High-level synthesis and arithmetic optimizations

Yohann Uguen

N°d'ordre NNT : 2019LYSEI099

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

Opérée au sein de :

(INSA de Lyon, CITI lab)

Ecole Doctorale InfoMaths EDA 512
(Informatique Mathématique)

Spécialité de doctorat : Informatique

Soutenue publiquement le 13/11/2019, par :

Yohann Uguen

High-level synthesis and arithmetic optimizations

Devant le jury composé de :

Frédéric Pétrot		**Président**
Professeur des Universités,	TIMA, Grenoble, France	
Philippe Coussy		**Rapporteur**
Professeur des Universités,	UBS, Lorient, France	
Olivier Sentieys		**Rapporteur**
Professeur des Universités,	Univ. Rennes, Inria, IRISA, Rennes	
Laure Gonnord		**Examinatrice**
Maître de conférence,	Université Lyon 1, France	
Martin Kumm		**Examinateur**
Professeur des Universités,	Université de Fulda, Allemagne	
Florent de Dinechin		**Directeur de thèse**
Professeur des Universités,	INSA Lyon, France	

Résumé

À cause de la nature relativement jeune des outils de synthèse de haut-niveau (HLS), de nombreuses optimisations arithmétiques n'y sont pas encore implémentées. Cette thèse propose des optimisations arithmétiques se servant du contexte spécifique dans lequel les opérateurs sont instanciés. Certaines optimisations sont de simples spécialisations d'opérateurs, respectant la sémantique du C. D'autres nécéssitent de s'éloigner de cette sémantique pour améliorer le compromis précision/coût/performance. Cette proposition est démontré sur des sommes de produits de nombres flottants. La somme est réalisée dans un format en virgule-fixe défini par son contexte.

Quand trop peu d'informations sont disponibles pour définir ce format en virgule-fixe, une stratégie est de générer un accumulateur couvrant l'intégralité du format flottant. Cette thèse explore plusieurs implémentations d'un tel accumulateur. L'utilisation d'une représentation en complément à deux permet de réduire le chemin critique de la boucle d'accumulation, ainsi que la quantité de ressources utilisées.

Un format alternatif aux nombres flottants, appelé posit, propose d'utiliser un encodage à précision variable. De plus, ce format est augmenté par un accumulateur exact. Pour évaluer précisément le coût matériel de ce format, cette thèse présente des architectures d'opérateurs posits, implémentés avec le même degré d'optimisation que celui de l'état de l'art des opérateurs flottants. Une analyse détaillée montre que le coût des opérateurs posits est malgré tout bien plus élevé que celui de leurs équivalents flottants.

Enfin, cette thèse présente une couche de compatibilité entre outils de HLS, permettant de viser plusieurs outils avec un seul code. Cette bibliothèque implémente un type d'entiers de taille variable, avec de plus une sémantique strictement typée, ainsi qu'un ensemble d'opérateurs ad-hoc optimisés.

Abstract

High-level synthesis (HLS) tools offer increased productivity regarding FPGA programming. However, due to their relatively young nature, they still lack many arithmetic optimizations. This thesis proposes safe arithmetic optimizations that should always be applied. These optimizations are simple operator specializations, following the C semantic. Other require to a lift the semantic embedded in high-level input program languages, which are inherited from software programming, for an improved accuracy/cost/performance ratio. To demonstrate this claim, the sum-of-product of floating-point numbers is used as a case study. The sum is performed on a fixed-point format, which is tailored to the application, according to the context in which the operator is instantiated.

In some cases, there is not enough information about the input data to tailor the fixed-point accumulator. The fall-back strategy used in this thesis is to generate an accumulator covering the entire floating-point range. This thesis explores different strategies for implementing such a large accumulator, including new ones. The use of a 2's complement representation instead of a sign+magnitude is demonstrated to save resources and to reduce the accumulation loop delay.

Based on a tapered precision scheme and an exact accumulator, the posit number systems claims to be a candidate to replace the IEEE floating-point format. A throughout analysis of posit operators is performed, using the same level of hardware optimization as state-of-the-art floating-point operators. Their cost remains much higher that their floating-point counterparts in terms of resource usage and performance.

Finally, this thesis presents a compatibility layer for HLS tools that allows one code to be deployed on multiple tools. This library implements a strongly typed custom size integer type along side a set of optimized custom operators.

Contents

Remerciements (Acknowledgements)

Cette thèse n'aurait pu être possible sans un certain nombre de personnes clés.

D'un point de vue scientifique, cette thèse s'appuie sur deux personnes: Mon directeur de thèse *Florent de Dinechin* pour l'arithmétique des ordinateurs et mon mentor rennais *Steven Derrien* pour la compilation. Je n'aurai pas pu rêver mieux que *Florent de Dinechin* en tant que directeur de thèse. Il a non seulement essayé de me transmettre ses connaissances mais également ses intuitions. Un puits de savoir me laissant croire que nous avions tous les deux des choses à nous apporter. Je serai à jamais reconnaissant de tout ce qu'il m'a transmis. Avant lui, *Steven Derrien* m'avait ouvert les portes de la recherche académique dès la licence. Sans retenue, il m'a formé à des outils modernes et confronté à des problématiques inexplorées. À mi-chemin entre la compilation et l'architecture, il a su me pousser vers un filon prometteur, filon qui a fini par déboucher sur cette thèse.

D'un point de vue personnel, j'ai toujours été aveuglément soutenu par *ma mère* et ma famille malgré quelques virages le long du chemin. Il semblerait que le fait d'avoir atteint *une* destination nous fasse relativiser l'importance de la destination originelle. Bien évidemment, le bonheur quotidien partagé avec *Léonie Caprais/Bliskiye* m'a offert un contexte de vie et de travail idéal. Elle a su rester ma supportrice officielle, malgré mon humour... discutable. Même lors de ses réactions bilatérales symétriques, elle a su être l'oreille attentive dont j'avais besoin. Son rôle était était essentiel et m'a permis de prendre du recul sur mon travail. Depuis mes débuts en informatique, la présence inconditionnelle de mon cher ami *Thomas Lefeuvre/Trackman_* était aussi indispensable à ma réussite. Pendant nos études comme dans les jeux vidéos, notre symbiose et notre travail d'équipe ont toujours payé. Et c'est sans compter sur son humour douteux (partagé) dans les moments difficiles (ou non). D'autres m'ont rejoint plus loin au cours de ce voyage comme *Jean-Baptiste Trystram/SuperMaki_*, m'enseignant la mécanique, que ce soit les mains dans le cambouis ou face à un tableau blanc.

Plus largement, l'ensemble des personnes que je côtoie m'ont, de près ou de loin, aidé dans cette aventure. Je pense notamment à *Guillaume Uguen* et *Fanny Marrot* avec leurs week-ends "dégustation", *Jonathan Tournier* pour les sessions d'escalade, ainsi que *Marion Trystram* et *Amandine Chaudier* pour les soirées lyonnaises et les week-ends à la montagne. Je pense aussi aux doctorants du CITI comme *Luc Forget* pour ses collaborations et son esprit critique, m'ayant apporté une nouvelle façon de travailler, mais aussi à tous les autres: *David Kibloff*, *Tristan Delizy*, *Gautier Berthou* etc. J'ai aussi une pensée pour le reste des membres du CITI et en particulier pour *Guillaume Salagnac* et nos échanges enrichissants.

À l'ensemble de ces personnes, merci.

Chapter 1

Introduction

> If you don't know how to start your introduction, you can always begin by saying: *Even the Romans...*
>
> An old German teacher...

Even the Romans faced the difficulty of building a good number system. Indeed, the Roman numerals lack some computation qualities. In that matter, a good number system should find a good trade-off between:

(1) being a compact representation,

(2) being easy to interpret,

(3) being easy to compute on.

The Roman numerals are a base-10 system, just like the usual base-10 Arabic numerals. In terms of (1) being a compact representation, the Roman numerals sometimes are more compact than the Arabic system (e.g $M = 1000$). However, most of the time they are not. Indeed, the representation of one decimal digit in the Arabic system requires between zero and four characters (e.g. $IX = 10 - 1 = 9$, $VIII = 5 + 1 + 1 + 1 = 8$). This also means that (2) the interpretation of a value requires many intermediate computations. The reader is invited to parse my birth year ($MCMXCII$), starting by identifying the character groups that correspond to each power of ten. Since Roman numerals encode a base-10 position system, it is however (3) possible to perform additions and subtractions. The corresponding algorithm is more complex than its base-10 Arabic system counterpart. Still, there were abacuses (mechanical computing devices) for Roman numerals.

The base 10 Arabic numerals improve Roman numerals on (1), (2) and (3). This probably explains why Roman numerals are now only used for aesthetics, for example on some clock faces.

The example of Roman numerals demonstrates that representing the set of integers using textual characters is not straightforward. Representing real numbers is even more difficult, especially if one intends to automate the computation. In a physical computing device (abacus or computer), a number is represented by a discrete finite quantity of information. As the set of integers is discrete, it is relatively easy to map a finite subset of integers to such a device. However the reals are a continuous set, therefore an additional difficulty is to chose the finite discrete subset that can be represented.

The general consensus on representation of real numbers in modern computers is the IEEE-754 standard [1]. It defines floating-point formats that are followed by most central processing units (CPUs). This standard is so well established that a CPU performance is usually measured as its peak FLOP/sec (floating-point operations per second). For that matter, the performance of the floating-point units (FPUs) has been, and is still, a subject of improvement [2, 3, 4, 5, 6, 7, 8, 9].

