] for standard posit addition and product

	Op	Format	LUT	DSP	Delay (ns)
[34]	+	<16, 1>	391	0	32.4
		<32, 2>	981	0	40.0
	×	<16,1>	218	1	24.0
		<32, 2>	572	4	33.0
MArTo	+	<16, 1>	**299**	0	**24.2**
		<32, 2>	**704**	0	**33.9**
	×	<16,1>	**213**	1	**19.4**
		<32, 2>	**483**	4	**28.9**

As no sources is provided, we report as-is the figures from [34], obtained for a Zynq-7000 (xc7z020clg484-1) with Vivado 2017.4. To limit the possible effect of tool improvement on the synthesis, MArTo synthesis results have been obtained for the same part with Vivado HLS/Vivado 2018.3, the oldest version available for download at time of experimentation.

Table 2.10: Comparison with [33] on standard posit addition and product

	Op	Format	ALM	DSP	Cycles	FMax (MHz)
[33]	+	<16, 1>	~500	0	~49	~550
		<32, 2>	~1000	0	~51	~520
	×	<16, 1>	~330	1	~35	~**600**
		<32, 2>	~600	**1**	~38	~**550**
MArTo	+	<16, 1>	**274**	0	**11**	**564**
		<32, 2>	**696**	0	**17**	**562**
	×	<16, 1>	**280**	1	**15**	**600**
		<32, 2>	**452**	2	**21**	445

Synthesis reported in [33] trarget Stratix V FPGA. Results are read from a graphic plot, hence the approximate values. As there is no version of the Intel HLS toolchain that supports both Stratix V and the C++ 11 standard used in MArTo, the C++ to HDL compilation is done using Vivado HLS. The obtained HDL is then synthesised and routed for Stratix V using Quartus. Despite being baroque, this toolchain seems to give good results, except for the <32, 2> product where it lacks the knowledge of the target's DSP possible configurations. Indeed, the product is computed using a 36x36 configuration of the DSP block, where a 27x27 configuration would be faster.

Table 2.11: Comparison with [35] on posit<32,6> addition and product

	Op	LUTs	DSPs	Cycles	Delay (ns)
[35]	+	946	0	**5**	4.1
	×	854	**1**	**6**	4.4
MArTo	+	**792**	0	**5**	**3.9**
	×	**435**	2	**6**	**4.1**

MArTo synthesis have been performed using Vivado HLS/Vivado 2020.1 using part xc7vx330t-ffg1157-3. Experimental settings of [35] use the same part, but tool version is not reported.

Table 2.12: Comparison with [50] for standard posit addition and product

	Op	Format	LUT	DSP	Delay (ns)
[50]	+	<16, 1>	383	0	27.25
		<32, 1>	939	0	35.8
	×	<16,1>	**201**	1	20.9
		<32, 1>	571	4	29.2
MArTo	+	<16, 1>	**300**	0	**25.5**
		<32, 1>	**672**	0	**34.5**
	×	<16,1>	205	1	**19.2**
		<32, 1>	**472**	4	**28.8**

MArTo synthesis have been performed using Vivado HLS/Vivado 2020.1 using part xc7z020clg484-1.

of floating-point for Xilinx FPGAs. It supports some of the IEEE features, such as infinity and NaN encoding. However, it is not IEEE-compliant: although the memory format is that of IEEE floats, subnormals are flushed to zero to save resources.

Considering that the Xilinx Float adders use DSP blocks to implement some of the shifters, the hardware costs of Xilinx adders and IEEE adders (obtained with MArTo) are really comparable. This illustrates that the overhead of subnormal handling in floating-point adders is small. Conversely, there is in Table 2.14 a very large difference in the resources used in multipliers. This demonstrates the cost of hardware subnormal handling in floating-point multipliers.

The Xilinx IP pipelining also seems to be fixed, and do not benefit from a relaxed clock constraint to reduce the latency, hence their large latency.

Since Xilinx floats lack subnormal support, the following bases on MArTo only the posit versus IEEE comparisons.

Comparison between posit and IEEE-754 operators

Table 2.13 compares combinatorial implementations of posits and floats of the same size on addition and multiplication. In this table, the "posit→posit" lines present results for the classical posit operators of Figure 2.4. The "PIF→PIF" lines presents results for the posit-compatible PIF operators that use the architecture of Figure 2.11b, including the inplace round component.

A first observation is that posit arithmetic is indisputably both larger and slower than IEEE-754 arithmetic. This contradicts the comparison in [34], which seems to use a very poor IEEE implementation.

As expected, the PIF-to-PIF operators are lighter and faster than the posit-to-posit ones. They still pay the price in area of a wider significand datapath (see Table 2.7) compared to IEEE operators: for the adders, PIF-to-PIF consume more LUTs than IEEE operators; for multipliers, they consume more DSP blocks (there is a step effect due to the discrete nature of DSP blocks). Again we observe in the IEEE multipliers the logic cost of subnormal support, but we also observe a comparable cost in the PIF multiplier, essentially due to the inplace round logic. Still, the PIF to PIF operators achieve delays that are closer to those of IEEE operators than to those of posit operators, which was their main motivation.

Note that the area cost of PIF/posit conversions (altogether about half the size of a complete IEEE adder) must still be paid in a posit arithmetic unit that uses the PIF-to-PIF approach. Only its delay (altogether about half the delay of a complete IEEE adder) is avoided. However, there is another advantage

Table 2.13: Synthesis results of combinatorial operators

(a) Combinatorial adder

	N	LUT	(ratio)	delay	(ratio)
posit→posit	16	312	1.33	11.1 ns	1.27
	32	647	1.49	15.8 ns	1.33
	64	1550	1.59	21.6 ns	1.35
PIF→PIF	16	237	1.01	9.7 ns	1.10
	32	562	1.29	12.9 ns	1.08
	64	1244	1.27	14.7 ns	0.92
IEEE→IEEE	16	234	1	8.8 ns	1
	32	434	1	11.9 ns	1
	64	976	1	16.0 ns	1

(b) Combinatorial multiplier

	N	LUT	(ratio)	DSP	delay	(ratio)
posit→posit	16	182	1.03	1	11.3 ns	1.39
	32	466	1.37	4	15.8 ns	1.62
	64	1213	1.58	16	21.1 ns	1.48
PIF→PIF	16	120	0.68	1	7.8 ns	0.96
	32	291	0.86	4	11.5 ns	1.17
	64	695	0.90	16	15.3 ns	1.08
IEEE→IEEE	16	176	1	1	8.1 ns	1
	32	340	1	2	9.8 ns	1
	64	768	1	9	14.3 ns	1

(c) Posit - PIF conversion operators

	N	LUT	delay
posit→PIF	16	61	2.59 ns
	32	106	4.74 ns
	64	278	5.52 ns
PIF→posit	16	41	2.12 ns
	32	98	2.50 ns
	64	301	2.83

Table 2.14: Synthesis results of pipelined operators

(a) Pipelined adder

	N	LUT	Reg.	DSP	cycles	delay
Posit	16	320	128	0	4	2.69 ns
	32	719	460	0	7	2.83 ns
	64	1635	1207	0	10	2.93 ns
IEEE	16	193	137	0	4	2.90 ns
	32	435	337	0	6	2.88 ns
	64	1001	880	0	10	2.99 ns
Xilinx Float	32	167	355	2	10	2.43 ns
	64	628	758	3	10	2.43 ns

(b) Pipelined multiplier

	N	LUT	Reg.	DSP	cycles	delay
Posit	16	213	80	1	4	2.85 ns
	32	443	198	4	6	2.93 ns
	64	1140	811	16	12	4.10 ns
IEEE	16	189	122	1	4	2.69 ns
	32	381	246	2	6	2.74 ns
	64	783	801	9	8	2.67 ns
Xilinx Float	32	82	151	3	5	2.72 ns
	64	115	494	11	10	2.75 ns

in a PIF-to-PIF PAU: the hardware for these conversions is naturally shared between different operations (such sharing is also possible in principle in a posit-to-posit PAU, but then it will restrict instruction-level parallelism).

Table 2.14 compares pipelined versions of the same operators, targeting a frequency of 333 MHz (3ns cycle time), and producing one output per clock cycle. As the latency estimated by the Vivado HLS tool is usually pessimistic, the reported latencies are obtained by an automated exploration that finds the smallest pipeline depth allowing the design to run with the target clock period. The script performing this exploration is open source, and is also accessible from the MArTo repository for reproducibility.

There is no PIF to PIF line in this table: for this setup, the PIF to PIF approach fails to provide any latency improvement (the arithmetic operators require the same number of cycles, and sometimes require one more cycle).

Table 2.15: Synthesis results for a sum of 1000 products (U: Unsegmented, S32 and S64: Segment sizes of 32 and 64 bits).

		LUT	Reg.	DSP	cycles	delay
quire 16	U	1200	1026	1	1019	2.70 ns
	S32	978	1062	1	1021	2.68 ns
	S64	1004	958	1	1019	2.36 ns
quire 32 (512 bits)	U	5884	6235	4	1031	*3.65* ns
	S32	3641	7237	4	1040	2.89 ns
	S64	3513	5189	4	1033	2.78 ns
Kulisch 32 (559 bits)	S32	3624	7632	2	1034	2.937
	S64	3612	5165	2	1026	2.801
IEEE Float 32		840	711	2	6012	2.92 ns
IEEE Float 64		1798	1723	9	8015	*3.33* ns
Xilinx Float 32		445	544	3	6008	2.72 ns
Xilinx Float 64		809	1386	11	8013	2.70 ns

We therefore choose not to report these results, which we consider synthesis artifacts as they are inconsistent with the expectations and with Table 2.13.

The cost of supporting all rounding modes in IEEE

Tables 2.13 and 2.14 report result for IEEE operators that only support round to nearest, ties to even. However, supporting the five IEEE-754 rounding modes increase only very slightly the hardware cost. For instance, adding this support to the 32-bit adder increases its area from 434 to 458 LUTs and actually decreases the delay from 11.9 to 11.7ns (another synthesis artifact). It remains well below the posit cost.

2.3.2 Quire versus standard operations

Synthesis results for the quire are given in Table 2.15, where we use MArTo to write a C++ loop that performs the sum of 1000 products and return the result as a posit. They are compared to a similar loop using floating-point Kulisch accumulator, and using regular floating-point hardware.

Quire is presented in unsegmented (U) version along with two segmented versions (S32 and S64 for segments of 32 or 64 bits). For 32 bits, the unsegmented version is not able to achieve 3ns cycle time, due to the long carry

propagation.

The Kulisch accumulator used here is also based on a 2's complement segmented accumulator [45, variant 3], with an IEEE-compliant final conversion to float (round to nearest, ties to even). The implementation has been validated against MFPR [51] simulations.

Unsurprisingly, the cost and performance of a posit32 quire and a Kulisch accumulator for 32 bits floats are almost identical.

An exact accumulator consumes vastly more resources than standard operators: a factor 10 for 32-bit floats (a smaller factor for posits, but only due to the higher cost of the standard operators). Such factors should not come as a surprise: the 512 bits of the posit32 quire are indeed 18 times the 27 bits of the posit32 significand. This is the price of the accuracy of an exact accumulator.

Another advantage of exact accumulation is that it offers a latency reduction proportional to the latency of the floating-point or posit adder (here a factor 6-7). This is thanks to the fact that the accumulation loop is an exact fixed-point addition, which offers opportunities to exploit more parallelism in its computations[39], [42].

Detailed synthesis results of the quire sub-components are given in Table 2.16. The *quire addition* line reports the cost for the architecture of Figure 2.9, including the large shifter and the fixed-point accumulation loop. This component accepts a new input every cycle. The two other lines describe the conversion of the quire result back to posit. The *carry propagation* is a loop component that adds zeroes, and is mostly merged with the *quire addition* component. However, there is an irreducible latency for the final carry propagation once the accumulation is over.

The latency overhead of the expensive conversion from quire to float or posits is easily amortized for large loops. However, it is also clear that a hardware quire will be very inefficient when used for small sequences of operations (e.g. fused multiply and add, complex arithmetic, small matrices or convolutions, etc).

2.3.3 Case study conclusion

This case study both allowed to demonstrate the relevance of having an HLS library for arithmetic components, and to compare the hardware cost of posit and IEEE-754 basic operation implementation.

Regarding the first point, it is simple with such a library to replace the arithmetic type in use for a given application by using a few typedefs. This allows easy experimentation to determine the format that gives the best accuracy/performance trade-off for a given application.

Table 2.16: Detailed synthesis results of hardware posit quire

			LUT	Reg.	Cycles	Delay (ns)
posit16	Quire addition	U	618	885	4	2.576
		S32	403	585	3	1.886
		S64	444	606	3	1.984
	Carry propagation	S32	6	390	3	1.539
		S64	2*	261	2	1.651
	Quire to posit		480	166	3	2.735
posit32	Quire addition	U	3609	4986	7	3.212
		S32	1305	2265	3	2.791
		S64	1389	2276	3	2.791
	Carry propagation	S32	281	2874	8	2.851
		S64	189	2391	7	2.183
	Quire to posit		1845	1457	17	2.878

On the second point, the take-away message of this study is that the indisputable complexity of the IEEE-754 standard, much attacked by posit proponents, does not necessarily translate into expensive hardware. Among the features that the posit system discards as useless, most (in particular overflow management, NaNs, and directed rounding mode) were designed to be implementable at very little cost. The only really expensive feature is subnormal support, due to rounding happening in a variable position of a bit vector. Posit arithmetic, despite the simplicity and elegance of the number system, involves such variable-position rounding, and therefore entail an overhead that is comparable in nature to subnormal support.

2.4 Limits and future work

A first issue with the current library design is the imposed default choice for IEEE-754 number rounding mode. This is however only a design issue and could be solved by adding a policy parameter to the `IEEENumber` template class determining the default rounding mode.

A more fundamental issue comes from the limits of C++ template metaprogramming. While the C++ template system has been shown to be Turing-complete, many obstacles hinders the development of a full library based only on this system.