As CPUs are general-purpose processors, they are built from a low count of fast but complex cores. Alternatively, graphics processing units (GPUs) are built from a very high count of slow and simple cores. CPUs are therefore optimized for latency of general purpose applications, while GPUs are optimized for throughput of highly parallel applications. In both cases, because of their genericity, these processors implement an instruction based mechanism. The latter requires to be able to decode and execute said instructions before storing the results.

When optimizing a specific application, performance improvements can be achieved through the use of a custom hardware accelerator. In this context, the overheads due to genericity can be mitigated. For example, there is no need for instructions and their associated mechanisms. Also, the wide variety of operators can be limited to the required ones. Those can even be replicated as many times as possible. Regarding floating-point operators, the application might not even require strict IEEE-754 compliance [10, 11].

Further optimizations can be made through operator specialization. For instance, a complete floating-point divisor/multiplier might not be required if the application only performs a division/multiplication by a constant [12, 13, 14].

Depending on the application accuracy requirements and the nature of the input data, the datapath of the circuit can be tailored so that enough but no more bits than needed are computed. Examples of such circuits range from artificial intelligence performed using ternary arithmetic [15] to digital signal processing filters [16].

All these optimizations are enabled by moving away from generic processors to application-specific accelerators.

To implement an accelerator, several technologies can be used. Application-specific integrated circuits (ASICs) offer the most performance. However, the manufacturing process of such hardware is very expensive. This cost can only be mitigated by a large production. An alternative is to use circuits that can be reconfigured after manufacturing such as field-programmable gate arrays (FPGAs). The cost of such circuits is reduced as they can be produced at a large scale. For a given application, FPGAs can offer better performance than CPUs at a fraction of the cost of an ASIC.

The highly customizable nature of FPGAs comes at the price of a more difficult programming model than the one of general purpose processors. Indeed, programming a FPGA is done through low level hardware description languages (HDLs). These are notoriously error prone and difficult to debug. However, because of the very mature nature of HDLs, many libraries and code generators provide highly optimized designs for programmers to build around.

To increase the productivity of hardware designers, new techniques try to compile classical programming languages to HDLs. Such a process is called high-level synthesis (HLS). This compilation flow is less mature than classical HDL synthesis. HLS-generated designs has long been inferior to handwritten HDL components regarding performance and resource consumption. However, HLS offered increased productivity. This is less and less true: in many situations, HLS now not only improves productivity, but also the quality of

the results. This thesis shows several contributions to this trend, all related to arithmetic computations.

First, most arithmetic optimizations in current HLS tools rely on their underlying compiler frameworks. In particular, the instruction selection of the CPU backend is used as a *better than nothing* optimization. However, the FPGA arithmetic community has long worked on better FPGA-specific optimizations. A first axis of this thesis, described in Chapter 3, is to try to fill this gap between the compiler designers community and the arithmetic community, when targeting FPGAs.

As a second axis, Chapters 4 and 5 are experiments in using HLS to explore the architecture design space for two non-standard arithmetic objects: Kulisch's exact floating-point accumulators and Gustafson's posits (an alternative to floating-point for representing real numbers). The use of C++ templates enables genericity beyond what HDL generators can offer. Furthermore, templated operators are compiled and optimized by the HLS tools in their context. In both cases, the operators libraries designed this way are the state of the art.

When performing such architecture explorations, the use of a custom-size integer library is required for tailoring the datapath. As HLS tools do not all use the same integer library, source code cannot be shared between tools. Furthermore, each FPGA vendor uses its own HLS tool, which makes the generated code suboptimal on FPGAs of a different vendor. This is problematic to compare novel designs to the state-of-the-art if not targeting the same FPGA brand. A third and last axis of this thesis, described in Chapter 6, is to build an abstraction layer on top of vendor custom-size integer libraries. Thus, it allows a programmer to only write one component for all supported tools. Furthermore, the proposed library embeds a type-safe semantic and offers several useful operators.

Beforehand, a preliminary Chapter (2) introduces the necessary notions to understand this thesis. It starts by reminding the basic formats for representing real numbers and perform arithmetic computations. It then gives the necessary an overview of FPGAs architecture and their programming model.

Chapter 2

Context

As this thesis is about arithmetic optimizations for high-level synthesis tools, this Chapter first introduces number representations onto which arithmetic optimizations are performed. A second part presents the hardware targeted in this work and their programming model, in particular it presents high-level synthesis tools.

2.1 Representing real numbers

The efficient manipulation of real numbers in computers relies on the representation in which the data is approximated. The choice of such a representation is often not straightforward as it is tied to a accuracy/speed/ease-of-use/cost trade-off. The most widely used ways of representing real numbers is floating-point arithmetic presented in 2.1.2. There are emerging alternative formats, such as posits that will be presented in Chapter 5. Floating-point and posit operators are built out of simpler fixed-point operations. Therefore, fixed-point arithmetic is presented in 2.1.1.

2.1.1 Fixed-point representation

A N-bit unsigned fixed-point format is defined by the weight w_{lsb} of its least significant bit (LSB) and the weight w_{msb} of its most significant bit (MSB), with $w_{\text{msb}} = w_{\text{lsb}} + N - 1$. A number in such a format holds N weighted binary digits x_i and represents the rational number

$$x = \sum_{i=w_{\text{lsb}}}^{w_{\text{msb}}} 2^i x_i$$

An example is given in Figure 2.1. Here $(w_{\text{msb}}, w_{\text{lsb}}) = (6, -9)$, hence $N = 16$. The LSB is then the 9^{th} fraction bit from the decimal point (which is actually here a binary point).

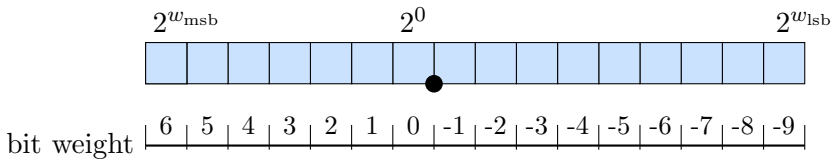


Figure 2.1: The bits of an unsigned fixed-point format, here $(w_{\text{msb}}, w_{\text{lsb}}) = (6, -9)$.

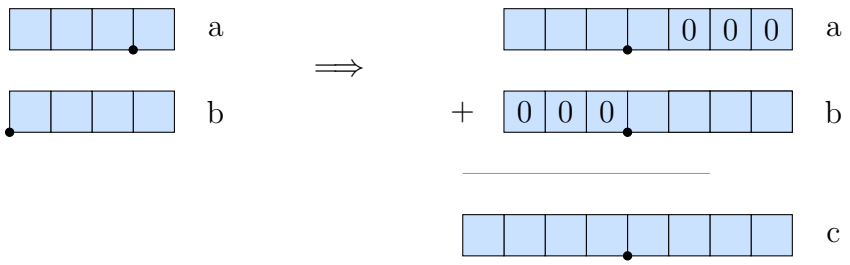


Figure 2.2: Illustration of the summation of unsigned fixed-point numbers with different formats.

Negative numbers are supported by either using a sign+magnitude or a 2's complement encoding. The sign+magnitude encoding adds a sign bit to specify the signedness of the fixed-point value. The 2's complement encoding stores negative numbers as their complement with respect to 2^N. As an example, the 2's complement of the 4-bit value 0110_b is 1010_b as $0110_b + 1010_b = 10000_b$. The 2's complement encoding has the advantage that the fundamental arithmetic operations of addition and subtraction are identical to those for unsigned binary numbers, which is not the case of the sign+magnitude arithmetic. A value encoded using the 2's complement encoding can be decoded as follows:

$$x = -2^{-w_{\text{msb}}} + \sum_{i=w_{\text{lsb}}}^{w_{\text{msb}}-1} 2^i x_i$$

Fixed-point arithmetic requires carefulness if the format changes during the computation. As a general example, lets consider a and b where a uses a (w_{msb_a},w_{lsb_a}) fixed-point format and b a (w_{msb_b},w_{lsb_b}) one. Their exact sum $c = a + b$ requires a (w_{msb_c},w_{lsb_c}) where $w_{\text{msb}_c} = \max(w_{\text{msb}_a}, w_{\text{msb}_b}) + 1$ and $w_{\text{lsb}_c} = \min(w_{\text{lsb}_a}, w_{\text{lsb}_b})$. Indeed, prior to performing the operation, the numbers are first aligned according to their bit weights. Figure 2.2 illustrates this alignment where a uses a (2, -1) format and b uses a (-1, -4) format. Here a has 3 decimal bits and 1 fractional bit where b has no decimal bit and 4 fractional bits. In that case, b must be arithmetically shifted so that it aligns with a, as a holds the most significant bits before the summation can occur. Therefore, the final result c, if computed exactly, requires a (3,-4) fixed-point format.

The extra bit added to $\max(w_{\text{msb}_a}, w_{\text{msb}_b})$ is to capture the output carry that can be produced by the addition. Indeed, once the inputs are aligned, the sum of two N-bit fixed-point values will produce the following exact answer:

$$\begin{aligned} z &= x + y \\ &= \sum_{i=w_{\text{lsb}}}^{w_{\text{msb}}} 2^i x_i + \sum_{i=w_{\text{lsb}}}^{w_{\text{msb}}} 2^i y_i \\ &= \sum_{i=w_{\text{lsb}}}^{w_{\text{msb}}} 2^i (x_i + y_i) \\ &= \sum_{i=w_{\text{lsb}}}^{w_{\text{msb}}+1} 2^i z_i \end{aligned}$$

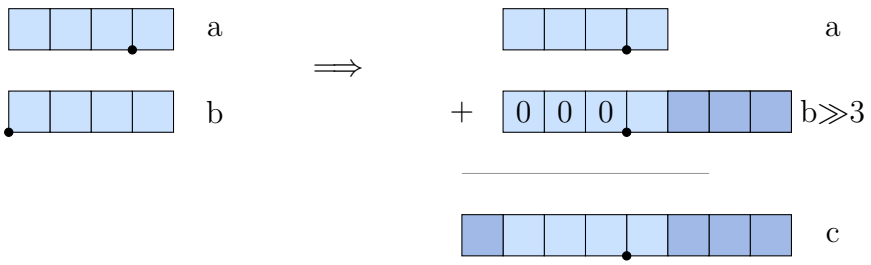


Figure 2.3: Illustration of the summation of two numbers usign different unsigned fixed-point formats using only the 4-bit integer arithmetic.

In a software context When programming a CPU, fixed-point numbers are computed using integer arithmetic. Support for fixed-point numbers is then easy and fast. However, values must use a format where N is of a standard integer size. The formats are chosen before the computation according to the manipulated values and their associated operations. However, when manipulating variables, chosing a format can be challeging as the range and the accuracy of the data might not be known beforehand. Having a software mechanism to change the format at runtime according to the current value of the variable would be very costly - it would be some form of floating point. As most programs rely on variables, fixed-point formats are difficult to use in the general case.

The programmer is in charge of handling the change of format of the variables after each operation. To illustrate this format management, let us consider 4-bit integer arithmetic. The example from Figure 2.2 can be modified to the one from Figure 2.3. The programmer must have chosen the formats of the three variables. Here, a and c uses a (2,-1) format and b uses a (-1,-4) format. The variable b must first be shifted by 3 to the right before the summation. The 3 LSBs of b are then discarded. The 4-bit addition is then performed and the extra bit of the exact sum must be discarded because it does not belong to the format of the result. All the discarded bits entail numerical errors.

For all these reasons, code written using a fixed-point representation is usually not flexible. Little code modifications might make the programmer change and debug his entire program again.

A software emulation of a larger fixed-point format can be obtained by chaining multiple standard size integers. This tedious process can be eased by using dedicated C/C++ libraries such as GMP [17]. This chaining approach reveals a performance/accuracy trade-off. Also, some previous work try to ease the fixed-point usage by inferring the correct fixed-point formats for a given application [18, 19, 20, 21, 22].

In a hardware context When designing custom hardware, the standard sizes are no longer a limitation. Each operator and datapath can be tailored. The size of the format then becomes a parameter of the architecture to control the accuracy. This still entails issues related to format conversion, operations on data formats, etc. Chapter 6 will include and in-depth discussion on these issues.

2.1.2 Floating-point representation

A N-bit floating-point format is defined by the size wd_e of its exponent e and the size wd_f of its signed significand f, with $N = 1 + wd_e + wd_f$. The extra bit is used to store the

sign s of the encoded value. Such an encoding is illustrated by Figure 2.4.

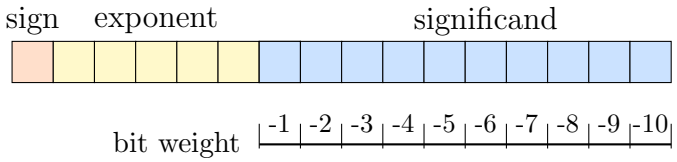


Figure 2.4: The bits of a floating-point format, here $(wd_e, wd_f) = (5, 10)$.

The value x of a normal floating-point number is defined [23] as:

$$x \;=\; (-1)^s \;\cdot\; 2^e \;\cdot\; 1.f$$

The notation $1.f$ expresses that an implicit bit is appended to the significand with a weight of 0. The exponent e must be signed so that small values may be represented as well as large values. Instead of using a 2's complement encoding, the choice here is a biased representation: the exponent bits represent an unsigned integer k, and the exponent e is $e = k - \text{bias}$, where $\text{bias} = 2^{wd_e-1} - 1$. This choice eases the comparisons as it makes the floating-point numbers to follow the natural order of their encodings (see Table 2.1 below). Hence, if f_i is the bit of f with weight i, the decoded floating-point value is:

$$x = (-1)^s \;\cdot\; 2^{k-\text{bias}} \;\cdot\; (1 + \sum_{i=-1}^{-wd_f} 2^i f_i)$$

1	1	0	0	1	1	1	0	0

Figure 2.5: Floating-point value example where $(wd_e, wd_f) = (3, 5)$.

To illustrate this decoding, Figure 2.5 shows a floating-point value on a $(wd_e, wd_f) = (3, 5)$ format. Here $s = 1$, $k = 4$ and $f = 2^{-1} + 2^{-2} + 2^{-3} = 0.875$. Hence, the decoded value is:

$$\begin{aligned} x &= (-1)^1 \;\cdot\; (2^{4-(2^{3-1}-1)}) \;\cdot\; 1.875 \\ &= -1 \;\cdot\; 2 \;\cdot\; 1.875 \\ &= -3.75 \end{aligned}$$

Subnormal numbers As the implicit bit is set to 1, values close to 0 (included) cannot be represented. To illustrate this phenomenon, Figure 2.6 represents normal floating-points ($(wd_e, wd_f) = (3, 2)$) on the real axis. The smallest positive and largest negative values that can be represented in this format are $1.00_b \cdot 2^{-3}$ and $-1.00_b \cdot 2^{-3}$. This leads to a large gap centered in 0. To cope with this phenomenon, the IEEE-754 standard chooses to set the implicit bit to 0 instead of 1 when the minimum exponent is reached (0 when unbiased). The floating-point number is no longer a normal but rather a subnormal (or denormal) number. Figure 2.7 shows the subnormal numbers in red on the real axis next the the normal numbers (in black).

To illustrate the natural order on floating-point values, Table 2.1 shows the decoding of all the positive values of a (wd_e, wd_f) = (3,2) format.

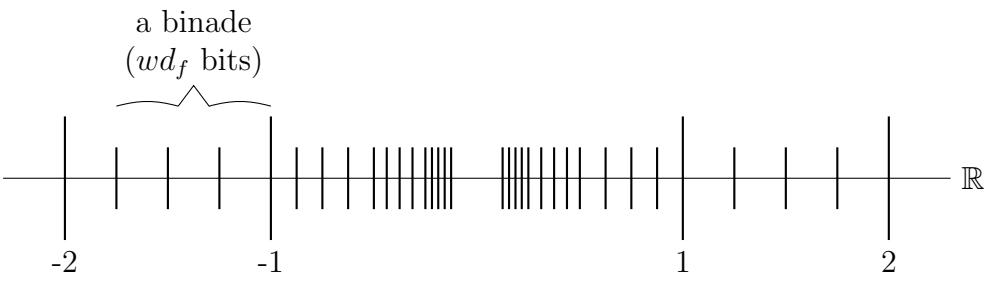


Figure 2.6: Illustration of floating-point density on the real axis $((wd_e, wd_f) = (3, 2))$

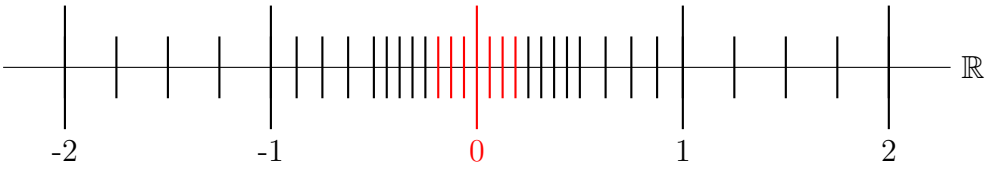


Figure 2.7: Illustration of floating-point density on the real axis $((wd_e, wd_f) = (3, 2))$. Normal numbers are represented in black and subnormals are represented in red. Note that there are two zeroes: $+0$ and -0.

Standard sizes and values The IEEE-754 standard [1] defines sizes, special values, arithmetic behaviors and rounding modes for floating-points. The three floating-point formats introduced with this standard are:

- binary32 $(wd_e = 8, wd_f = 23)$, formerly called single precision, which corresponds to the `float` type in C,
- binary64 $(wd_e = 11, wd_f = 52)$, formerly called double precision, which corresponds to the `double` type in C,
- binary128 $(wd_e = 15, wd_f = 112)$, formerly called quadruple precision.

Some special values are also encoded by specific combinations of the sign, exponent and significand: plus and minus infinity, positive and negative zeros and not a number (NaN). Hence each operator has special cases to handle special values arithmetic. For example, the addition of $x + \infty$ is defined as ∞ if x is finite, while $\infty - \infty$ returns $-$NaN.