1. the C++ standard version accepted by the tools accepts only very simple `constexpr` constructs, limiting a lot what is practically writable,
2. there is no tool to move step-by-step into the template instantiation mechanism, making such code difficult to debug and limiting the manageable complexity of code using this system,
3. template instantiation mechanism has impact on compilation time (while this overhead is negligible compared to the time required to place and route an FPGA design, it is very significant when iterating on the software version of the design to debug it),
4. there is no way to reuse external runtime libraries to precompute useful values at compilation time.

This last point is quite annoying: a lot of libraries exists to determine interesting values for arithmetic operators (for instance “good” coefficients for polynomial approximation of arbitrary functions) which would require a huge amount of expertise and a lot of work to rewrite as full compile-time libraries (and suffer from the issues listed above). Next chapter describes two approaches to solve these issues by getting support from the compiler.

3

HLS library for arbitrary fixed-point function approximations

The previous chapter has demonstrated the possibility of creating fully parametrized HLS operators for basic arithmetic operation using only pure C++ meta-programming. This is an interesting first step toward the "ultimate goal" for arithmetic support in HLS, that would be to allow the construction of optimized hardware for arbitrary arithmetic operators (in the sense of section 1.3). A second step toward this goal consists in supporting operators approximating arbitrary unary function, with fixed-point input format and destination format. This is an interesting objective because the generation of such hardware is a well studied problem (a brief overview of the existing solutions is given in section 3.3), but no existing HLS tool support it. Besides, architectures for approximating arbitrary functions requires complex value pre-computation (for instance polynomial coefficients, or function values for tabulation). Computing these values fall behind the limit of what can be easily and efficiently done using C++ meta-programming. That is why alternative methods are required to reach this objective. This chapter introduces these methods.

On the user side, a library allows HLS developers to build operator specifications in their code. Two methods to implement the library backend that convert these specifications to hardware description are proposed. Both of them require a modification of the HLS compilation flow.

The first method, presented in section 3.4, allows doing so without modifying the HLS compiler, by introducing an intermediate compilation step [52]. This approach is portable (as it is not bound to a specific HLS compiler) but requires a bit of extra work from the library user.

When modifying the compiler is an option, an alternative method, transparent to the user, is available. It is presented in section 3.5.

The HLS compiler used to implement this method is a custom C++20 compiler targeting Xilinx devices. It is introduced in section 3.1.

3.1 Custom C++ HLS compiler supporting C++20

This section describes the custom C++ compiler that has been developed for enabling experimentation with C++20/C23 constructs on HLS targeting Xilinx FPGAs.

This compiler has been initially developed to bring experimental SYCL support to Xilinx devices [53]. SYCL is a programming standard from the Khronos Group [22] to program applications using heterogeneous accelerators within a modern single-source C++ framework. Intel has developed a clang fork that support SYCL on various devices, rebranded the oneAPI DPC++[1] compiler [54]. This fork has been itself forked to add support for Xilinx FPGAs. Both forks rebases on the latest Clang/LLVM developments on a regular basis, which allows using all C++ constructs supported by recent `clang++`.

Internally, this custom compiler uses Vitis HLS, as a backend. Vitis HLS consists in a derivative from Clang/LLVM [55], with added support for custom pragmas and some specific optimization passes. The tool is frozen at LLVM version 7. Moving the Vitis frontend back from the past to align it with the current LLVM 15 seems to represent an enormous amount of work. Instead of doing so, the strategy is to feed the LLVM IR produced by a modern `clang++` 15 compiler to Vitis HLS, bypassing the Vitis HLS old `clang++` frontend. This architecture is visible on figure 3.1. The LLVM IR version produced by the recent Clang/LLVM is not directly compatible with Vitis HLS for two reasons:

- Vitis HLS understands only LLVM 7 IR, and LLVM 15 IR has some constructs that do not exist or are invalid in LLVM 7 IR,
- it expects some IR patterns that are no longer emitted by `clang++`. Without them, the quality of result can degrade abruptly. For instance, the canonical way to move objects in recent LLVM is via `memcpy` instructions, while Vitis works better with loads and stores.

So a few LLVM passes that downgrade modern LLVM IR to Vitis understandable IR have been developed. This compiler also offers a few custom C++ decorators to replace Vitis HLS pragmas.

A new Vitis-IP target has been added to the Xilinx SYCL compiler in order to allow the usage of the tool outside SYCL environments. With this target,

[1] For data parallel C++.

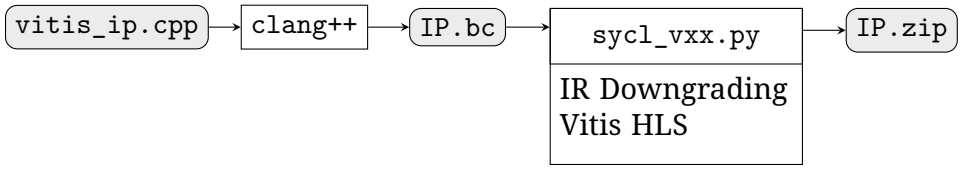


Figure 3.1: Custom C++ HLS compiler stages

the downgrading passes are run and `vitis_hls` is invoked on the result, to produce an IP package that can be imported in a Vivado block design.

A command to launch an IP synthesis is similar to a `clang++` invocation, for instance

```
clang++ --target=vitis_ip-xilinx vitis_ip.cpp -std=c++20 -o ip.zip
↪  --vitis-ip-part=xc7vx330t-ffg1157-1
```

When targeting `vitis_ip-xilinx`, the clang driver will schedule the operations presented on figure 3.1. The modern clang frontend translate the C++ source file to a LLVM 15 Intermediate representation (`IP.bc`). This representation is then fed to a python script, `sycl_vxx.py`, that will schedule the different passes that downgrade the IR version and lower the custom extensions in IR. The same script fed the produced LLVM 7 IR to Vitis HLS, whic produces the result IP (`IP.zip`).

3.2 A library to specify arithmetic operators

As stated earlier, the objective of this work consists in allowing the construction of hardware arithmetic operators having one fixed-point input and an arbitrary approximated function. This section introduces the library that is provided to the user for specifying such operators.

The library is actually made of two layers. A first layer describes elementary fixed-point operations. It can be seen as the counterpart of the `MArTo` library for fixed-point formats, and could actually have been part of it. However, as the compiler used for the experiments supports C++20 concepts and C23 BitInt, it was also the opportunity to test the design improvements enabled by these features. That is the reason why this fixed-point operations library is not a regular part of `MArTo`. In addition, C++20 support enables the use of functions of the standard library that became `constexpr`[2] with this standard. These functions would have otherwise needed to be reimplemented. The same work could however have been done using the ancient C++ stan-

[2]Which means they can be evaluated at compile time and used inside template parameters

dard supported by main vendor HLS tools, as shown in the previous chapter. The fixed-point layer of the library is detailed in section 3.2.1.

The second component of the library allows the definition of mathematical expressions, that are used to specify the function that the operator approximates. This library is detailed in section 3.2.3. The same remark as for the first layer applies: C++20 concepts and `constexpr` standard functions have been used beneficially within the library, but it is perfectly possible to develop an equivalent library without them. This is only a matter of productivity of library development.

3.2.1 C++ types for fixed-point number

In the fixed-point library, a fixed-point format is represented by the template class `FixedFormat`. The two first template parameters are respectively the most significant and least significant bit position (similarly to what is described in section 1.2.1). A third template parameter specifies whether the format is signed. For instance, the type `FixedFormat<31, 0, signed>` represents a fixed-point format which is equivalent to a 32-bit signed integer.

A `FixedFormat` is used to parametrize the template class `FixedNumber` which represents a value of a given format.

A `FixedNumber` can be constructed from the representation of its value. For instance, in Listing 3.1, the variable `X` is initialized with the value 2^{-2} because only its second bit (of weight -2) is set.

```
1  using my_format = FixedFormat<4, -3, unsigned>;
2  FixedNumber<my_format> X{0b00000010};
```

Listing 3.1: Construction of a `FixedNumber` from its representation.

A `FixedNumber` may also be constructed from the value of a standard arithmetic type, thanks to the static method `get_from_value`. This conversion returns the value of the target format closest to the source value. The tie breaking rule is toward $+\infty$.

Listing 3.2 presents an example of computation performed with the fixed-point library.

A user-defined literal (UDL) allows to create a `FixedNumber` from a constant value without having to specify its format. The narrower format that can represent exactly the expression will be automatically deduced from the operator. As decimal floating-point literals may have non-finite representation (for instance $0.1_{10} = 0.000110011001100\ldots_2$), only hexadecimal floating-point literals are accepted by this UDL in the current library state.

```
using radius_fmt = FixedFormat<3, -4, unsigned>;
using pi_fmt = FixedFormat<<1, -4, unsigned>;
auto circumference(FixedNumber<radius_format> radius) {
  // Generate a constant of value two
  // The deduced format is FixedNumber<1, 1, unsigned>{1};
  constexpr auto two = 2._fixed;
  // Generate an approximation of PI rounded to 2^(-4)
  // !! PI is already an approximation of π !!
  auto pi = FixedNumber<pi_fmt>::get_from_value(PI);
  // Diameter is a FixedNumber<4, -3, unsigned>;
  auto diameter = two * radius;
  // circumference is a FixedNumber<6, -7, unsigned>
  auto circumference = diameter * pi;
  // Eventually we get rid of low bits
  // Return type is FixedNumber<6, -4, unsigned>
  return circumference.round_to<-4>();
}
```

Listing 3.2: Format inference in the fixed-point library.

Finally, conversion from `FixedNumber` to standard arithmetic types are expressed with the `get_as` template method from `FixedNumber`. This method takes a target arithmetic type as template parameter and returns the value of this type which is the closest to the represented value (with the same tie-breaking rule as the other conversions).

3.2.2 Classical arithmetic computations

The proposed library defines standard arithmetic operations between `FixedNumber` of different `FixedFormat`, similarly to what the `hint` library does with arbitrary width integers. In the current experimental state, the modern C++ compiler presented in section 3.1 is the only target of this library. This compiler supports C23 `_BitInt`, so a backend using these types has been developed for `hint`. Fixed-point functions are built on this backend[3] to benefit from the strong semantic checking and useful API of `hint`[4].

[3] As the library uses languages features that are not supported by IntelHLS or Vitis HLS, it makes no sense to add genericity to function libraries to support these backends. However, this is a technical decision for the sake of efficiency of library development. It should be emphasized again that there is no conceptual obstacle to build a similar library using pre-C++14 constructs. This is however more tedious and error-prone, hence the technical decision of using C++20 in this experimental work.

[4] Another technical point is that in their current specification, 1-bit signed `_BitInt` are not supported natively. While this has been “fixed” by adding this support at the compiler level in the case of the custom compiler presented here, it will also be handled at the `hint` library

Addition, subtraction and multiplication are computed exactly, the `FixedFormat` of the result being deduced by the library to be wide enough to hold the result. Rounding, overflow or saturation must be managed explicitly: `FixedNumber` provides additional methods to slice it, round to a specific precision, or extend the number to a wider format.

The proper way to handle division and remainder is still not clear.

Some HLS tools provides Arbitrary fixed-point format libraries, as for instance `ap_fixed` and `ac_fixed`, the fixed-point counterparts of respectively `ap_int` and `ac_int`. These libraries offer quite similar functionalities in terms of elementary operation support. They also allow more flexibility in the format definition, by exposing policy parameters to set the overflow and rounding behaviors. They however suffer from the same issue as their integer counterparts concerning the lack of type checking and the transparent casting that can induce counter-intuitive behavior. The original feature from the proposed library is its ability to evaluate arbitrary mathematical expressions of fixed-point inputs. Next section describes how to represent such expressions.

3.2.3 Arbitrary mathematical function specification via its expression graph

Applications often require the evaluation of mathematical functions which are themselves defined as compositions of elementary operations and functions, for instance $f(x) = \log(1 + e^x)$. Such a function can be evaluated at run-time by a composition of library components (here an exponential, an addition and a logarithm). It is however often more efficient to get a single component to evaluate this expression, as illustrated by Figure 3.2. For this purpose, the proposed library also provides a mechanism to describe a function of one variable, then generate an operator to evaluate this function under an accuracy constraint.

In the proposed approach, a composite function may be defined as a C++ expression with a specific property: it has a free variable defined by the `FreeVariable` constructor. More precisely, this expression must be a tree of mathematical operations or elementary functions, and the leaves are either free variables or constants. A valid expression, for the purpose of building an evaluator, should have at most one free variable. The expression tree defines the function that the resulting operator should approximate. Its operand input format is the format of its `FreeVariable` leaf, and the output format is defined when requesting an evaluation.

Listing 3.3 shows how a simple expression tree and a hardware operator

level in the future.

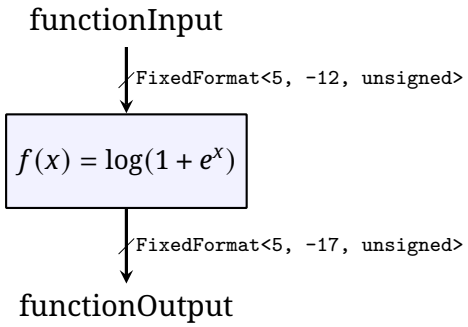


Figure 3.2: Example of function evaluator (here corresponding to listing 3.3).

```
1 FixedNumber<FixedFormat<5, -12, unsigned>> functionInput;
2 functionInput = (...) // HLS code computing functionInput
3 using result_t = FixedFormat<5, -17, unsigned>;
4 FreeVariable x{functionInput}; // defining a free variable
5 auto c = 1_cst;
6 auto f = log(c + exp(x));
7 auto functionOutput = evaluate<result_t>(f);
```

Listing 3.3: Construction of the expression tree corresponding to $f(x) = \log(1+e^x)$ (lines 4-6), and construction of an operator for this function (line 7).

to evaluate it can be specified. At line 4, a `FreeVariable` node is created from a fixed-point variable. The `FreeVariable` format is deduced from the source `FixedNumber`, and corresponds to the operator input format in the formalism of section 1.3. At line 5, a `Constant` leaf is created using a user-defined literal[5].