Rounding The IEEE-754 standard defines 5 rounding modes. Two rules round to the nearest value while the others rounds towards a direction. The rules that round to the nearest differ in the way they handle ties, i.e. values exactly between the two nearest floating point numbers. One is *round to nearest, ties to even* while the other is *round to nearest, ties away from 0*. The rules that perform a directed rounding are *towards 0*, *towards* ∞ and *towards* $-\infty$. The default behavior is to use the *round to nearest, ties to even* rounding. For a floating-point operator to be IEEE-754 compliant, it must implement these rounding modes. The output of a IEEE-754 compliant operator can then be written as:

$$fp_{op}(a, b) = rnd(op(a, b))$$

where $fp_{op}(a, b)$ is the output of the floating-point operator, $op(a, b)$ is the mathematical operation and $rnd(...)$ is the rounding rule.

Exp.	Signif.	Value	Comment
000	00	0	(positive) zero
000	01	$0.01 \cdot 2^{1-3} = 0.0625$	smallest positive subnormal
000	10	$0.10 \cdot 2^{1-3} = 0.125$	
000	11	$0.11 \cdot 2^{1-3} = 0.1875$	largest subnormal
001	00	$1.00 \cdot 2^{1-3} = 0.25$	smallest normal
001	01	$1.01 \cdot 2^{1-3} = 0.3125$	
001	10	$1.10 \cdot 2^{1-3} = 0.375$	
001	11	$1.11 \cdot 2^{1-3} = 0.4375$	
010	00	$1.00 \cdot 2^{2-3} = 0.5$	
010	01	$1.01 \cdot 2^{2-3} = 0.625$	
010	10	$1.10 \cdot 2^{2-3} = 0.75$	
010	11	$1.11 \cdot 2^{2-3} = 0.875$	
011	00	$1.00 \cdot 2^{3-3} = 1$	
011	01	$1.01 \cdot 2^{3-3} = 1.25$	
011	10	$1.10 \cdot 2^{3-3} = 1.5$	
011	11	$1.11 \cdot 2^{3-3} = 1.75$	
100	00	$1.00 \cdot 2^{4-3} = 2$	
100	01	$1.01 \cdot 2^{4-3} = 2.5$	
100	10	$1.10 \cdot 2^{4-3} = 3$	
100	11	$1.11 \cdot 2^{4-3} = 3.5$	
101	00	$1.00 \cdot 2^{5-3} = 4$	
101	01	$1.01 \cdot 2^{5-3} = 5$	
101	10	$1.10 \cdot 2^{5-3} = 6$	
101	11	$1.11 \cdot 2^{5-3} = 7$	
110	00	$1.00 \cdot 2^{6-3} = 8$	
110	01	$1.01 \cdot 2^{6-3} = 10$	
110	10	$1.10 \cdot 2^{6-3} = 12$	
110	11	$1.11 \cdot 2^{6-3} = 14$	largest normal
111	00	$+\infty$	
111	01	NaN	
111	10	NaN	(with different payloads)
111	11	NaN	

Table 2.1: All positive values of a floating-point (wd_e, wd_f) = (3,2) format.

Reproducible results The rounding errors of floating-point operators implies that a sequence of operations cannot be reordered to ensure the reproducibility of the results. An illustrative example is to consider a custom floating-point format where $wd_f = 3$ and the operation $fp\_(fp\_+(a, b), c)$ where $a = 1.101_b \cdot 2^3$, $b = 1.010_b \cdot 2^0$ and $c = 1.100_b \cdot 2^3$. This sequence of operations cannot be replaced by $fp\_+(fp\_(a, c), b)$ even if the associated mathematical sequence of operations $a + b - c$ is equivalent to $a - c + b$. Figure 2.8 illustrates this example.

The sum of a and b (top) is rounded so that the result fits in wd_f bits of significand.

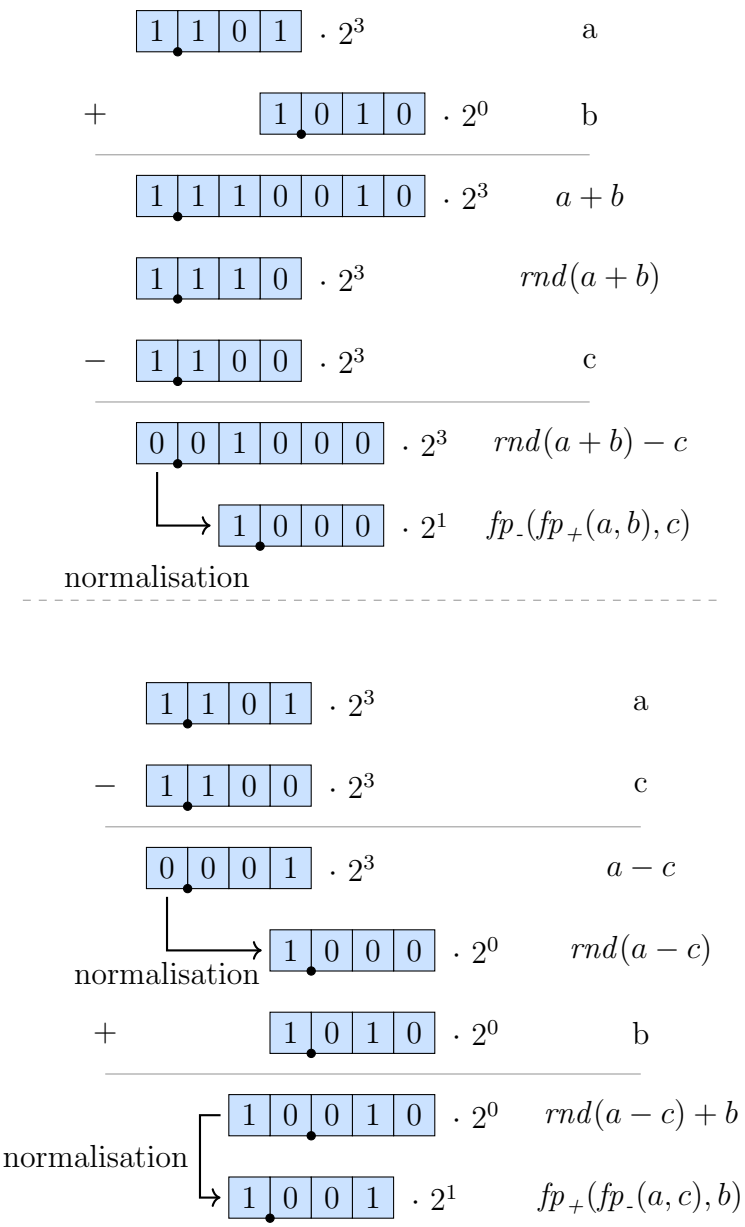


Figure 2.8: Illustration of non-associativity of the floating-point addition ($wd_f = 3$).

Hence, one bit of b is lost in the process. However, performing a different sequence of operations (bottom) can modify the rounding error.

Implementation Combining all these properties, any implementation of an IEEE-754 floating-point operation requires to: decode the inputs; detect special values; compute the arithmetic function; normalize the computed result (shift significand to have its most significant bit at 1 while modifying the exponent accordingly); round the result. These steps require either time for software implementations ([24]) or both time and resources for hardware implementations.

The subtleties of the encoding are completely hidden to the programmer and rely on the underlying more complex hardware or software implementation. Therefore it makes it easy to use.

2.2 Field Programmable Gate Arrays (FPGAs)

A field programmable gate array (FPGA) is a hardware component that can be configured by software in such a way that it allows to simulate complex logic functions. This Section aims at giving the reader enough information to understand the basics of FPGAs architecture, programming model and the corresponding synthesis reports. It does not provide a complete understanding of modern FPGAs and their complexity.

2.2.1 FPGAs architecture

FPGAs are based on small memories (lookup tables) connected together through an interconnect network. Each vendor builds upon this concept with its own hardware structure/organization. However, the general programming model and functionalities are similar. The FPGAs vendors considered in this thesis are Xilinx and Intel (formerly Altera).

Before detailing FPGAs' architecture, it is important to differentiate two moments in their life cycle:

- configuration time: the FPGA is configured to simulate a specific function
- execution time: data is processed by the simulated function and produces the output

Configurable logic functions using LookUp Tables (LUTs)

A LUT is a memory, holding a truth table, that can be configured by software. Given α input bits, a LUTα will return a single bit from its 2^α entries. Hence, it is capable of simulating any boolean function on α bits.

A simple example of a LUT is given in Figure 2.9 (left). It takes 2 bits of input (LUT2) and outputs a single bit. These 2 bits are used to select one of the configured bit (illustrated by ▢) using a multiplexer. Hence, changing the configuration bits changes the boolean function simulated.

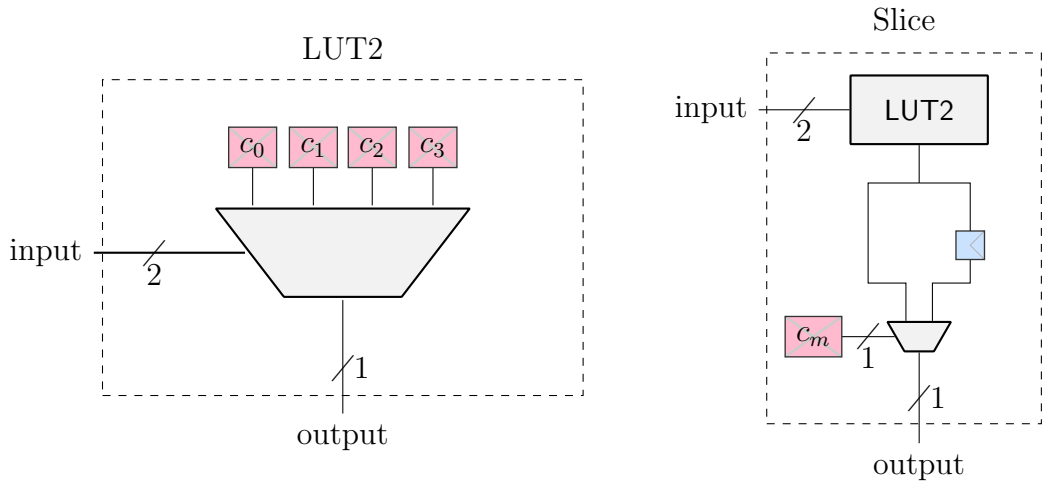


Figure 2.9: Example of a LUT2 (left) and its integration in a simple slice with a register (right).

The configuration bits are set prior to the execution and cannot be modified during the execution. However, during the execution, the data bits are used as inputs to compute the output.

Modern FPGAs use between 4-input and 6-input LUTs. The combination of such LUTs is the basis of FPGA's architecture.

Slice and Adaptive Logic Modules (ALMs)

A Slice or ALM (for Xilinx and Intel FPGAs respectively) is a combination of one or more LUTs with registers. The latter are memories that can store data bits synchronously during the execution and are represented by ◻ in Figures. In the case of a Slice/ALM, they are used to store the output of the LUT, as illustrated by Figure 2.9 (right). An extra configuration bit (c_m) selects between the LUT output and the previously stored value. From now on, we will refer as a slice for naming both Slices and ALMs.

Modern slices will combine multiple LUTs and registers as well as one or more full adders. An example of such a slice is depicted in Figure 2.10. This slice takes two 2-bit input signals that address two LUT2. Two configuration bits control which signals to forward between the direct output of the LUTs and their sum. Finally, two more configuration bits select the two output bits between the delayed result and the immediate result.

A new configuration bit is added to the slice (c_c) to fill the carry input port of the full adder. However, a special input of the slice (C_{in}) can be used instead of c_c. Therefore, another configuration bit is used to perform that selection. This example is still a simplistic illustration of modern FPGAs slices.

This slice example has 6 configuration bits (plus 4 in each LUT2); and 4 data bits.

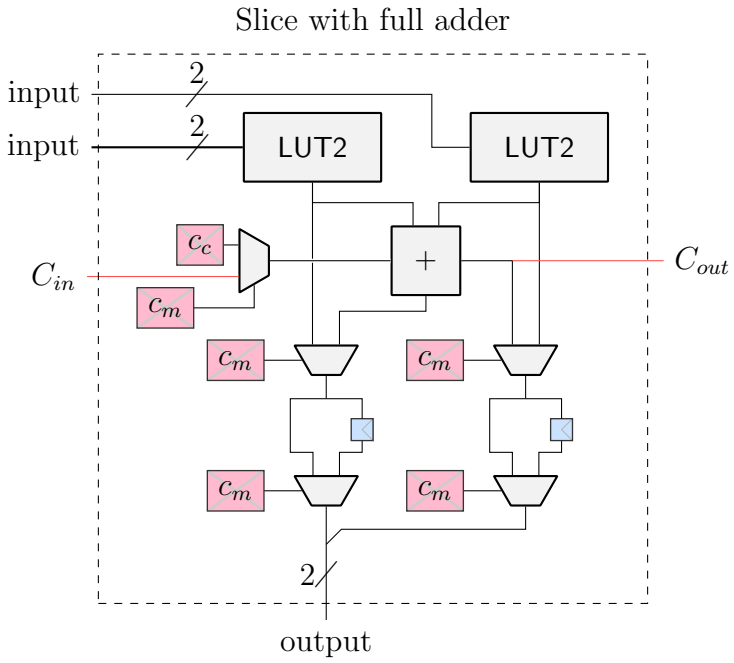


Figure 2.10: Example of a slice with two LUT2, two registers and a full adder.

Interconnect

An interconnect is configured to route slices inputs and outputs, allowing to simulate more complex logic functions. Such an interaction between slices is depicted in Figure 2.11.

The interconnect is a set of wires connected to the slices in a grid manner. At each crossing point, a switch matrix decides where to route each wire. Similarly to the slice configuration bits, the switch matrix configuration is performed by software before the start of the computation. There has been a lot of research on switch matrices [25, 26, 27]. The switch matrix details are currently well hidden behind FPGA synthesis tools.

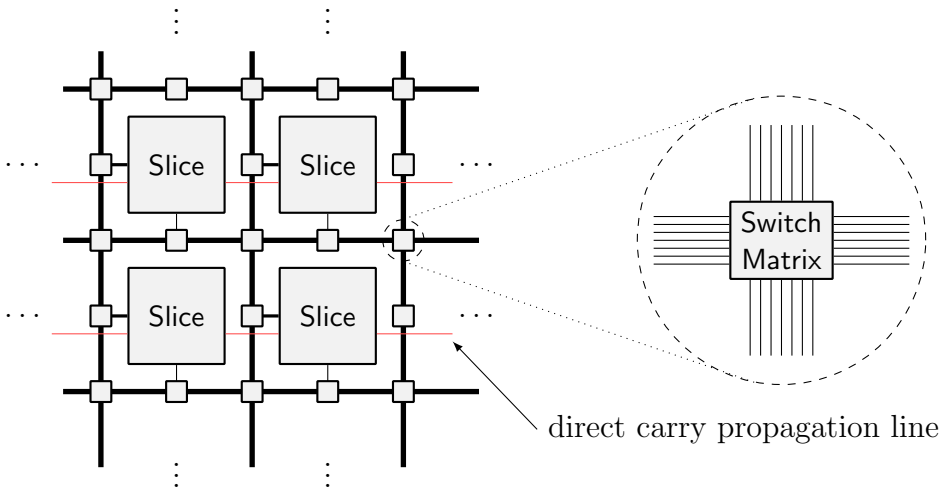


Figure 2.11: Example of slices connected through an interconnect.

A special type of connection creates a direct link between adjacent slices without going through the interconnect. This allows to perform faster operations as it removes the latency of the routing. It is depicted by the red connections from Figure 2.11. This mechanism is particularly used for carry propagation when performing additions, giving it its name: *carry chain*. However it can also be used for other functions such as wide AND or wide OR.

Block RAM (BRAM)

The registers inside a slice can be linked to build a larger memory. However, such a sparse distribution of data may imply longer routing delays. To cope with this, FPGAs are equipped with small block RAMs (BRAMs) connected to the interconnect. These store data in a much more dense fashion, hence allowing LUTs to be used for other purposes.

Digital Signal Processing (DSP) blocks

FPGAs contain dedicated hardware blocks (DSPs) for accelerating commonly used functions. For example, integer multipliers and multiply-accumulate units are available as DPS blocks on both Intel and Xilinx FPGAs. Most recent Intel FPGAs even embed DSPs with integrated single-precision floating-point multipliers [28].

Figure 2.12 illustrates a combination of the above mentioned components to make a simple FPGA. The interconnect is also connected to different controllers which allow communication with the outside world.

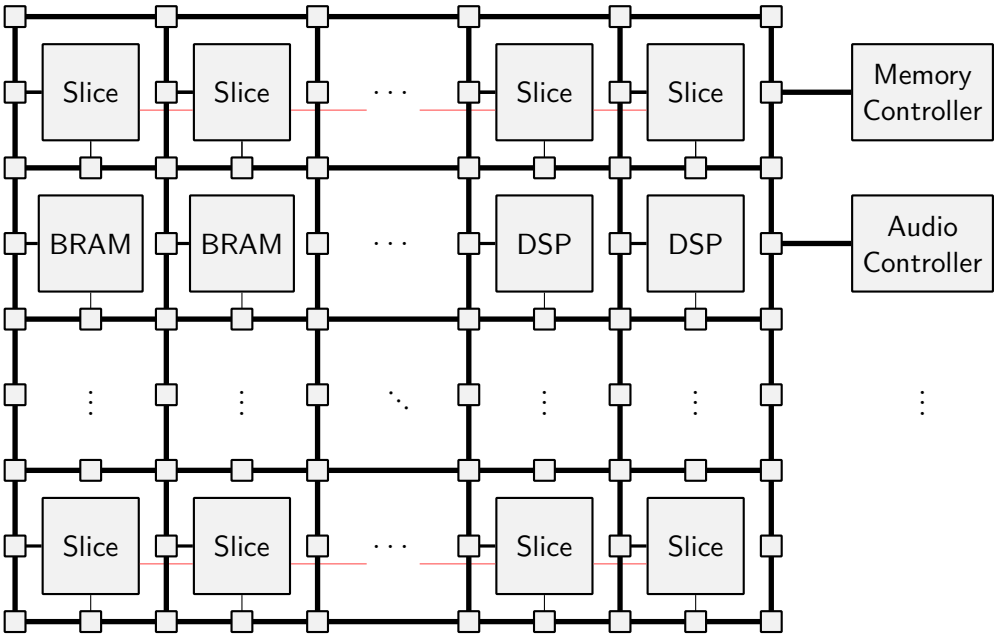


Figure 2.12: Example of a populated FPGA with slices, DSPs and BRAMs.