The library also has special constants to represent numbers like π, which are kept symbolic (infinitely accurate) in the expression tree defining a function. The deduced type of variable `f` at Line 6 represents the expression tree corresponding to the function $f : x \rightarrow \log(1 + e^x)$. The call to the `evaluate` template function specifies an arithmetic operator:

- the function it approximates is given by its argument (here `f`),
- the output format is given as template parameter,
- the input format is obtained from the function free variable leaf,
- the rounding relation is implicit, as only faithful rounding at the moment.

An arguably better design in terms of separation of concern would have consisted in having format-agnostic `FreeVariable`, and pass the operator input as a second argument to the `evaluate` function, that could then be called

[5]It is also possible to create it by specifying the exact type.

`apply`. This might become the new API in future library versions.

Once a specification is obtained, Hardware matching it can be generated. Next section introduces briefly the architecture used to implement these hardware operators.

3.3 Fixed-point function approximation architectures

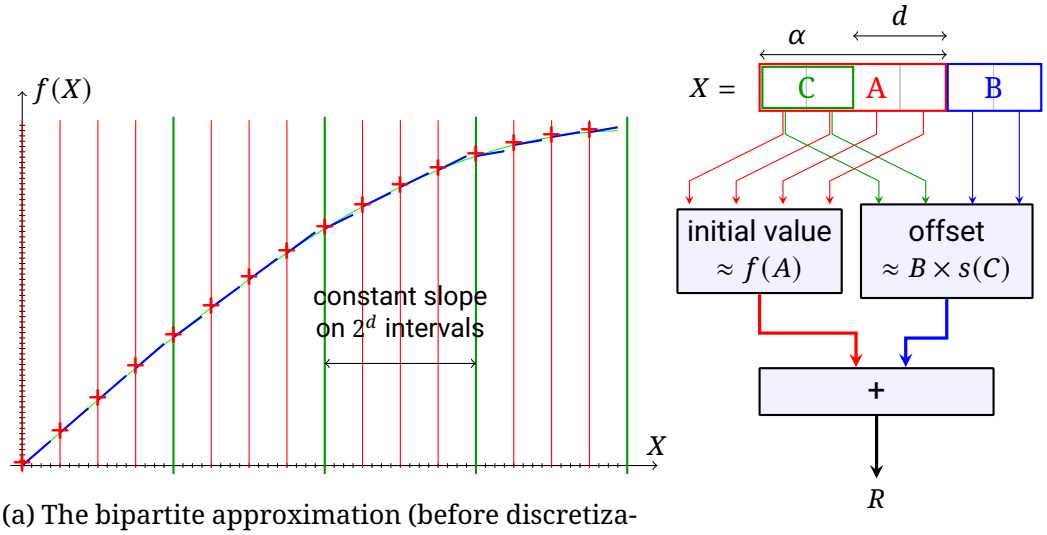
Multiple architecture exists for hardware implementation of arithmetic operators. Two main families exists for general function approximation: table-based methods and polynomial methods.

3.3.1 Table-based hardware arithmetic operators

The simplest form of table-based function approximation operator is plain tabulation. A pre-computed table contains the function output value for each possible input value. The runtime computation consists only in reading the value associated to the input in the table. This architecture is interesting for small input width operators, as it has a low latency. However, as the storage requirement evolves exponentially with the input width, it becomes costly with wide inputs. In this case, an alternative consists in using a multipartite table decomposition.

The simplest form of multipartite method is the bipartite[56], [57] table method. As illustrated by Figure 3.3, this method is based on a piecewise linear approximation. The input domain is partitioned in 2^α sub-intervals by splitting the input as $X = A + 2^{-\alpha}B$, where A (the interval index) consists of the α leading bits of X and B (the index within one interval) consists of the least significant bits. The linear approximation on each subdomain indexed by A is $f(X) \approx f(A) + 2^{-\alpha}B \times s(A)$ where $s(A)$ is the slope of the approximation segment. The idea of the bipartite method is to tabulate the multiplications $B \times s(A)$. To reduce the cost of this tabulation, the slopes are shared between 2^d consecutive intervals, so instead of $B \times s(A)$ it is possible to tabulate $B \times s(C)$ where C consists of the $\alpha - d$ leading bits of A.

Many evolutions to this method have been developed (see [58] and references therein, and [59] for a method for compressing the table of initial values). Due to lack of time, the current flow only provides a basic bipartite approximation: its purpose is to demonstrate that the approach is not restricted to simple tabulation. The current implementation computes the parameters α and d, and fills the corresponding tables for a faithful operator. Plain tabulation is used when no bipartite approximation with a better table storage cost is found.



(a) The bipartite approximation (before discretization)

(b) Bipartite architecture

Figure 3.3: Example bipartite approximation architecture, here replacing a table of 2^6 entries with two tables of 2^4 entries. Source: [32]

3.3.2 Polynomial approximation methods

Polynomial approximation methods consists in finding a polynomial that is close enough to the function to approximate on the operator input domain. The approximated function value is then obtained by evaluating the polynomial, using for instance the Horner evaluation scheme. Finding good polynomial for a given function is not trivial, as two optimization goal coexist: the approximation error should be small enough to respect the overall accuracy constraint, and at the same time the polynomial has to be efficiently evaluated. The later point implies in particular that the polynomial degree should not be too high, and that the coefficients should have a compact binary representation to avoid wide intermediate products.

Tools like `sollya` [60] allows finding good polynomial for a specified function and a specified error bound. These tools are already used by HDL generators such as FloPoCo [25]. The main contribution of this thesis is not to improve the underlying methods, but enable the usage of these tools in an HLS context. The following sections present two methods for achieving this purpose.

3.4 Compiler agnostic specialization generator method

The first method is useful when the HLS compiler is a black box, that cannot be modified. It consists in splitting the HLS compilation process in two steps.

This process is detailed on figure 3.4. The first involves compiling the application for execution on traditional CPU. This compilation step can be done with any off-the-shelf C++ compiler. The executable resulting from this compilation is called the specialization generator.

Running the specialization generators produces C++ implementations for all the operators present in the source application. These specializations are dumped in a specialization header, called `operators.hpp` on figure 3.4. This generated header file can then be included in the application source file `app.cpp`, and be compiled with an HLS compiler.

Here again, there is no constraint on the HLS compiler, except that it should be able to accept both the application and specialization header code[6].

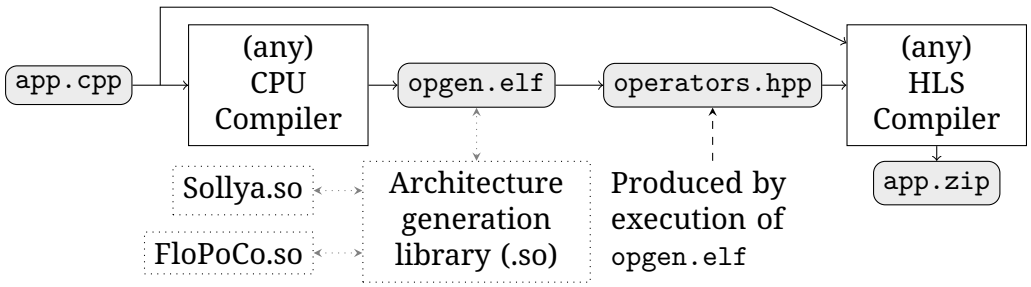


Figure 3.4: Architecture of the generator of template specializations

Following sections detail the internal working of the specialization generator.

3.4.1 C++ types for arithmetic operator evaluation

Listing 3.4 present a toy example that illustrates the main mechanisms that are used in the specialization generation method. One of these mechanism is template specialization. It allows specifying that a different code from the default template code should be generated when a template is instantiated with certain parameters. This is visible with the `TemplatedClass` specialization at line 9 of the listing. The class definition that follows will be selected when the template is instantiated with `int` as its parameter. For all other parameters, the default template class defined at line 2 is used. The second mechanism is the dispatching mechanism used to instantiate a template instantiation based on a function argument type. This is illustrated by the `dispatcher`

[6] A last emphasis is made here to highlight that, while it means supporting C++20 with the proposed library, it is not a strong requirement. Indeed, the concept mechanisms used in the library can be implemented by exploiting the C++ SFINAE (Substitution Failure Is Not An Error) rule, and the `constexpr` functions used can be implemented manually with C++11 constructs.

templated function. When called with a `float` parameter (line 24), the template parameter is deduced to be `float`, so the argument is forwarded to `TemplatedClass<float>` which is instantiated using the default template. At line 27, the call with an `int` argument is dispatched to the `int` specialization of `TemplatedClass`.

The `evaluate` function of the library is used similarly to require a specific instantiation of a template class that depends on the expression type and output format. The outline of this templated class, `ArithOp`, is given in listing 3.5. Its first template parameter, `ET`, is the type that represents the expression tree to evaluate. The second template parameter, `OutputType`, is the destination type of the operator.

The `evaluate` method of `ArithOp` is responsible for computing the result of the operator evaluation. Instead of dispatching the expression to a static method of `ArithOp`, `evaluate`[7] dispatches it to the `evaluate` member method of a templated ArithOp variable. This detail is important, as it has for consequence that the constructor of this variable will be called before entering the `main` function.

In the default `ArithOp` template, this constructor has two responsibilities:

- determine an architecture to evaluate the specific expression of its template parameters,
- generate code that implement this architecture and dump it in the specialization header.

The first point is achieved by using an architecture generation library, which query tools such as `sollya` and `FloPoCo` to determine an efficient hardware implementation for the given expression tree. In detail, a runtime representation the expression tree is built and converted to a representation understandable by the external tools.

In the bipartite generation case, `sollya` is used to build the highly accurate reference table of the function values. This table is then processed to find a good bipartite decomposition. If no good decomposition is found, then the tool falls back to a plain tabulation. The chosen algorithm and associated tables are then saved in the specialization header.

Listing 3.6 gives a simplified example of what the implementation of the specialization of `ArithOp` corresponding to the `evaluate` call at line 7 of listing 3.3. This specialization is generated when running the `opgen.elf` executable. It can be seen that the type specialization is fully specified by a type representing the expression tree $\log(1 + e^x)$ (with x the free variable).

[7] The free library function.

```cpp
template<typename T>
class TemplatedClass{
  static void myAction(T) {
    std::cout << "Default template!\n";
  }
};

// In our case, would be defined in a specialization header
template<>
class TemplatedClass<int> {
  static void myAction(int value) {
    std::cout << "Int specialization: " << value << "\n";
  }
};

template<T>
void dispatcher(T val) {
  TemplatedClass<T>::myAction(val){};
}

int main() {
  // Dispatch to TemplatedClass<float>:
  // Will print "Default specialization"
  dispatcher(3.14f);
  // Dispatch to TemplatedClass<int>:
  // Will print "Int specialization: 42"
  dispatcher(42);
}
```

Listing 3.4: Simple example of template specialization and dispatching.

```
template<ExpressionType ET, FixedFormatT OutputType>
struct ArithOp {
  ArithOp(){
    // Find function evaluation plan
    // and dump it to the specialization header
   }

  // empty default implementation
  FixedNumber<OutputType> evaluate(ET expr) { /*...*/ }
};
```

Listing 3.5: Outline of the generic `ArithOp` template class.

```
template<>
struct ArithOp<
  /*ExprType=*/
  LogExpr<
    SumExpr<
      Constant<1, ...>,
      ExponentialExpr<
        FreeVariable<...>>>>,
  /*OutputType=*/FixedFormat<5, -17, unsigned>> {
auto evaluate(LogExpr<...> expr) {
  // Detailed instantiation, for instance polynomial evaluation
  // or table read
}
};
```

Listing 3.6: Simplified example of generated `ArithOp` specialization for $\log(1+e^x)$ with output format $\mathrm{ufix}(5,-17)$.

Such specializations are generated for each operator specification present in the program. As a consequence, when including the specialization header, there is no more use of the default template. As these specializations have an empty constructor, and their `evaluate` method are pure functions that only use fixed point constructs, they can be fed to the HLS compiler.

3.4.2 Application example

The next sections present two applications that have been built using the library in specialization generator mode[52]. One of these examples is also shown to integrate well in a SYCL environment.

A narrow LNS adder

As seen in section 1.2.3, sum and subtraction in LNS format requires evaluating the function $f_{\oplus} : x \mapsto \log_2(1 + 2^x)$. This function is a good example of composite functions that can be implemented using the techniques described in this chapter.

In order to demonstrate the capabilities of the library, a simple LNS adder has been developed. The simplified adder is presented in listing 3.7.

```
1  // lns_t is a specialization of FixedNumber
2  lns_t LNSAdder(lns_t op1, lns_t op2) {
3    auto min_exp = min(op1, op2);
4    auto max_exp = max(op1, op2);
5    min_exp -= max_exp;
6    auto diff_fv = FreeVariable{min_exp};
7    auto f = log2(1_cst + pow(2_cst, diff_fv));
8    auto rounded_f = f + lns_t::rounding_constant;
9    auto result_exp_diff = evaluate<lns_t::add_eval_t>(rounded_f);
10   max_exp += result_exp_diff;
11   return max_exp;
12 }
```

Listing 3.7: A simplified LNS adder

The LNS Adder has been synthesized for an `lns_t` having an exponent of `FixedFormat<5,-6, unsigned>`. This requires to evaluate $f_{\oplus}$ with an accuracy constraint of 2^{-8}. The report after synthesis and place-and-route shows that the design uses two BRAM, which matches the expectations.

Additive sound synthesis

Additive synthesis is a sound synthesis technique typically used in electronic music to create timbres by adding sine waves together [61] as in the following computation :

$$y(t) = \sum_{k=1}^{K} r_k(t) \sin(2\pi f_k t + \phi_k) \ .$$

It allows very rich sounds when a lot of sines are used, but then it is very compute-intensive. When used in an interactive configuration with hardware in the loop to simulate real instruments interacting in real-time with the real world physics, it requires very low latency.