The complex task of configuring each slice and the interconnect is well hidden behind synthesis tools. Indeed, these are able to generate a bitstream (that is composed of all the configuration bits), that program the FPGA, from an input program. The methodology used by synthesis tools for generating such a configuration file is not described here. However, it is useful for the reader to know that the final step before the bitstream generation is called *place-and-route*. This step is very computationally intensive (it may last days for large FPGAs).

2.2.2 FPGAs programming model

Figure 2.13 illustrates the typical flows for programming FPGAs. The synthesis (right) is the process of generating a bitstream from a low-level description of a circuit, using a hardware description language (HDL). Hand writing a component using a HDL is today the standard for creating optimized designs. Another approach is to generate a HDL program from a higher-level language. This is the process of high-level synthesis (Figure 2.13, left). Both approaches are described in the next sections.

Hardware Description Languages (HDLs)

The usual way of describing a circuit is to use a HDL such as VHDL or Verilog. Such languages do not follow a usual sequential semantic. Instead, they describe components connected together through wires and buses with a partially parallel semantic.

Writing a component using a HDL is considered quite tedious. First, it departs from traditional programming. Second, debugging tools give the user a view of the state of the component at given a time. This requires the programmer to inspect the state of wires, and follow where each wire is connected. In contrast, when using a high-level language debugger, the user follows a sequential program step by step where only one change occurs at the time.

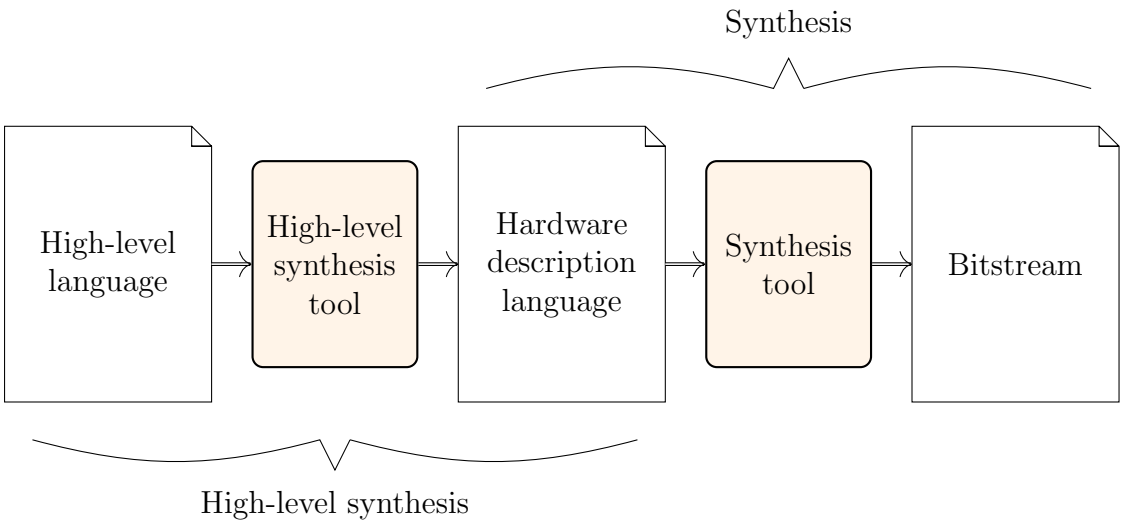


Figure 2.13: Design flow for programming FPGAs.

In the entire architecture, the longest distance between two registers is called the critical path (CP). The maximum frequency at which a design can operate is therefore computed from the length of the CP. Indeed, the time between two ticks should be long enough for a every signals to exit their registers and to be stored in the next. Hence, the shorter the CP is, the higher the frequency can be. The commonly used method to increase the frequency is to cut the CP in parts by inserting registers. This delays the output of the design by as many cycles as there are registers on the CP. The frequency is then increased at the cost of a greater latency and a higher register usage. This process, called pipelining, makes debugging even more difficult as an error can now also be due to wrongly timed events.

In order to ease the debugging of HDL designs, subsets of higher-level languages can be used as domain specific languages (DSLs) to generate faithful HDL. This allows to describe and simulate a component using the higher-level language environment. Examples of open-source DSLs are MyHDL [29], for using the Python environment or Chisel [30], for the Scala environment.

A component can be distributed as an intellectual property (IP). Thus, the IP is characterized by its maximum frequency (with an impact on its latency and register cost). When integrating it into an existing design, the IP may:

- hold the CP as its maximum frequency is not high enough
- delay the result and use too much registers as the maximum frequency achievable is too high

For optimized designs, it is therefore better to write a specific IP for the clock frequency targeted by the current design. Some techniques allow for automatic retiming of circuits [31, 32, 33]. However, in our case, when designing custom arithmetic operators, not only the timing of the circuit must be customizable but also the sizes of the datapath.

HDL generators Most arithmetic operators can be customized in input and output sizes. Operators can be described as parametrized HLS, but the complexity of the code

scales poorly with the number of parameters. Therefore, most arithmetic IPs comes from HDL generators rather than from standard IP collections.

Commercial tools such as Matlab HDL coder allow users to generate portable Verilog or VHDL from Matlab functions and Simulink models. For targeting specific FPGAs, vendors provide their own tools such as Xilinx system generator or Intel DSP Builder.

One of the most notorious open-source HDL generator for arithmetic cores is FloPoCo [34]. It generates custom arithmetic IPs that can achieve user specified frequency. The main focus of FloPoCo is to generate circuits that compute functions at a user specified accuracy without computing useless bits. The IP's datapath is tailored to the application, and the pipeline depth to the required performance.

In most cases, the IP requires control signals, it will have to be connected to a finite state machine (FSM). This FSM also has to be described using a HDL. However, multiple tools allow for HDL generation from a graphical design of the FSM.

High-Level Synthesis (HLS)

Generating IPs and connecting them together increases productivity compared to writing everything using a HDL by hand. However, in most cases, development and debugging time is still slower than high-level programming. HLS tools [35], which is the process of transforming a high-level description to an HDL (Figure 2.13, left), tries to fill that gap in productivity. The most popular commercial tools are Xilinx Vivado HLS, Intel HLS, CatapultC where the most popular academic tools are LegUp [36], Bambu [37], GAUT [38], AUGH [39], among others [35].

Programmers are offered some high-level input languages such as C/C++, SystemC, C#, OpenCL to write a program. Compared to using a HDL, the component is described by its behavior instead of its implementation. The users benefit from these high-level languages compilers for fast simulation and debugging tools for more traditional debugging. However, only a subset of such languages can be used. For example, dynamic memory allocation, access to standard input/output streams generally can't be synthesized.

HLS tools automatically insert registers in the designs to reach the clock target, and generate the corresponding IP. Connection to specific FPGAs pins are also automated by such tools, as the target FPGA is known at compile time. The FSM required for the design is then included in the IP as derived from the program control flow. Therefore it makes it easier for one to describe complex circuits using standard high-level languages constructions.

Specific coding style HLS tools require a specific coding style. In fact, two functionally equivalent programs can result in designs with very different resource usage/latency. However, this specific coding style is described in some best practice guides [40, 41, 42].

A first example illustrates how to save resources on a simple code example (Figure 2.14, left). In that code snippet, the component takes four integer inputs and a selection boolean. Depending on that boolean, the component will output the product result of either `a` and `b`, or `c` and `d`. Synthesis results of this component reveal that two multiplier are generated and one multiplexer (Vivado 2018.3) as illustrated by Figure 2.14 (right). This result is suboptimal as the multipliers require more resource and a longer critical path than the multiplexers. A simple fix to that code is given in Figure 2.15 where selecting

```
int component(int a, int b, int c,
              int d, bool select){
        int result;
        if(select){
                result = a*b;
        }
        else{
                result = c*d;
        }
        return result;
}
```

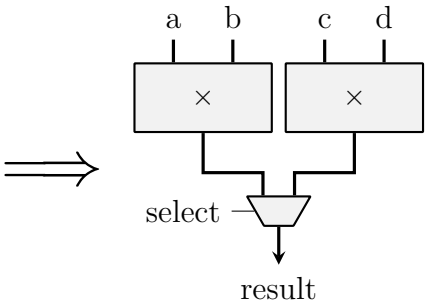


Figure 2.14: Suboptimal resource utilisation of a HLS component written in C/C++.

```
int component(int a, int b, int c,
              int d, bool select){
        int operand1, operand2;
        if(select){
                operand1 = a;
                operand2 = b;
        }
        else{
                operand1 = c;
                operand2 = d;
        }
        return operand1*operand2;
}
```

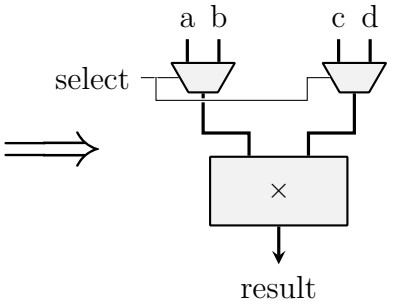


Figure 2.15: Corrected C/C++ component from Figure 2.14.

the operands first and then performing the multiplication (left) produces the expected hardware (right).