In order to check the impact of the library, an additive synthesis kernel was developed both with `FixedNumber` and `binary32` computation. Both versions

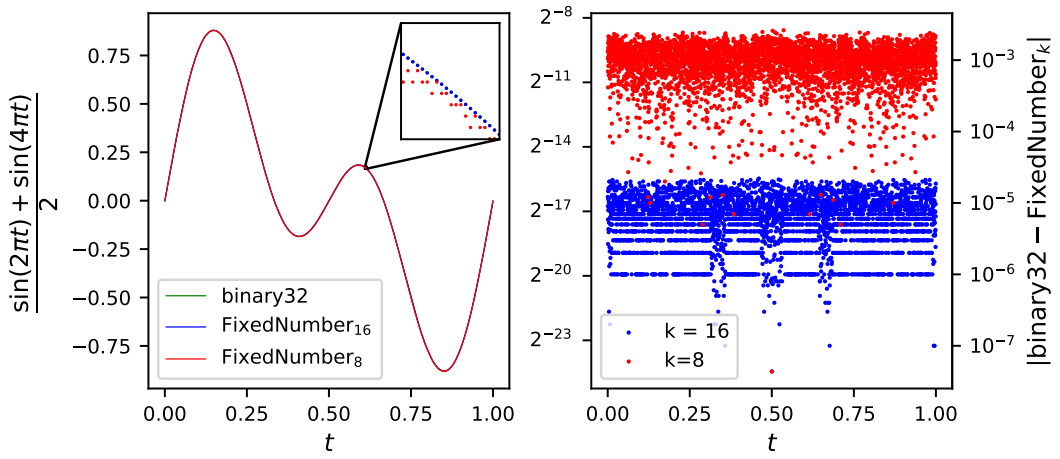


Figure 3.5: Output of an additive synthesis with two frequencies of equal amplitudes and the computational error (as absolute difference on a logarithmic scale).

have an underlying collection of 256 oscillators, having a frequency $f_k = \frac{k*1000}{256}$ and phases $\phi_k = 0$. The amplitudes r_k is an input of the kernel. In both versions, the oscillators are instantiated using template unrolling, to ensure that they are inlined.

In the `FixedNumber` implementation, the r_k are fixed-point number of format `FixedFormat<-1, -8, unsigned>`. The sine is computed from a wave table of 2^{12} entries generated by the library. The function thus evaluated is $f(x) = sin(2\pi x)$ for x in $[0;1)$[8]. Two experiments, $\texttt{FixedNumber}_8$ and $\texttt{FixedNumber}_{16}$ have been performed with an evaluator accuracy constraint of 2^{-8} and 2^{-16} respectively.

In the `binary32` implementation, the `std::sin` function from the mathematical library is used. All the computations are done in `binary32`.

Function values over one period for all the formats and difference with `binary32` for $\texttt{FixedNumber}_8$ and $\texttt{FixedNumber}_{16}$ are plotted on Figure 3.5. The graphs for all the format coincide perfectly, and as expected, the absolute difference to `float` is smaller for $\texttt{FixedNumber}_{16}$ than for $\texttt{FixedNumber}_8$.

The evaluation is performed using our compiler and targeting a Virtex ultrascale plus FPGA with part number `xcvu13p-fhga2104-3-e`. Both pipelined and unpipelined solutions have been generated for the three formats. Results are given after place and route. The design metrics are reported in table 3.1.

[8]Here trigonometric identities could be used to perform argument range reduction. See [62] for an illustration of this technique. The long-term objective of the library is that such optimizations are performed transparently to the user, so this optimization technique is not relevant in user code here.

Experiment		Area				Timing		
Pipeline	Format	LUT	Flip-flop	DSP	BRAM	Latency	Critical path (ns)	II
Unpipelined	`float`	12389	15222	175	0	653	2.787	-
	$\texttt{FixedNumber}_{6}$	7932	6985	256	129	257	2.907	-
	$\texttt{FixedNumber}_{8}$	8882	7284	256	257	297	2.901	-
	$\texttt{FixedNumber}_{16}$	5777	6917	256	513	284	2.995	-
	$\texttt{FixedNumber}_{22}$	6551	7304	256	769	270	2.945	-
Pipelined	`float`	141491	166723	1304	0	226	2.995	128
	$\texttt{FixedNumber}_{6}$	7442	4977	256	128	5	2.803	1
	$\texttt{FixedNumber}_{8}$	7568	6500	256	256	5	2.701	1
	$\texttt{FixedNumber}_{16}$	4513	8806	256	512	8	2.806	1
	$\texttt{FixedNumber}_{22}$	5627	9385	256	768	8	3.074	1

Table 3.1: Area and timing metrics comparison between `float` and various `FixedNumber` for an additive synthesizer of 256 oscillators. II stands for Initiation Interval, the number of clock cycles that pass before the pipeline is ready to be fed a new input.

These results show that fixed-point is definitely a good option for this application. Three factors can explain this better quality of results. The first one is a question of bitwidth of the argument that have to be moved in the design. Indeed, the 256 8 bits amplitude coefficients input of the `FixedNumber` case can be implemented as a very wide input register while it is not possible for the 256 32bits coefficients of the input. Secondly, the intrinsic arithmetic of floating point is more complex than fixed-point on the operation involved in this application. Lastly, the fact that floating-point arithmetic is based on external IP integrated by the design can hinder some optimization opportunities.

Single-source end-to-end LNS with SYCL

As the compiler used in our developments also offers an experimental support for compiling SYCL programs targeting AMD FPGAs, we tried to use the library in a full end-to-end single-source application. In the SYCL compilation flow, the computation kernels are outlined by the compiler, and compiled separately for the hardware targets. Instead of producing a Vitis IP, it generates a x86 fat binary that embeds the hardware kernel binary, and contains all the glue to invoke the kernel execution on the FPGA. A small SYCL application was developed, consisting in one kernel that computes the sum of two LNS number having an exponent of format `FixedFormat<7,-8>`, using the adder shown on listing 3.7. This application has been built to run on Xilinx devices. This required a bit of manual adjustment of the SYCL compilation flow, but nothing that could not be automated in the future. It was then possible to run the program that moved the operands to the FPGA, launched the kernel, and get the result back. No resources are reported here, as the hardware synthesized to ensure communications between the FPGA and host CPU is way bigger than the small LNS adder.

3.4.3 Limits of the approach by specialization generation

Using the type system to represent the expression raise a fundamental issue. Indeed, it is not possible to distinguish two variables sharing the same type. This is in particular problematic with the `FreeVariable` class. Listing 3.8 illustrates this issue: the `FreeVariable` `x` and `y` share the same type due to the fact they are created with values of the same `FixedNumber` type. As such, there is no way to distinguish between a valid expression tree that uses multiple time the same `FreeVariable` (line 6) from an invalid expression that mix usages of `x` and `y` (line 8). The approach followed by the library is very conservative: only expression trees with at most one free variable leaf are considered valid. With this approach, code from listing 3.8 raise a compile-issue complaining

of an invalid expression both for `valid_expr` and `invalid_expr`. This avoids error from the user at a price of reduced expressivity.

```
using input_t = FixedNumber<FixedFormat<5, -12, unsigned>>;
auto some_compute(input_t xVal, input_t yVal) {
  FreeVariable x{xVal}
  FreeVariable y{yVal};
  // Valid because only one free variable
  auto valid_expr = sin(x) + cos(x);
  // Invalid because two distinct free variables
  auto invalid_expr = sin(x) + cos(y);
  using vexpr_t = decltype(valid_expr);
  using iexpr_t = decltype(invalid_expr);
  // The assert will pass
  static_assert(is_same_v<vexpr_t, iexpr_t>);
}
```

Listing 3.8: Example of expression sharing the same type but representing distinct expression tree due to the lack of variable instance type identifying mechanism.

An alternative would be to ask the user to manually specify an uniquing type as parameter to `FreeVariable` (e.g. `FreeVariable<class MyUniqID>{}`). However, this would clutter the library API, stays quite error-prone and would have the side effect that identical expression of two distinct variables will no more share the same architecture generation step.

Another limit of this approach is the relative difficulty to perform optimizations on expression tree. Supposing that the library accept the expression $x + x$ (which is not the case as x appears twice in the expression tree), it would be desirable to have a mechanism to optimize it as $2x$. Similarly, some trigonometric identities could be applied to "canonicalize" the expression representation and limit the architectural exploration for mathematically identical expression of different expression graph. This would require a way to apply a collection of rewriting rules to optimize the mathematical expression. This is actually a good part of the job of optimizing compiler job. However, these compilers first build intermediate representation (IR) of the program that are easy to manipulate. Optimizations are then performed on this IR. Following section explores a method that follows this approach.

3.5 Compiler support for fixed-point functions in HLS

The second approach used to provide support for fixed point function in HLS using the same user interface relies on compiler support. The overall architecture is presented in section 3.5.1. In order to support the actual fixed-function compilation process, intermediate representations have been developed to represent such expressions. These IR are described in section 3.5.2.

3.5.1 Fixed-point function compilation architecture

The architecture diagram of the solution is presented on figure 3.6. The user has only one compilation step to launch.

On this backend, operations on expression and `FreeVariable` constructor are implemented using custom compiler built-ins. When compiling application code, the compiler identify these built-ins, and is able to extract the operator specification from the rest of the code. The rest of the code that does not correspond to operator specification is directly compiled into the compiler internal representation (LLVM IR in this case). This corresponds to the `IP.bc` file on figure 3.6. In parallel, the operator specifications are compiled into a custom IR, which corresponds to `ArithOps.ir` on the figure. A specific compiler (the expression compiler on the figure) is used to convert this high level specification into low level LLVM IR (`ArithOps.bc`). This LLVM IR and `IR.bc` are fed to `sycl_vxx.py`, that will link them together and run optimization and downgrading passes on the resulting code before feeding it to Vitis HLS.

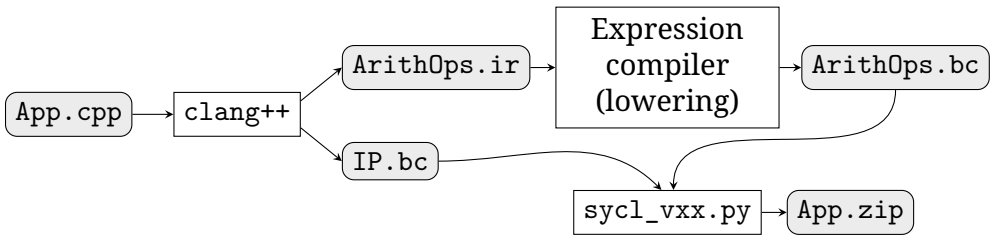
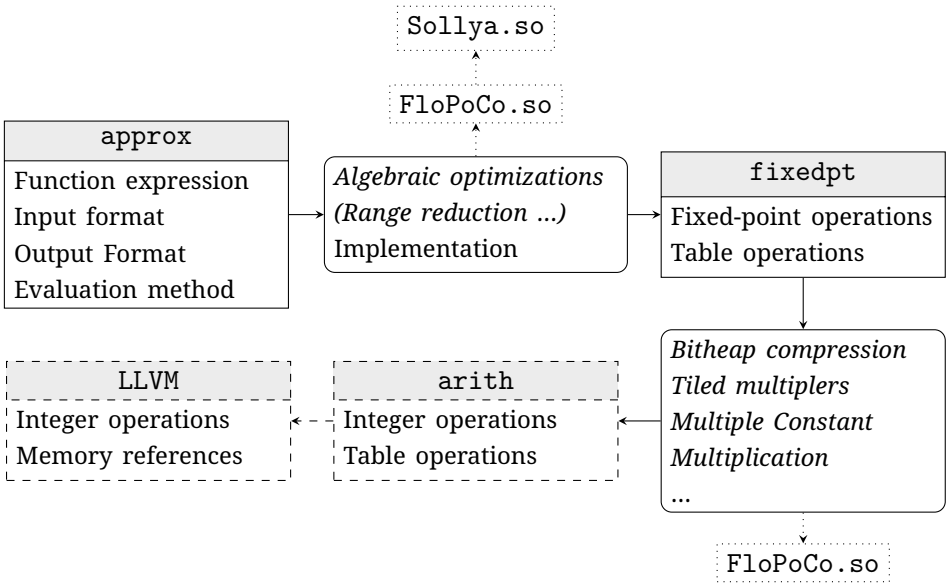


Figure 3.6: Architecture diagram of fixed-point function extracting compiler.

The lowering done by the expression compiler is a multi-stage process, that involves multiple intermediate representations of the operator at different abstraction level. These intermediate representations and some optimizations that are (or could be) performed on them are detailed on figure 3.7. They consists in MLIR [63] dialects. MLIR, for multi-level intermediate representation, is a compilation framework that allows the definition of custom intermediate representations. These IR are called dialects, and consists in collections of custom types and operation on these types. The MLIR compiler framework

provides all the infrastructure needed to define these dialects, apply rewriting rules for optimization and define conversion rules to lower a high-level dialect to a lower-level one.



Italic optimizations are not yet implemented. Dashed IR frames are standard MLIR dialects.

Figure 3.7: IR and optimization used in the process of lowering high-level operator specifications to implementation details.

A standard MLIR dialect exists to represent LLVM code. The aim of the lowering block is then to translate the high-level operator specification given in our custom dialect (the `approx` dialect) to low-level implementation in LLVM IR dialect. Once this representation is obtained, it can be directly translated to LLVM IR, that can be linked with the rest of the application. Next section details these custom MLIR dialects.

3.5.2 Intermediate representations for fixed-point functions

Two custom MLIR dialects describing arithmetic operators have been developed. The first one, the `approx` dialect is an almost one to one mapping with the library constructs for representing mathematical expression. This is illustrated by listing 3.10 that gives the high-level representation produced by the compiler for the operator specified in listing 3.9. The main difference between the C++ code and the `approx` dialect representation of the arithmetic operator is the requirement of having a named node for each intermediate node in the expression graph.

A second custom dialect, the `fixedpt` dialect, is used to represent actual fixed-point operations. An MLIR conversion pass lowers the `approx` dialect

```
auto a = FreeVariable(val_fixed);
auto b = a * pi / 0x2p0_cst;
auto c = sin(b);
auto res = evaluate<
    FixedFormat<0, -4, signed>,
    // Force polynomial approximation (default is heuristic based)
    approx::basic_poly
  >(c);
```

Listing 3.9: C++ specification of an operator approximating $x \mapsto \sin(0.5 * x * \pi)$.

```
%2 = approx.variable %1 : <0, -4, s>
%3 = approx.math "pi"()
%cst = approx.constant <2, <2, 0, u>, "2.0">
%4 = approx.math "mul"(%2, %3)
%5 = approx.math "div"(%4, %cst)
%6 = approx.math "sin"(%5)
%7 = approx.evaluate basic_poly of %6 as <0, -4, s>
```

Listing 3.10: High level representation of operator of listing 3.9 using the `approx` dialect.

to the `fixedpt` dialect. This pass interacts with the external tools FloPoCo and Sollya to determine this plan, and generate the corresponding `fixedpt` IR. Listing 3.11 gives the low-level implementation description of the operator from listing 3.9 using this dialect. This dialect still exposes some level of abstraction. For instance, in this dialect addition or product are variadic operations. This leaves the implementation detail decision to be decided in lower-level dialects.

```
%1 = fixedpt.constant -9 : <0, -4, s>, "-0.5625"
%2 = fixedpt.constant 0 : <1, 0, s>, "0.0"
%5 = fixedpt.constant 25 : <1, -4, s>, "1.5625"
%8 = fixedpt.constant 1 : <-4, -5, s>, "0.03125"
%3 = fixedpt.mul %1 : <0, -4, s>, %0 : <0, -4, s> truncate <1, -8, s>
%4 = fixedpt.add %3 : <1, -8, s>, %2 : <1, 0, s> nearest <2, -8, s>
%6 = fixedpt.mul %4 : <2, -8, s>, %0 : <0, -4, s> truncate <3, -12, s>
%7 = fixedpt.add %6 : <3, -12, s>, %5 : <1, -4, s> nearest <4, -12, s>
%9 = fixedpt.mul %7 : <4, -12, s>, %0 : <0, -4, s> truncate <5, -16, s>
%10 = fixedpt.add %9 : <5, -16, s>, %8 : <-4, -5, s> nearest <6, -16, s>
```

Listing 3.11: Implementation of listing 3.9 operator in terms of fixed-point operations described in `fixedpt` dialect.

Dialects used for low-level implementation description are standard MLIR dialects. Fixed-point operations are lowered to integer operations represented using the `arith` dialect. Current state of conversion from `fixedpt` to `arith` dialect mainly consists in adding the good shifts and truncations to ensure good relative positioning of fixed point operands. It also handles the rounding logic of fixed point ops (with `nearest` tie breaking rule being towars $+\infty$ in this case). This dialect is then converted to the `llvm` dialect and converted to LLVM IR that is linked with the rest of the application before starting the HLS process. As the `llvm` and `arith` dialect are very similar, and the former is a one to one mapping of LLVM IR, only the resulting LLVM IR for the example operator is reported on listing 3.12.

```
%4 = sext i5 %3 to i10 ; extend variable %3 which has type i5 (5-bit integer)
                       ; to i10 (10-bit integer)
%5 = mul i10 %4, -9
%6 = sext i10 %5 to i11
%7 = add i11 %6, 1
%8 = ashr i11 %7, 1
%9 = trunc i11 %8 to i8
%10 = sext i8 %9 to i13
%11 = sext i5 %3 to i13
%12 = mul i13 %10, %11
%13 = sext i13 %12 to i14
%14 = add i14 %13, 8
%15 = ashr i14 %14, 4
%16 = trunc i14 %15 to i9
%17 = add i9 %16, 200
%18 = sext i9 %17 to i14
%19 = sext i5 %3 to i14
%20 = mul i14 %18, %19
%21 = sext i14 %20 to i15
%22 = add i15 %21, 8
%23 = ashr i15 %22, 4
%24 = trunc i15 %23 to i9
%25 = add i9 %24, 4
%26 = sext i9 %25 to i10
%27 = ashr i10 %26, 3
%28 = trunc i10 %27 to i5
```

Listing 3.12: LLVM IR corresponding to the implementation of listing 3.9.

The `tensor` and `memref` standard dialects are used to represent the value tables at high and low abstraction level respectively, for table based approximation methods.

3.5.3 Current state of the expression compiler prototype

The described compiler supports plain tabulation and basic polynomial approximation using Horner scheme. FloPoCo offers other function approximation methods (the state of the art in multipartite approximation [58] and very good piecewise polynomial approximation [64]). These methods rely on fixed-point operations that are already implemented in the `fixedpt` dialect, so adding their support to the compiler should not be complicated. However, this requires refactoring on the FloPoCo side, in order to better separate the approximation logic from the VHDL generation code (that the complete HLS compilation flow replaces). This refactoring has been initiated, but completing it is left for future work.

Correctness of execution has been tested exhaustively in relatively narrow formats, and QoR of synthesized solutions has been manually checked on some examples to ensure sound results. However, there is no automatic testing tools yet, and this compiler project is still at an early development stage.

As it can be seen on listing 3.10, the current `approx` dialect represents all computation nodes with the `approx.math` operation. The first argument of this operation determines the actual computation performed by the node. This design choice was made in order to allow quick addition of computation nodes, but this is not a good choice when it comes to applying high level expression rewriting rule.

Besides, a lot of work remains to be done to optimize the evaluation strategy. For instance, the lowering from the `approx` dialect to the `fixedpt` dialect could benefit from algebraic optimization, for instance detecting how the evaluation range can be reduced. The lowering from the `fixedpt` to the `arith` dialect could also benefit from low-level optimization, for instance by using bitheap compression [65] for efficient sum computations.

In its current state the compiler can help an expert to build an optimized arithmetic operator, and still allow a non-expert to at least get a fixed-point operator for arbitrary function, but the ultimate goal of easily and automatically generate operator with excellent quality of result is not reached yet.

As a final remark, having an efficient expression compiler is not only beneficial to the integrated workflow. Indeed, there is no fundamental obstacle to developing a backend that generate HLS-supported C++ code from `arith` IR (for instance using the `hint` library). As such, it is perfectly possible to integrate the expression compiler in the specialization generator flow, by interfacing it with the “Architecture generation library” from figure 3.4. This will however not solve the issue of variable type identification. As such, the integrated flow is preferable when possible.

Conclusion and perspectives

The work presented on this thesis contributed to bring support for application-specific arithmetic operators to High Level Synthesis tools.

Part of this can be done using pure host language constructions (in this thesis, C++ was used as it is the language used by mainstream HLS tools), in a portable way. On this work, this has been illustrated in chapter 2 with the implementation of elementary operations for various floating-point formats. This allows application developers to write their code once and change the underlying numerical format with a simple `typedef`. This facilitates the choice of a format that provide a good accuracy vs performance trade-off on a per-application basis. In the future, this source-only approach can be extended to other operators, as long as they do not require complex pre-computations. This includes for instance as mixed-precision or fused operators.

For operators requiring complex pre-computations, the HLS support is still possible, but requires adaptation of the HLS compilation flow. This is shown in chapter 3 with operators that evaluate arbitrary mathematical function with fixed-point input and output format. Implementations of arbitrary function evaluators use function-specific pre-computed values such as approximation polynomial coefficients. Computing these values is complex enough that special tools have been developed in this purpose, such as FloPoCo or Sollya. Two methods have been proposed to use these tools to compute function-specific values before injecting the function evaluation algorithm in the standard HLS compilation flow. The first is based on a sequence of compilation stages, using unmodified off-the-shelf compilers, and is therefore portable. It uses a pre-compilation stage that produces an executable that invoke the existing tools to produce the HLS-enabled operator implementation. That implementation can then be fed to the HLS

compiler. This approach is however limited due to it being based on the C++ type system to extract the expression graph of the mathematical function to evaluate. The second method solves this issue by adding built-in support for these operators inside the HLS compiler.

While this breaks the portability of the solution, as it requires modifying the compiler, the greatest part of the work can be factored to be reused in different compilers by using a compiler-agnostic intermediate representation (IR) for the operators. The compiler should produce an IR describing the operator high-level specification. A compiler-independent tool, the expression compiler, can convert this specification to a low-level implementation, which can be imported back in the original compiler. This design has been tested using a custom HLS compiler based on `clang++` and Vitis HLS. The complexity of developing the intermediate representation for operators and the mechanism to lower it to implementation details was greatly reduced by the MLIR framework.

While the work presented in this thesis focuses in providing high level functionalities that were previously unsupported by HLS tools, the same technics could be used to port other hardware-specific optimizations to HLS tools. Part of the reason that operators generated by specialized tools such as FloPoCo have good latency and area comes from the clever usage of primitives such as bit heaps [65] or multiple constant multipliers [66]. In number of cases, it is possible to automatically identify the places where such primitives could be used beneficially at the abstraction level of fixed point operations in the IR. Having an IR representation for such operations could then be of interest to improve area and latency of library-generated operators. While this is a medium term perspective, a shorter-term goal is the support of additional functions evaluation architectures.

In a nutshell, supporting a greater part of existing state of the art constitutes a first progression axis for the proposed system. A second progression axis consists in offering higher level constructs. An example of this would be to allow the user to create a filter by frequency response specification. This is in phase with the objective of allowing non-specialized programmers to write programs with good productivity, by specifying what should be computed and delegating to the tool the choice on how to compute it. Embedded domain-specific languages in the C++ code seems to be a good interface for specifying such computations. Optimizations can be performed at each abstraction level view of the operator: from the high level function description (with algebraic optimizations), to the low level implementation (by using efficient fixed-point primitives for instance). Having an IR and associated optimizations for each abstraction level enables efficient implementation of these optimizations. Be-

sides, the optimizations on low-level implementation details are not specific to mathematical function evaluation, and can be reused in other contexts.

The current system has two main limitations:

1. Vendor HLS compilers backend are closed-source, and constitutes a black-box part of the design. It often requires retro-engineering to understand which IR construct will give good implementation result, and it makes the system very fragile to backend version change.
2. There is no cost model to estimate reliably the QoR of an implementation without having to perform the full synthesis, placing and routing steps.

The first limitation could be lifted by integrating with open-source HDL backend. The CIRCT project[67] is a collection of MLIR dialects representing digital circuits, that can be lowered to Verilog. Targeting these IR does not pose fundamental difficulties. While the ideal would be to have an open-source toolchain for end-to-end source-to-bitstream conversion (as the HDL can be considered to be another IR in this process), the HDL mapping to FPGA primitive is a bit more predictable than the source-to-HDL conversion realized by HLS backends. In the current state of the compiler however, only the operator codes go through the MLIR pipeline. The rest of the code is delegated to the HLS backend that will, among others, perform all the transformation required to get hardware that respects the call graph semantics of the original program. With the CIRCT based solution, this should also be done inside the MLIR pipeline. This is however not a fundamental issue, as proof of concepts of HLS compilers using the CIRCT framework have been developed [68].

Lifting the second limitation is more complicated. Indeed, it requires not only good knowledge of the hardware internal, but also of the synthesis and placement algorithms used to produce the FPGA bitstream out of the HDL description. This information is often considered sensitive by hardware vendors, and kept internally. Besides, experiments have shown that current vendor cost model does not perform well on small designs such as arithmetic operators. For instance, some logical optimizations are not taken into account by HLS cost models, which can result in local overestimates of latency of component parts, leading the pipelining system to insert more pipeline stages than what would really be required.

Improving these models is probably seen by vendors as too costly compared to the benefit they get by providing HLS solutions. This is however a chicken and egg problem, as the interest of HLS is limited by the QoR it can produce. Getting a better QoR by providing easy to use and to test libraries with state-of-the-art implementations is a good first step. The development in recent year of high quality compilation framework and hardware construc-

tion languages is of great help to achieve this first step.

A

Source code for floating-point adders

This annex reports the MArTo library code used to implement addition for IEEE-754 and posit. In this code, the `Wrapper` template template parameter specify the `hint` backend for which the operator is implemented. See section 2.1.3 page 48 for more details.

Listing 2.7 page 57 gives an example of user-written code that will trigger the instantiation of one of the adder presented here, depending on the actual type of `my_fp_type`.

A.1 IEEE-754 Adder

The IEEE adder uses the `IEEENumber`, which is a helper class built on `hint` wrappers to provides useful methods such as exponent or fraction field extraction. The code for this class is presented on listing A.1. Listing A.2 presents the complete adder algorithm.