A second example illustrates how to improve the latency of a computation loop. The code snippet provided in Figure 2.16 computes the set of products from arrays `a` and `b`, storing the results in array `c`. If we consider that one iteration of the loop requires 3 steps/cycle at a given frequency, the resulting scheduling of such a component will be the one from Figure 2.17 (left). Indeed, iteration 0 will occupy the resources from the first step of the loop at cycle 0, then it will move to the resource of loop step 2 at cycle 1 and

```
void component(int a[SIZE], int b[SIZE], int c[SIZE]){
        for(int i=0; i<SIZE; i++){
                c[i] = a[i]*b[i];
        }
}
```

Figure 2.16: Example of a HLS component performing a computation loop written in C/C++.

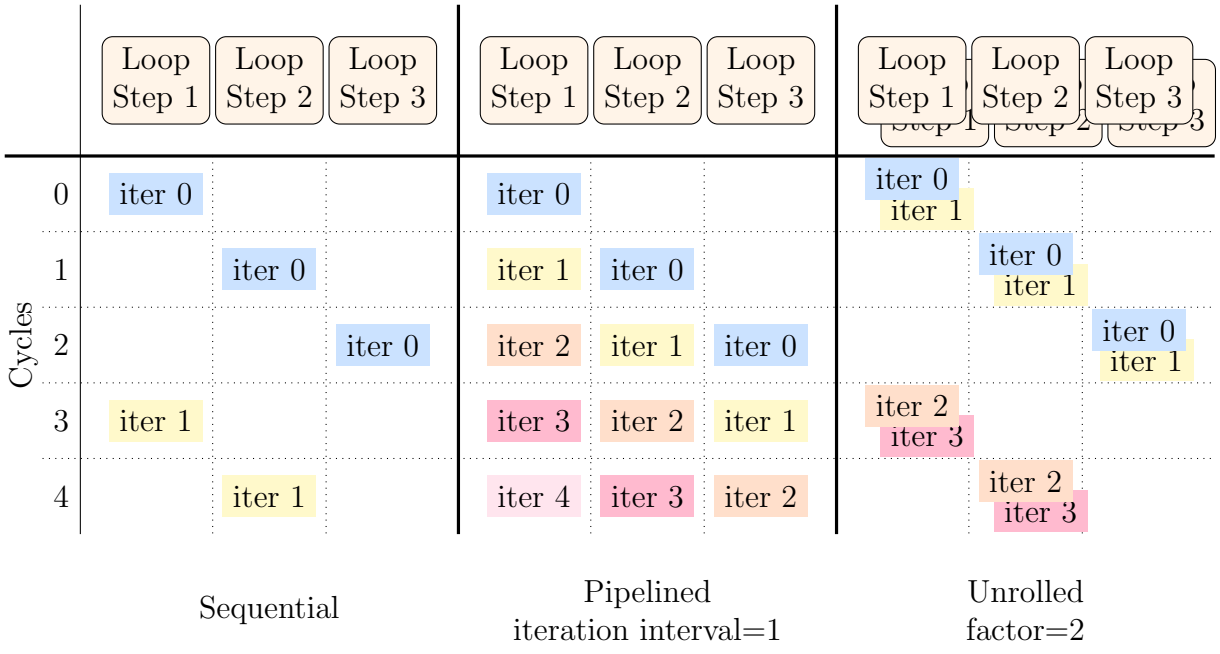


Figure 2.17: Different scheduling for the computation loop from Figure 2.16.

so on. The iteration 1 will only start when the computation of iteration 0 is over. In our case, the loop instructions are independent and this schedule can be improved (both its throughput and resource activity) by using another scheduling policy.

HLS compilers can be directed towards a specific resource allocation, loop schedule or interface by adding pragmas. These compiler directives will only be used for generating HDL and won't affect simulation. In the code example of Figure 2.16, one can specify the HLS compiler to pipeline the loop. Multiple loop iterations will interleave and share the hardware resources (as illustrated in Figure 2.17, middle). In this case, the loop iterations

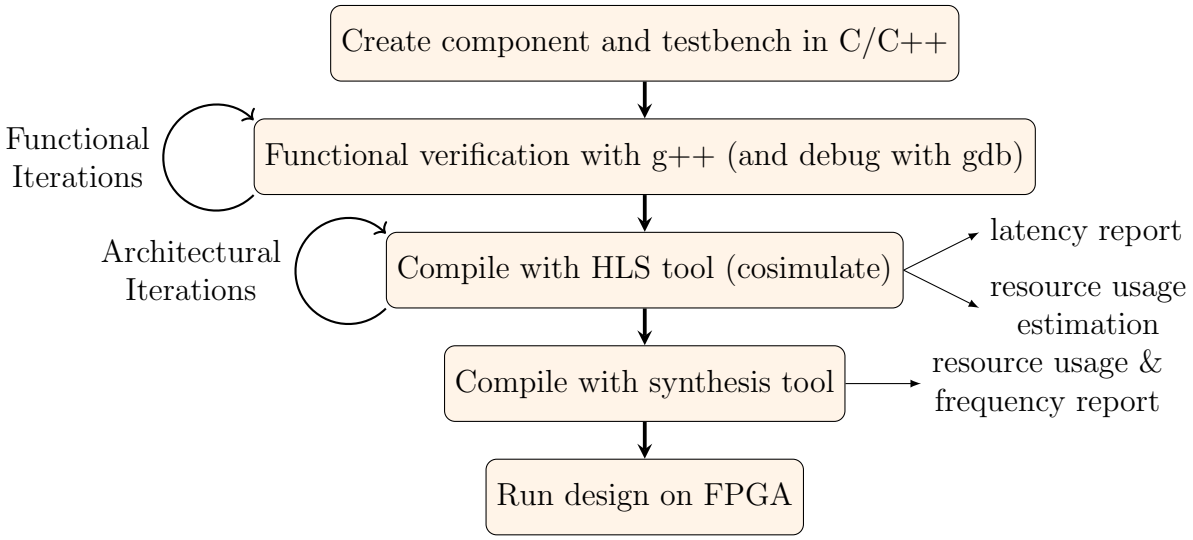


Figure 2.18: HLS design flow example using C/C++.

are all independent, hence a new iteration can be started each cycle. This start distance is called *iteration interval (II)*. Alternatively, one can choose to duplicate the resources in order to execute several instructions in parallel (shown in Figure 2.17, right); in that case unrolling by a factor 2.

The complete HLS design flow is described in Figure 2.18. The user creates a component using the high-level language and debugs it using high-level debuggers. Once the component has the expected behavior, it can be compiled by the HLS compiler. The latter will provide resource estimations and latency reports, according to which the programmer can iterate to improve his design. The generated HDL can then be verified using cosimulation before finally using synthesis tools.

IPs integration

Once an IP is ready, whether it has been written by hand, generated using HDL generators or through HLS, it needs to be connected to the outside world. Its inputs and outputs must be connected to the correct FPGA pins. However, each specific chip has different functionalities and connections. Traditional flows then require deep FPGA knowledge and hand-writing HDLs.

Recent development environment such as Intel FPGA SDK for OpenCL or Xilinx SDAccel provide a complete abstraction of the FPGA connections. Indeed, a programmer can program a FPGA only by writing in a high-level language, the tools will then manage pin connections and data-transfers.

Chapter 3

Bridging high-level synthesis and application specific arithmetic

This Chapter results from a joint collaboration with Steven Derrien and Victor Lezaud.

High-level synthesis tools rely heavily on compiler optimizations [43, 44, 45]. As most of these optimizations were designed for standard CPUs, it is relevant to question if they make sense in an FPGA context. It is also relevant to attempt to identify new optimizations that were not investigated previously because they make sense only in this FPGA context. This is the main objective of this Chapter, with a focus on arithmetic-related optimizations.

Consider for example the integer multiplication by a constant. Figure 3.1 implements a simple integer multiplication by 7. Figure 3.2 shows the assembly code of Figure 3.1, when compiled with gcc 7.4.0 without any particular optimization flag. We can see that the multiplication by 7 has been transformed by the compiler into a *shift and add* algorithm: $7x = 8x - x = x \cdot 2^3 - x$ where the multiplication by 2^3 is a simple shift left by 3 bits (this multiplication by 8 may also be implemented by the `lea` instruction in a slightly less obvious way, and this is what happens, both on GCC or Clang/LLVM, when using -O2 optimization).

As a consequence, the architecture produced by HLS tools based on GCC or Clang/LLVM will implement this algorithm. This optimization makes even more sense in HLS, since the constant shifts reduce to wires and therefore cost nothing. Indeed, the synthesis of Figure 3.1 in Vivado HLS reports 32 LUTs, the cost of one addition. Experiments with Vivado HLS (based on Clang/LLVM) and Intel HLS (based on GCC) show that for all the constant multiplications that can be implemented as an addition, these tools instantiate an adder instead of a multiplier.

```
int mul7(int x){
  return x*7;
}
```

Figure 3.1: C code.

```
(...)
       a: 89 d0       mov  %edx,%eax
       c: c1 e0 03    shl  $0x3,%eax
       f: 29 d0       sub  %edx,%eax
(...)
```

Figure 3.2: Objdump of Figure 3.1 when compiled with gcc.

```
int mul2228241 ( int x ){
  return x*2228241;
}
```

Figure 3.3: C code.

```
(...)
        10: ... imul $0x220011,%edi,%eax
        16: ... retq
(...)
```

Figure 3.4: Objdump of Figure 3.3 compiled with Clang/LLVM -O2.

```
int mul2228241 ( int x ){
  int t = (x<<4) + x;
  return  (t<<17) + t;
}
```

Figure 3.5: C code using a shift-and-add algorithm.