Listing A.1: MArTo code for the `IEEENumber` class

```
1  template<unsigned int WE,
2           unsigned int WF,
3           template<unsigned int, bool> class Wrapper>
4  class IEEENumber : public Wrapper<WE + WF + 1, false>
5  {
6    private:
7      typedef Wrapper<WE+WF+1, false> basetype; // Underlying storage type
8      template<unsigned int W>
9      using us_wrapper = Wrapper<W, false>;
10   public:
11     static constexpr unsigned int _WE = WE;
12     static constexpr unsigned int _WF = WF;
13     using rounding_type_t = us_wrapper<3>;
14
```

```
15      IEEENumber(Wrapper<WE + WF + 1, false> const val = {0}):
16          Wrapper<WE+WF+1, false>{val}{}
17
18      inline us_wrapper<1> getSign() const{
19        return basetype::template get<WE+WF>();
20      }
21
22      inline us_wrapper<WE> getExponent() const {
23        return basetype::template slice<WF + WE - 1, WF>();
24      }
25
26      inline us_wrapper<WF> getFractionalPart() const {
27        return basetype::template slice<WF - 1, 0>();
28      }
29
30      inline us_wrapper<WF+WE> getExpFrac() const {
31        return basetype::template slice<WF+WE - 1, 0>();
32      }
33
34      inline us_wrapper<1> getLeadBitVal() const {
35        return getExponent().or_reduction();
36      }
37
38      inline us_wrapper<1> isInfinity() const {
39        return getExponent().and_reduction()
40                            .bitwise_and(
41                              getFractionalPart().nor_reduction()
42                            );
43      }
44
45      inline us_wrapper<1> isNaN() const {
46        return (getExponent().and_reduction()) &
47               (getFractionalPart().or_reduction());
48      }
49  };
```

Listing A.2: MArTo code for an IEEE-754 adder

```
template<
    unsigned int WE,
    unsigned int WF,
    template<unsigned int, bool> class Wrapper>
inline IEEENumber<WE, WF, Wrapper> ieee_add(
  IEEENumber<WE, WF, Wrapper> const in0,
  IEEENumber<WE, WF, Wrapper> const in1,
  IEEERoundingMode const roundingMode = IEEERoundingMode::RoundNearestTieEven
  )
{
  // getExponent etc are syntaxic sugar for call to slice() at the right positions
  auto exp0 = in0.getExponent();
  auto exp1 = in1.getExponent();

  auto sign0 = in0.getSign();
  auto sign1 = in1.getSign();

  auto frac0 = in0.getFractionalPart();
  auto frac1 = in1.getFractionalPart();

  auto exp0IsZero = exp0.nor_reduction();
  auto exp1IsZero = exp1.nor_reduction();
  auto exp0AllOne = exp0.and_reduction();
  auto exp1AllOne = exp1.and_reduction();
  auto frac0IsZero = frac0.nor_reduction();
  auto frac1IsZero = frac1.nor_reduction();
  auto exp0IsNotZero = exp0.or_reduction();
  auto exp1IsNotZero = exp1.or_reduction();

  auto expfrac0 = in0.getExpFrac();
  auto expfrac1 = in1.getExpFrac();

  auto diff0 = exp0.modularSub(exp1);
  auto diff1 = exp1.modularSub(exp0);

  auto effsub = sign0 ^ sign1;

  auto swap = expfrac1 > expfrac0;

  auto maxExp = Wrapper<WE, false>::mux(swap, exp1, exp0);
  auto minExp = Wrapper<WE, false>::mux(swap, exp0, exp1);
  auto expdiff = Wrapper<WE, false>::mux(swap, diff1, diff0);

  auto maxSign = Wrapper<1, false>::mux(swap, sign1, sign0);
  auto minSign = Wrapper<1, false>::mux(swap, sign0, sign1);

```

```
47  auto maxFrac = Wrapper<WF, false>::mux(swap, frac1, frac0);
48  auto minFrac = Wrapper<WF, false>::mux(swap, frac0, frac1);
49
50  // Special case detection
51  auto maxExpIsZero = Wrapper<1, false>::mux(swap, exp1IsZero, exp0IsZero);
52  auto minExpIsZero = Wrapper<1, false>::mux(swap, exp0IsZero, exp1IsZero);
53  auto maxExpAllOne = Wrapper<1, false>::mux(swap, exp1AllOne, exp0AllOne);
54  auto minExpAllOne = Wrapper<1, false>::mux(swap, exp0AllOne, exp1AllOne);
55  auto maxFracIsZero = Wrapper<1, false>::mux(swap, frac1IsZero, frac0IsZero);
56  auto minFracIsZero = Wrapper<1, false>::mux(swap, frac0IsZero, frac1IsZero);
57
58  //Special case logic
59  auto maxIsInfinity = maxExpAllOne & maxFracIsZero;
60  auto maxIsNaN = maxExpAllOne & maxFracIsZero.invert();
61  auto maxIsZero = maxExpIsZero & maxFracIsZero;
62  auto minIsInfinity = minExpAllOne & minFracIsZero;
63  auto minIsNaN = minExpAllOne & minFracIsZero.invert();
64  auto minIsZero = minExpIsZero & minFracIsZero;
65
66  auto bothSubNormals = maxExpIsZero;
67  auto maxIsNormal = Wrapper<1, false>::mux(swap, exp1IsNotZero, exp0IsNotZero);
68  auto minIsNormal = Wrapper<1, false>::mux(swap, exp0IsNotZero, exp1IsNotZero);
69
70  auto infinitySub = exp0AllOne.concatenate(exp1AllOne)
71                               .concatenate(frac0IsZero)
72                               .concatenate(frac1IsZero)
73                               .concatenate(effsub)
74                               .and_reduction();
75  auto resultIsNan = maxIsNaN.concatenate(minIsNaN)
76                             .concatenate(infinitySub)
77                             .or_reduction();
78  auto onlyOneSubnormal = minExpIsZero & maxExpIsZero.invert();
79  auto explicitedMaxFrac = maxIsNormal.concatenate(maxFrac);
80  auto explicitedMinFrac = minIsNormal.concatenate(minFrac);
81
82  //alignment
83  auto maxShiftVal = Wrapper<WE, false>{WF+3};
84  auto allShiftedOut = expdiff > maxShiftVal;
85
86  auto shiftValue = Wrapper<WE, false>::mux(
87          allShiftedOut,
88          maxShiftVal,
89          expdiff
90        ).modularSub(onlyOneSubnormal.template leftpad<WE>());
91
92  Wrapper<WF+3, false> extendedMinFrac = explicitedMinFrac.concatenate(
93                                            Wrapper<2, false>{0});
94
95  auto shiftedMinFracSticky = shifter_sticky(extendedMinFrac, shiftValue);
96  auto beforeComp = Wrapper<1, false>{0}.concatenate(shiftedMinFracSticky
```

```
                                        .template slice<WF + 3, 1>());

  auto shiftedMinFrac = beforeComp ^
                        Wrapper<WF+4, false>::generateSequence(effsub);

  auto stickyMinFrac = shiftedMinFracSticky.template get<0>();

  // Addition
  auto carryIn = effsub & stickyMinFrac.invert();
  auto extendedMaxFrac = explicitedMaxFrac.concatenate(Wrapper<1, false>{0})
                                          .concatenate(carryIn)
                                          .template leftpad<WF+4>();

  auto signifcandResult = extendedMaxFrac + shiftedMinFrac;

  // Renormalization
  auto isNeg = signifcandResult.template get<WF + 4>();
  auto z1 = signifcandResult.template get<WF+3>();
  auto z0 = signifcandResult.template get<WF+2>();

  auto lzcInput = signifcandResult.template slice<WF+3, 1>();
  auto lzc = lzoc_wrapper(lzcInput, {0});

  constexpr unsigned int lzcsize = hint::Static_Val<WF+3>::_storage;

  static_assert (lzcsize<=WE,
    "The adder works only for wE > log2(WF).\n"
    "Are you sure you need subnormals ?\n"
    "If yes, contact us, we will be happy to make it work for you.");

  auto subnormalLimitVal = Wrapper<lzcsize, false>{WF+3};

  auto maxExpIsOne = (maxExp == Wrapper<WE, false>{1});

  auto lzcGreaterEqExp = (lzc.template leftpad<WE>() >= maxExp);
  auto lzcSmallerEqExp = (lzc.template leftpad<WE>() <= maxExp);
  auto lzcSmallerMaxVal = lzcInput.or_reduction();
  auto fullCancellation = lzcInput.nor_reduction();

  auto normalOverflow = z1;
  auto lzcOne = z1.invert() & z0;
  auto subnormalOverflow = lzcOne & maxExpIsZero;
  auto cancellation = z1.invert() & z0.invert();

  auto overflow = normalOverflow | subnormalOverflow;

  auto isLeftShiftLZC = overflow |
      (lzcSmallerMaxVal.invert() & bothSubNormals) |
      (cancellation & maxIsNormal & lzcSmallerEqExp) |
      (maxIsNormal & lzcSmallerMaxVal.invert());
```

```
147  auto isLeftShiftExp = lzcSmallerMaxVal & lzcGreaterEqExp & maxIsNormal;
148
149  auto shiftFirstStage = Wrapper<lzcsize, false>::mux(
150      isLeftShiftLZC,
151      lzc,
152      Wrapper<lzcsize, false>{1}
153    );
154  auto normalisationShiftVal = Wrapper<lzcsize, false>::mux(
155      isLeftShiftExp,
156      maxExp.template slice<lzcsize-1, 0>(),
157      shiftFirstStage
158    );
159
160  auto normalisedSignif = signifcandResult << normalisationShiftVal;
161  auto significandPreRound = normalisedSignif.template slice<WF+2, 3>();
162  auto lsb = normalisedSignif.template get<3>();
163  auto roundBit = normalisedSignif.template get<2>();
164  auto sticky = stickyMinFrac |
165                normalisedSignif.template slice <1, 0>()
166                                .or_reduction();
167
168  auto deltaExpIsZero = ~z1 & (z0 ^ bothSubNormals);
169  auto deltaExpIsMinusOne = z1 | (z0 & bothSubNormals);
170  auto deltaExpIsLZC = ~(z1 | z0 | bothSubNormals) &
171                       lzcSmallerEqExp &
172                       lzcSmallerMaxVal;
173
174  auto deltaExpExp = ~(deltaExpIsLZC | deltaExpIsZero | deltaExpIsMinusOne);
175
176  auto deltaExpCin = deltaExpExp | deltaExpIsMinusOne | deltaExpIsLZC;
177  auto deltaBigPartIsZero = deltaExpIsZero | deltaExpIsMinusOne;
178
179  auto deltaExpUnmasked = Wrapper<WE, false>::mux(
180        deltaExpIsLZC,
181        ~((lzc.modularSub(Wrapper<lzcsize, false>{1})).template leftpad<WE>()),
182        ~maxExp
183      );
184
185  auto maskSequence = Wrapper<WE, false>::generateSequence(~deltaBigPartIsZero);
186  auto deltaExpBeforeCorrection = deltaExpUnmasked & maskSequence;
187
188  auto expPreRound = maxExp.addWithCarry(deltaExpBeforeCorrection, deltaExpCin)
189                           .template slice<WE-1, 0>();
190  auto expSigPreRound = expPreRound.concatenate(significandPreRound);
191
192  auto roundUpBit = ieee_getRoundBit(
193                      maxSign,
194                      lsb,
195                      roundBit,
196                      sticky,
```

```
                      roundingMode);

  auto unroundedInf = expPreRound.and_reduction();

    auto roundingCode =
        Wrapper<8, false>{static_cast<uint8_t>(roundingMode)}
            .template slice<2, 0>();
    auto b0 = roundingCode.template get<0>();
  auto b1 = roundingCode.template get<1>();
  auto b2 = roundingCode.template get<2>();
  auto forbidden_inf = ((b2 & b0 & (b1 == maxSign)) |
  ↪  ~(roundingCode.or_reduction())) &
                       unroundedInf &
                       ~maxIsInfinity &
                       ~resultIsNan;

  auto expSigRounded = expSigPreRound.modularAdd(
                        roundUpBit.template leftpad<WE+WF>());
  auto finalExp = expSigRounded.template slice<WF+WE-1, WF>();

  auto resultIsZero = ~(fullCancellation & finalExp.or_reduction());
  auto resultIsInf = ~resultIsNan & (
      (maxIsInfinity & minIsInfinity & ~effsub) |
      (maxIsInfinity ^ minIsInfinity) |
      finalExp.and_reduction()
    );

  auto constInfNanExp = Wrapper<WE-1, false>::generateSequence({1})
                            .concatenate(resultIsNan | ~forbidden_inf);
  auto constInfNanSignif = Wrapper<WF, false>::generateSequence(
                                resultIsNan | forbidden_inf);

  auto constInfNan = constInfNanExp.concatenate(constInfNanSignif);
  auto finalRes = Wrapper<WE+WF, false>::mux(
         resultIsNan | resultIsInf,
         constInfNan,
         expSigRounded);

  auto bothZeros = maxIsZero & minIsZero;
  auto signBothZero = minSign & maxSign;

  Wrapper<1, false> isRoundDown{roundingMode == IEEERoundingMode::RoundDown};
  auto negZeroOp = resultIsZero & isRoundDown & (effsub | ~bothZeros);
  auto signR = ~resultIsNan & // NaN forces sign to be zero
      ((resultIsZero & (signBothZero | negZeroOp)) | // Special case (+0) + (-0)
      (~resultIsZero & maxSign));

  return {signR.concatenate(finalRes)};
}
```

A.2 Posit adder

In the posit case, two helper classes exists that play the same role as `IEEENumber`. The `PositEncoding` provide useful methods related to posit encodings. The `PositIntermediateFormat` is useful to manipulate posit intermediate format (PIF) values (see section 2.2.3 for more information). These class are not reported here as they are very similar to `IEEENumber`. Before performing the addition, posit encodings are converted to PIF using a posit decoder (presented on listing A.3). Then the two PIFs can be added, using the adder code from listing A.5. Finally, the result has to be rounded back using a posit encoder, presented on listing A.4.

Listing A.3: MArTo code for the posit to pif decoder

```
template<
  unsigned int N,
  unsigned int WES,
  template<unsigned int, bool> class Wrapper>
inline Wrapper<PositDim<N, WES>::WE, true> getExponent(
    Wrapper<hint::Static_Val<N-2>::_storage + 1, false> range_count,
    Wrapper<N-3, false> shifted_fraction,
    Wrapper<1, false> sign,
    typename enable_if<PositDim<N, WES>::HAS_ES>::type* = 0
  )
{
  auto es = shifted_fraction.template slice<N - 4, N-3-WES>();
  auto ext_sign = Wrapper<WES, false>::generateSequence(sign);
  auto decoded_es = es.bitwise_xor(ext_sign);
  return range_count.concatenate(decoded_es).as_signed();
}

template<
  unsigned int N,
  unsigned int WES,
  template<unsigned int, bool> class Wrapper>
inline Wrapper<PositDim<N, WES>::WE, true> getExponent(
    Wrapper<hint::Static_Val<N-2>::_storage + 1, false> range_count,
    Wrapper<N-3, false>,
    Wrapper<1, false>,
    typename enable_if<not PositDim<N, WES>::HAS_ES>::type* = 0
  )
{
  return range_count.as_signed();
}

template<
  unsigned int N,
  unsigned int WES,
```

```
  template<unsigned int, bool> class Wrapper>
inline PositIntermediateFormat<N, WES, Wrapper, true>
posit_decoder(PositEncoding<N, WES, Wrapper> positN)
{
  //Sign bit
  auto s = positN.template get<N-1>();
  //First regime bit
  auto count_type = positN.template get<N-2>();
  //Remainder
  auto input_shift = positN.template slice<N-3, 0>();

  auto zero_NAR = input_shift.or_reduction().bitwise_or(count_type).invert();
  auto is_NAR = zero_NAR.bitwise_and(s);
  auto is_zero = zero_NAR.bitwise_and(s.invert());

  auto implicit_bit = s.invert().bitwise_and(zero_NAR.invert());
  constexpr int rangeCountSize = hint::Static_Val<N-2>::_storage;
  auto lzoc_shifted = hint::LZOC_shift<N-2, N-2>(input_shift, count_type);

  auto rangeCount = lzoc_shifted.lzoc.template leftpad<rangeCountSize+1>();
  auto usefulBits = lzoc_shifted.shifted.template slice<N-4, 0>();
  auto fraction = usefulBits.template slice<N-4-WES, 0>();

  auto neg_count = s.bitwise_xor(count_type).invert();
  auto extended_neg_count = Wrapper<rangeCountSize + 1,
  ↪  false>::generateSequence(neg_count);
  auto comp2_range_count = rangeCount.bitwise_xor(extended_neg_count);

  auto exponent = getExponent<N, WES>(
        comp2_range_count,
        usefulBits,
        s
      );

  return PositIntermediateFormat<N, WES, Wrapper, true>(
        is_NAR,
        final_biased_exp,
        s,
        implicit_bit,
        fraction
      );
}
```

Listing A.4: MArTo code for the PIF to posit encoder

```
template<
  unsigned int N,
  unsigned int WES,
  template<unsigned int, bool> class Wrapper>
inline Wrapper<PositDim<N, WES>::WF + WES, false> buildEsSignifSequence(
    Wrapper<1, false> sign,
    Wrapper<PositDim<N, WES>::WF, false> significand,
    Wrapper<PositDim<N, WES>::WE, false> exponent,
    typename enable_if<PositDim<N, WES>::HAS_ES>::type* = 0
  )
{
  auto sign_sequence_wes = Wrapper<WES, false>::generateSequence(sign);

  auto es_wo_xor = exponent.template slice<WES-1, 0>();
  auto es = es_wo_xor.bitwise_xor(sign_sequence_wes);
  auto ret = es.concatenate(significand);
  return ret;
}

template<
  unsigned int N,
  unsigned int WES,
  template<unsigned int, bool> class Wrapper>
inline Wrapper<PositDim<N, WES>::WF, false> buildEsSignifSequence(
    Wrapper<1, false>,
    Wrapper<PositDim<N, WES>::WF, false> significand,
    Wrapper<PositDim<N, WES>::WE, false>,
    typename enable_if<not PositDim<N, WES>::HAS_ES>::type* = 0
  )
{
  return significand;
}

template<
  unsigned int N,
  unsigned int WES,
  template<unsigned int, bool> class Wrapper>
inline Wrapper<1, false> exp_overflow(
      Wrapper<PositDim<N, WES>::WE, false>,
      typename enable_if<PositDim<N, WES>::HAS_ES>::type* = 0
    )
{
    return {0};
}

template<
```

```
47    unsigned int N,
48    unsigned int WES,
49    template<unsigned int, bool> class Wrapper>
50  inline Wrapper<1, false> exp_overflow(
51        Wrapper<PositDim<N, WES>::WE, false> exp,
52        typename enable_if<not PositDim<N, WES>::HAS_ES>::type* = 0
53      )
54  {
55    constexpr auto WE = PositDim<N, WES>::WE;
56    Wrapper<WE, false> emax{PositDim<N, WES>::EMax};
57    auto biggerEmax = (exp.as_signed() >= emax.as_signed());
58    auto res = biggerEmax;
59    return res;
60  }
61
62  template<
63    unsigned int N,
64    unsigned int WES,
65    template<unsigned int, bool> class Wrapper>
66  inline PositEncoding<N, WES, Wrapper>
67  posit_encoder(PositIntermediateFormat<N, WES, Wrapper, false> positValue) {
68    constexpr auto S_WF = PositDim<N, WES>::WF;
69    constexpr auto S_WE = PositDim<N, WES>::WE;
70    constexpr auto S_WES = WES;
71    constexpr auto K_SIZE = S_WE - S_WES;
72
73    auto exp = positValue.getExp();
74    auto sign = positValue.getSignBit();
75    auto significand = positValue.getFraction();
76
77    //K_SIZE
78    auto k = exp.template slice<S_WE-1, S_WES>();
79
80    // N-3
81    auto esAndSignificand = buildEsSignifSequence<N, WES, Wrapper>(
82      sign,
83      significand,
84      exp);
85
86    Wrapper<2, false> zero_one{1};
87    Wrapper<2, false> one_zero{2};
88
89    auto isNegative = k.template get<K_SIZE-1>() ^ sign;
90    auto leading = Wrapper<2, false>::mux(isNegative, zero_one, one_zero);
91
92    //N-1
93    auto reverseBitAndEsAndSignificand = leading.concatenate(esAndSignificand);
94
95    // K_SIZE - 1
96    auto low_k = k.template slice<K_SIZE-2, 0>();
```

```
  auto absK = Wrapper<K_SIZE-1, false>::mux(
        k.template get<K_SIZE-1>(),
        low_k.invert(),
        low_k
      );
  //N
  auto rBESSignif = reverseBitAndEsAndSignificand.concatenate(
      positValue.getGuardBit());



  //N+1
  auto shifted = shifter_sticky(
        rBESSignif,
        absK,
        rBESSignif.template get<N-1>()
    ); //TODO rajouter le fillbit

  //N-1
  auto unroundedResult =  shifted.template slice<N, 2>();

  auto guard = shifted.template get<1>();
  auto sticky = shifted.template get<0>().bitwise_or(
      positValue.getStickyBit());

  auto roundOverflow = exp_overflow<N, WES, Wrapper>(exp);
  auto forbidRound = ~isNegative & roundOverflow;
  auto forceRound = isNegative & roundOverflow;
  auto roundingBit = (forceRound | guard &
                        (sticky | unroundedResult.template get<0>())
                     ) & ~forbidRound;

  auto roundedResult = unroundedResult.modularAdd(roundingBit.template
  ↪  leftpad<N-1>());

  auto normalOutput = sign.concatenate(roundedResult);
  auto zero = Wrapper<N-1, false>::generateSequence({0});
  auto isNaRBit = positValue.getIsNaR();
  auto specialCasesValue = isNaRBit.concatenate(zero);

  auto isSpecial = (~positValue.getSignBit() & ~(positValue.getImplicitBit())) |
                   isNaRBit;

  return Wrapper<N, false>::mux(isSpecial, specialCasesValue, normalOutput);
}
```

Listing A.5: MArTo code for a posit adder

```
template<
```

```
  unsigned int N,
  unsigned int WES,
  template<unsigned int, bool> class Wrapper
>
inline PositIntermediateFormat<N, WES, Wrapper, false> posit_add(
    PositIntermediateFormat<N, WES, Wrapper, true> in1,
    PositIntermediateFormat<N, WES, Wrapper, true> in2
){
  constexpr auto S_WF = PositDim<N, WES>::WF;
  constexpr auto S_WE = PositDim<N, WES>::WE;
  constexpr auto S_WES = WES;
  constexpr auto K_SIZE = S_WE - S_WES;

  //Sort in order to have exponent of in1 greater than exponent of in2
  auto in2IsZero = in2.isZero();
  auto in1IsZero = in1.isZero();
  auto oneIsZero = in1IsZero | in2IsZero;

  auto input2Significand = in2.getSignedSignificand();
  auto input1Significand = in1.getSignedSignificand();

  auto exp1 = in1.getExp().as_signed();
  auto exp2 = in2.getExp().as_signed();

  auto in1IsGreater = (~in1IsZero & (exp1 > exp2)) | in2IsZero;

  auto subExpOp1 = Wrapper<S_WE, true>::mux(in1IsGreater, exp1, exp2);
  auto subExpOp2 = Wrapper<S_WE, true>::mux(in1IsGreater, exp2, exp1);
  auto mostSignificantSignificand = Wrapper<S_WF+2, false>::mux(
    in1IsGreater,
    input1Significand,
    input2Significand
  );

  auto lessSignificantSignificand = Wrapper<S_WF+2, false>::mux(
    in1IsGreater,
    input2Significand,
    input1Significand
  );

  auto mostSignifSign = Wrapper<1, false>::mux(
    in1IsGreater,
    in1.getSignBit(),
    in2.getSignBit()
  );

  auto lessSignifSign = Wrapper<1, false>::mux(
    in1IsGreater,
    in2.getSignBit(),
    in1.getSignBit()
```

```
52  );
53
54  // Relative shift of exponents
55
56  auto shiftValue = subExpOp1.modularSub(subExpOp2).as_unsigned();
57  auto shiftedSignificand = shifter_sticky(
58        lessSignificantSignificand.concatenate(Wrapper<2, false>{0}),
59        shiftValue,
60        lessSignifSign
61      );
62
63  auto shiftedTop = shiftedSignificand.template slice<S_WF+2+2, 3>();
64  auto guards = shiftedSignificand.template slice<2, 1>();
65  auto sticky_low = shiftedSignificand.template get<0>();
66
67  auto addOp1 = mostSignifSign.concatenate(mostSignificantSignificand);
68  auto addOp2 = lessSignifSign.concatenate(shiftedTop);
69
70  auto addRes = addOp1.modularAdd(addOp2);
71  auto toCount = addRes.template get<S_WF+2>();
72  auto usefulRes = addRes.template slice<S_WF+1, 0>();
73
74  auto lzoc_shifted = LZOC_shift<S_WF+4, S_WF+4>(usefulRes.concatenate(guards),
    ↪  toCount);
75
76
77  auto & lzoc = lzoc_shifted.lzoc;
78  auto & shifted = lzoc_shifted.shifted;
79  auto frac = shifted.template slice<S_WF+3, 3>();
80  auto round = shifted.template get<2>();
81  auto sticky = sticky_low.bitwise_or(
82      shifted.template get<1>()
83             .bitwise_or(shifted.template get<0>()));
84
85  auto final_exp = subExpOp1.subWithCarry(lzoc.template
    ↪  leftpad<S_WE>().as_signed(), {1})
86                            .template slice<S_WE-1, 0>();
87
88  auto isResultNar = in1.getIsNaR().bitwise_or(in2.getIsNaR());
89
90  PositIntermediateFormat<N, WES, Wrapper, false> result {
91        round,
92        sticky,
93        isResultNar,
94        final_exp,
95        toCount,
96        frac.template get<S_WF>(),
97        frac.template slice<S_WF-1, 0>()
98  };
99  return result;
```

```
100 }
```

Bibliography

[1] J. Hormigo and J. Villalba, "New formats for computing with real-numbers under round-to-nearest," *IEEE Transactions on Computers*, vol. 65, no. 7, pp. 2158–2168, 2016, URL: https://doi.org/10.1109/TC.2015.2479623 (cit. on p. 12).

[2] T. C. Chen and I. T. Ho, "Storage-efficient representation of decimal data," *Commun. ACM*, vol. 18, no. 1, pp. 49–52, Jan. 1975, ISSN: 0001-0782, URL: https://doi.org/10.1145/360569.360660 (cit. on p. 14).

[3] "ISO/IEC/IEEE International Standard - Floating-point arithmetic," IEEE, Tech. Rep. 754-2019, 2020, URL: https://doi.org/10.1109/IEEESTD.2020.9091348 (cit. on p. 15).

[4] Y. Suzuki, "Building modern javascript engine," in *Proceedings of the 53rd IPSJ programming symposium*, vol. 2012, Jan. 2012, pp. 171–176 (cit. on p. 21).

[5] A. Agrawal, S. M. Mueller, B. M. Fleischer, *et al.*, "DLFloat: A 16-b floating point format designed for deep learning training and inference," in *IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 2019, pp. 92–95, URL: https://doi.org/10.1109/ARITH.2019.00023 (cit. on p. 21).

[6] "BFloat16 – hardware numerics definition," Intel, Tech. Rep., Nov. 2018, URL: https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf (visited on 02/17/2023) (cit. on p. 22).

[7] E. Chung, J. Fowers, K. Ovtcharov, *et al.*, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018, URL: https://doi.org/10.1109/MM.2018.022071131 (cit. on p. 22).

[8] "Standard for Posit arithmetic," Posit Working Group, Tech. Rep., Mar. 2022, URL: https://posithub.org/docs/posit_standard-2.pdf (visited on 02/20/2023) (cit. on pp. 23, 70).

[9] "Posit standard documentation," Posit Working Group, Tech. Rep. Release 3.2 Draft, Jun. 2018, URL: https://posithub.org/docs/posit_standard.pdf (visited on 02/20/2023) (cit. on pp. 23, 70).

[10] E. Swartzlander and A. Alexopoulos, "The sign/logarithm number system," *IEEE Transactions on Computers*, vol. C-24, no. 12, pp. 1238–1242, 1975, URL: https://doi.org/10.1109/T-C.1975.224172 (cit. on p. 26).

[11] N. G. Kingsbury and P. J. Rayner, "Digital filtering using logarithmic arithmetic," *Electronics Letters*, vol. 2, no. 7, pp. 56–58, 1971 (cit. on p. 26).

[12] "Ultrascale architecture Configurable Logic Block," Xilinx, Tech. Rep., Feb. 2017, URL: https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb (visited on 03/09/2023) (cit. on pp. 28, 32).

[13] "Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules user guide," Intel, Tech. Rep., Mar. 2022, URL: https://cdrdv2.intel.com/v1/dl/getContent/666917?explicitVersion=true (visited on 03/09/2023) (cit. on p. 29).

[14] "Ultrascale architecture dsp slice," Xilinx, Tech. Rep., Aug. 2021, URL: https://www.xilinx.com/content/dam/xilinx/support/documents/user_guides/ug579-ultrascale-dsp.pdf (visited on 03/13/2023) (cit. on p. 32).

[15] A. M. Caulfield, E. S. Chung, A. Putnam, *et al.*, "A cloud-scale acceleration architecture," in *49th Annual International Symposium on Microarchitecture*, 2016, pp. 1–13, URL: https://doi.org/10.1109/MICRO.2016.7783710 (cit. on p. 32).

[16] A. Sarkar and S. Banerjee, "Fpga implementation of dna sequence alignment with traceback," in *4th International Conference on Electronics, Communication and Aerospace Technology*, 2020, pp. 47–52, URL: https://doi.org/10.1109/ICECA49313.2020.9297554 (cit. on p. 32).

[17] "Verilog hardware description language," IEEE, Tech. Rep. 1364-2005, 2006, URL: https://doi.org/10.1109/IEEESTD.2006.99495 (cit. on p. 34).

[18] "VHDL language reference manual," IEEE, Tech. Rep. 1076-2019, 2019, URL: https://doi.org/10.1109/IEEESTD.2019.8938196 (cit. on p. 34).

[19] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225, ISBN: 9781450311991, URL: https://doi.org/10.1145/2228360.2228584 (cit. on p. 36).

[20] "Vitis High-Level Synthesis user guide," AMD Xilinx, Tech. Rep. UG1399-2022.2, Dec. 2022, URL: https://docs.xilinx.com/viewer/book-attachment/NsrqATHzUj6if4Toia~ORQ/eysSTISAO7ZIMF3n0HIRrQ (visited on 03/21/2023) (cit. on p. 37).

[21] F. Ferrandi, V. G. Castellana, S. Curzel, *et al.*, "Invited: Bambu: An open-source research framework for the high-level synthesis of complex applications," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, IEEE, Dec. 2021, pp. 1327–1330, URL: https://doi.org/10.1109/DAC18074.2021.9586110 (cit. on p. 37).

[22] "SYCL 2020 specification," Khronos SYCL working group, Tech. Rep. revision 6, 2022, URL: https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf (visited on 03/27/2023) (cit. on pp. 37, 86).

[23] M. Kinsner, "Sycl intel dataflow pipes," Tech. Rep. revision 4, Dec. 2, 2021, URL: https://github.com/intel/llvm/blob/2022-06/sycl/doc/extensions/supported/sycl_ext_intel_dataflow_pipes.asciidoc (visited on 03/27/2023) (cit. on p. 37).

[24] A. Ballman, M. Blower, T. Hoffner, and E. Keane, "Adding a fundamental type for n-bit integers," ISO/IEC JTC1 SC22 WG14 committee, Tech. Rep. N2709, Apr. 23, 2021, URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2709.pdf (visited on 03/20/2023) (cit. on p. 39).

[25] F. de Dinechin, "Reflections on 10 years of FloPoCo," in *26th IEEE Symposium of Computer Arithmetic (ARITH-26)*, Jun. 2019 (cit. on pp. 40, 60, 93).

[26] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the hardware cost of the posit number system," in *29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 106–113, URL: https://doi.org/10.1109/FPL.2019.00026 (cit. on pp. 43, 57).

[27] L. Forget, Y. Uguen, F. de Dinechin, and D. Thomas, "A type-safe arbitrary precision arithmetic portability layer for HLS tools," in *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, HEART 2019, Nagasaki, Japan, June 6-7, 2019*, ACM, 2019, 5:1–5:6, URL: https://doi.org/10.1145/3337801.3337809 (cit. on p. 43).

[28] A. Takach, *Algorithm c (AC) datatypes*, https://github.com/hlslibs/ac_types, 2018 (cit. on p. 44).

[29] A. Prost-Boucle, O. Muller, and F. Rousseau, "Fast and standalone design space exploration for high-level synthesis under resource constraints," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 79–93, 2014 (cit. on p. 44).

[30] J. O. Coplien, "Curiously recurring template patterns," *C++ Report*, vol. 7, no. 2, pp. 24–27, 1995 (cit. on p. 51).

[31] U. Kulisch, *Computer arithmetic and validity: theory, implementation, and applications*. Walter de Gruyter, 2013, vol. 33 (cit. on p. 58).

[32] F. de Dinechin and M. Kumm, *Application-Specific Arithmetic*. Springer, 2023, To appear (cit. on pp. 60, 93).

[33] A. Podobas and S. Matsuoka, "Hardware implementation of POSITs and their application in FPGAs," in *International Parallel and Distributed Processing Symposium*, IEEE, 2018, pp. 138–145 (cit. on pp. 62, 66, 76).

[34] R. Chaurasiya, J. Gustafson, R. Shrestha, *et al.*, "Parameterized posit arithmetic hardware generator," in *36th International Conference on Computer Design (ICCD)*, IEEE, 2018, pp. 334–341 (cit. on pp. 62, 66, 76, 78).

[35] M. K. Jaiswal and H. K.-H. So, "PACoGen: A hardware posit arithmetic core generator," *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019 (cit. on pp. 62, 77).

[36] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2004 (cit. on pp. 63, 64).

[37] J.-M. Muller, N. Brunie, F. de Dinechin, *et al.*, *Handbook of Floating-Point Arithmetic, 2nd edition*, Anglais. Birkhauser Boston, 2018, ISBN: 978-3319765259 (cit. on pp. 63, 64, 66).

[38] H. Zhang and S. Ko, "Design of power efficient posit multiplier," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 861–865, 2020 (cit. on p. 66).

[39] N. Brunie, "Modified Fused Multiply and Add for exact low precision product accumulation," in *24th Symposium on Computer Arithmetic (ARITH-24)*, IEEE, Jul. 2017 (cit. on pp. 66, 70, 82).

[40] J. Johnson, "Rethinking floating point for deep learning," arXiv, 1811.01721, 2018 (cit. on p. 66).

[41] F. de Dinechin, B. Pasca, O. Creţ, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *Field-Programmable Technologies*, IEEE, 2008, pp. 33–40 (cit. on pp. 66, 71).

[42] Y. Uguen, F. de Dinechin, V. Lezaud, and S. Derrien, "Application-specific arithmetic in high-level synthesis tools," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 1, 2020 (cit. on pp. 66, 82).

[43] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Design Automation Conference*, ACM, 2012 (cit. on p. 66).

[44] U. W. Kulisch, *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, 2002, ISBN: 3211838708 (cit. on p. 70).

[45] Y. Uguen and F. de Dinechin, "Design-space exploration for the Kulisch accumulator," working paper or preprint, Mar. 2017, URL: https://hal.archives-ouvertes.fr/hal-01488916 (cit. on pp. 70, 71, 82).

[46] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Performance-efficiency trade-off of low-precision numerical formats in deep neural networks," in *Next Generation Arithmetic*, ACM, 2019, 3:1–3:9, URL: https://doi.org/10.1145/3316279.3316282 (cit. on p. 73).

[47] P. Lindstrom, S. Lloyd, and J. Hittinger, "Universal coding of the reals: Alternatives to IEEE floating point," in *Next Generation Arithmetic*, ACM, 2018 (cit. on p. 73).

[48] F. De Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: The good, the bad and the ugly," in *Next Generation Arithmetic*, ACM, 2019 (cit. on p. 73).

[49] N. Buoncristiani, S. Shah, D. Donofrio, and J. Shalf, "Evaluating the numerical stability of posit arithmetic," in *International Parallel and Distributed Processing Symposium*, IEEE, 2020, pp. 612–621 (cit. on p. 73).

[50] F. Xiao, F. Liang, B. Wu, J. Liang, S. Cheng, and G. Zhang, "Posit arithmetic hardware implementations with the minimum cost divider and square root," *Electronics*, vol. 9, no. 10, 2020, ISSN: 2079-9292 (cit. on p. 77).

[51] 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, 2007 (cit. on p. 82).

[52] L. Forget, G. Harnisch, R. Keryell, and F. de Dinechin, "A single-source C++20 HLS flow for function evaluation on FPGA and beyond," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, ACM, 2022, pp. 51–58, URL: https://doi.org/10.1145/3535044.3535051 (cit. on pp. 85, 97).

[53] "SYCL for vitis: Experimental fusion of triSYCL with intel SYCL oneAPI DPC++ upstreaming effort into clang/llvm," AMD. (2022), URL: https://github.com/triSYCL/sycl (cit. on p. 86).

[54] "OneAPI DPC++," Intel. (2022), URL: https://github.com/intel/llvm (cit. on p. 86).

[55] "The llvm compiler infrastructure." (2022), URL: http://llvm.org (cit. on p. 86).

[56] D. Das Sarma and D. Matula, "Faithful bipartite rom reciprocal tables," in *12th Symposium on Computer Arithmetic*, ACM, 1995, URL: https://doi.org/10.1109/ARITH.1995.465381 (cit. on p. 92).

[57] D. Sunderland, R. Strauch, S. Wharfield, H. Peterson, and C. Cole, "Cmos/sos frequency synthesizer lsi circuit for spread spectrum communications," *IEEE Journal of Solid-State Circuits*, vol. 19, no. 4, 1984, URL: https://doi.org/10.1109/JSSC.1984.1052173 (cit. on p. 92).

[58] S.-F. Hsiao, P.-H. Wu, C.-S. Wen, and P. K. Meher, "Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 5, 2015, URL: https://doi.org/10.1109/TCSII.2014.2386232 (cit. on pp. 92, 107).

[59] M. Christ, L. Forget, and F. de Dinechin, “Lossless differential table compression for hardware function evaluation,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1642–1646, 2021 (cit. on p. 92).

[60] S. Chevillard, M. Joldeş, and C. Lauter, “Sollya: An environment for the development of numerical codes,” in *Mathematical Software - ICMS 2010*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., ser. Lecture Notes in Computer Science, vol. 6327, Heidelberg, Germany: Springer, Sep. 2010, pp. 28–31 (cit. on p. 93).

[61] A. L. M. Douglas, *The electrical production of music*. New York: Philosophical Library, 1957 (cit. on p. 98).

[62] F. de Dinechin, M. Istoan, and G. Sergent, “Fixed-point trigonometric functions on FPGAs,” *SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 83–88, 2013 (cit. on p. 99).

[63] C. Lattner, M. Amini, U. Bondhugula, *et al.*, “MLIR: Scaling compiler infrastructure for domain specific computation,” in *International Symposium on Code Generation and Optimization*, IEEE, 2021, pp. 2–14, URL: https://doi.org/10.1109/CGO51591.2021.9370308 (cit. on p. 103).

[64] F. de Dinechin, M. Joldes, and B. Pasca, “Automatic generation of polynomial-based hardware architectures for function evaluation,” in *Application-specific Systems, Architectures and Processors*, IEEE, 2010 (cit. on p. 107).

[65] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, “Arithmetic core generation using bit heaps,” in *Field-Programmable Logic and Applications*, Sep. 2013 (cit. on pp. 107, 110).

[66] M. Kumm and P. Zipf, “Hybrid Multiple Constant Multiplication for FPGAs,” in *IEEE International Conference on Electronics, Circuits and Systems, (ICECS)*, 2012, pp. 556–559 (cit. on p. 110).

[67] “CIRCT – circuit IR compilers and tools.” (2021), URL: https://github.com/llvm/circtl (cit. on p. 111).

[68] H. Ye, C. Hao, J. Cheng, *et al.*, “Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation,” in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 741–755, URL: https://doi.org/10.1109/HPCA53966.2022.00060 (cit. on p. 111).

UNIVERSITÉ DE LYON

INSA

FOLIO ADMINISTRATIF

THÈSE DE L'INSA LYON, MEMBRE DE L'UNIVERSITÉ DE LYON

NOM : FORGET — *Date de soutenance* : 29/06/2023
Prénoms : Luc

Titre : Description and compilation of ad-hoc arithmetic operators in the context of High-Level Synthesis

Nature : Doctorat — *Numéro d'ordre* : 2023ISAL0046

École doctorale : Informatique & Mathématique (InfoMath)

Spécialité : Informatique

Résumé :
Les techniques de synthèse de haut niveau permettent aux programmeurs non spécialistes de générer des descriptions de circuits numériques en utilisant des langages de programmation généralistes. Cependant, les outils existants ne supportent qu'un petit nombre de formats numériques et un petit nombre d'opérateurs standards. Cette thèse présente plusieurs techniques pour rajouter le support de nouveaux formats et de nouveaux opérateurs. Dans un premier temps, l'étude se focalise sur ce qui est réalisable en se restreignant aux fonctionnalités de métaprogrammation du standard C++ supporté par les outils HLS. Une bibliothèque d'opérateurs élémentaires pour les formats IEEE-754 et posit de taille arbitraire est proposée. Elle sert de base à une étude de cas comparant le coût matériel de l'implémentation de ces deux formats. L'implémentation d'évaluateurs de fonctions mathématiques arbitraires se heurte aux limites de la première approche. Dans un second temps, l'étude se porte sur les possibilités offertes par la modification du flot de compilation HLS, avec comme objectif de supporter cette fonctionnalité. Une bibliothèque permettant au développeur de spécifier des opérateurs pour approximer des fonctions arbitraires en précision arbitraire est présentée. Deux approches pour l'interfaçage de cette bibliothèque avec les outils de HLS sont proposées, selon que l'on a ou pas accès aux sources des compilateurs HLS.

Mots-clés : Synthèse de haut niveau, formats numériques, IEEE-754, Approximation de fonctions, compilation, Posit

Laboratoire(s) de recherche : Centre of Innovation in Telecommunications and Integration of Service (CITI)

Directeur de thèse : Florent DE DINECHIN

Président du jury : Fabrice RASTELLO

Composition du jury :
Roselyne CHOTIN (Examinatrice)
Steven DERRIEN (Rapporteur)
Fabrizio FERRANDI (Rapporteur)
Anastasia VOLKOVA (Examinatrice)