Now consider the multiplication by another constant in Figure 3.3. On this example, we observe that Clang/LLVM x86 backend keeps the operation as a multiplication.

Indeed, the synthesis of this operator by Vivado HLS on a Kintex reports 2 LUTs and 2 DSPs, which are the resources needed to implement a 32-bit multiplier.

However, although the constant looks more complex, it barely is: the multiplication by 2228241 can be implemented in two additions only if one remarks that $2228241 = 17 \cdot 2^{17} + 17$: first compute $t = 17x = x \cdot 2^4 + x$ (one addition), then compute $2228241x = t \cdot 2^{17} + t$ (another addition). Still, neither Clang/LLVM nor GCC use a shift-and-add in this case. The rationale could be the following: the cost of one addition will always be lower than or equal to the cost of a multiplication, whatever the processor, so replacing one multiplication with one addition is always a win. Conversely, it may happen on some (if not most) processors that the cost of two additions and two shifts is higher than the cost of one multiplication.

Is this true in an HLS context? The best architecture for this multiplication, achieved by the C program of Figure 3.5, consists of two adders: one that computes the 32 lower bits of $t = 17x = x \cdot 2^4 + x$ (and should cost only 28 LUTs, since the lower 4 bits are those of x); one that computes the 32 lower bits of $t \cdot 2^{17} + t$, and should cost 32-17=15 LUTs, for the same reason (the 17 lower bits are those of t). The total cost should be 43 LUTs.

For this program, Vivado HLS indeed reports 46 LUTs, very close to the predicted 43 (and not much higher than the cost of the multiplication by 7).

In summary, what we observe here is that the arithmetic optimization has been completely delegated to underlying compiler's x86 backends, and we have a case here for enabling further optimizations. Indeed, hardware constant multiplication has been the subject of much research [46, 47, 48, 49, 50, 51], some of which is specific to FPGAs [12, 52, 53, 14].

The broader objective of the present work is to list similar opportunities of hardware-specific arithmetic optimizations that are currently unexploited, and demonstrate their effectiveness. We classify these optimizations in two broad classes.

In Section 3.1, we discuss optimization opportunities that strictly respect the semantic of the original program. The previous multiplications by constants examples belong to this class, we also discuss divisions by constants, and we add in this Section a few floating-point

optimizations that make sense only in a hardware context. This Section should be perfectly uncontroversial: all optimizations in this class should be available in an HLS flow as soon as they improve some metric of performance. The only reason it is not yet the case is that the field of HLS is still relatively young.

The second class, discussed in Section 3.2 is more controversial and forward-looking. It includes optimizations that relax (and we argue, only for the better) the constraint of preserving the program semantics. In this Section, we assume that the programmer who used floating-point data in their programs intended to compute with real numbers, and we consider optimizations that lead to cheaper and faster, but also more accurate hardware. This approach is demonstrated in depth on examples involving floating-point summations and sums-of-products.

In each case, we use a compilation flow illustrated by Figure 3.6, that involves one or several source-to-source transformations using the GeCoS framework [54] to improve the generated designs. Source-to-source compilers are very convenient in an HLS context, since they can be used as optimization front-ends on top of closed-source commercial tools.

Finally, we discuss in Section 3.3 what we believe HLS tools should evolve to.

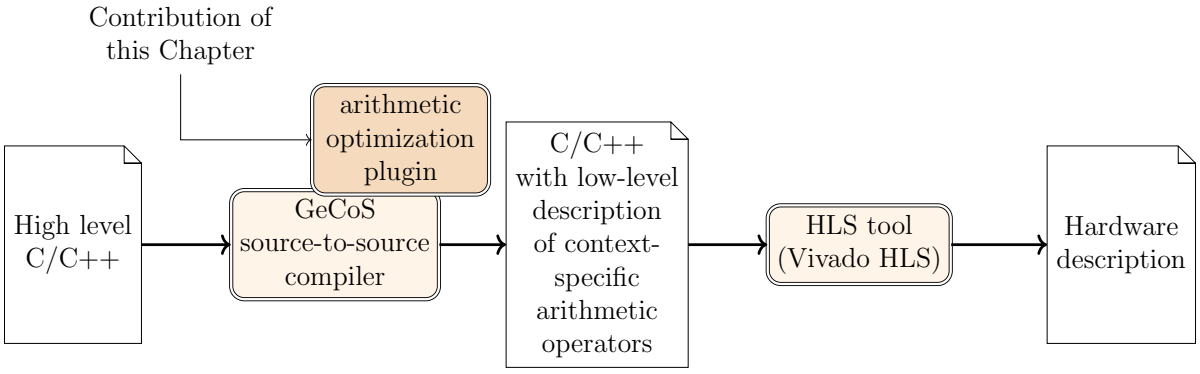


Figure 3.6: The proposed compilation flow.

3.1 Optimization examples that do not change the program semantic

The arithmetic optimizations that fit in this Section go well beyond the constant multiplications. In particular, there are opportunities of floating-point optimizations in FPGAs that are more subtle than operator specialization.

3.1.1 Floating-point corner-case optimization

Computing systems follow the IEEE-754 standard on floating-point arithmetic, which was introduced to normalize computations across different CPUs. Based on this standard, the C standard prevents compilers from performing some floating-point optimizations. Here are some examples that can be found in the C11 standard [55]:

- x/x and 1.0 are not equivalent if x can be zero, infinite, or NaN (in which case the value of x/x is NaN).

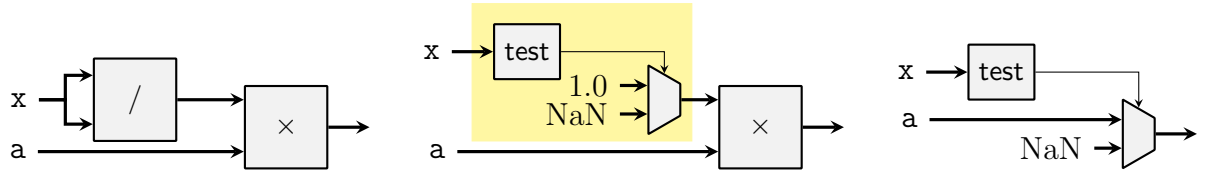
- $x - y$ and $-(y - x)$ are not equivalent because $1.0 - 1.0$ is $+0$ but $-(1.0 - 1.0)$ is -0 (in the default rounding direction).
- $x - x$ and 0 are not equivalent if x is a NaN or infinite.
- $0 \times x$ and 0 are not equivalent if x is a NaN, infinite, or -0.
- $x + 0$ and x are not equivalent if x is -0, because (-0) + (+0), in the default rounding mode (to the nearest), yields +0, not -0.
- $0 - x$ and $-x$ are not equivalent if x is +0, because -(+0) yields -0, but 0 - (+0) yields +0.

Of course, programmers usually don't write x/x or $x + 0$ in their code. However, other optimization steps, such as code hoisting, or procedure specialization and cloning, may lead to such situations: their optimization is therefore relevant in the context of a global optimizing compiler [43].

Let us consider the first example (the others are similar): A compiler is not allowed to replace x/x with 1.0 unless it is able to prove that x will never be zero, infinity or NaN. This is true for HLS as well as for standard compilers. However, it could replace x/x with something like `(is_zero(x)||is_infty(x)||is_nan(x))?NaN:1.0;`. This is, to our knowledge, not implemented. The reason is again probably that in software, the test on `x` becomes more expensive than the division.

However, if implemented in hardware, this test is quite cheap: it consists in detecting if the exponent bits are all zeroes (which capture the 0 case) or all ones (which captures both infinity and NaN cases). The exponent is only 8 bits for single precision and 11 bits for double-precision.

In an FPGA context, it therefore makes perfect sense to replace x/x (Figure 3.7a) with an extremely specialized divider depicted on Figure 3.7b. Furthermore, the two possible values are interesting to propagate further (1.0 because it is absorbed by multiplication, NaN because it is extremely contagious). Therefore, this optimization step enables further ones, where the multiplexer will be pushed down the computation, as illustrated by Figure 3.7c.



(a) Original graph (b) Divider specialization (c) Multiplier specialization

Figure 3.7: Optimization opportunities for floating-point $x/x * a$.

Note that this figure replaces `1.0*x` by `x`: this is a valid floating-point optimization, in the sense that it is valid even if x is a signed zero, an infinity or a NaN.

Occurrences of $x - x$, $0 \times x$, $x + 0$, $0 - x$ can similarly be replaced with a multiplexer and very little logic, and may similarly enable further optimizations.

Since these arithmetic optimizations are expected to be triggered by optimizations (procedure specialization) and trigger further optimizations (conditional constant propagation), they need to be implemented and evaluated within an optimizing compiler.

```
int div7(int in){
  return in/7;
}
```

Figure 3.8: C code.

```
(...)
        0: ... movslq %edi,%rax
        3: ... imul $0xffffffff92492493,%rax,%rcx
        a: ... shr $0x20,%rcx
        e: ... add %ecx,%eax
       10: ... mov %eax,%ecx
       12: ... shr $0x1f,%ecx
       15: ... sar $0x2,%eax
       18: ... add %ecx,%eax
       1a: ... retq
```

Figure 3.9: Objdump of Figure 3.8 when compiled with Clang -O2.

The source-to-source flow depicted on Figure 3.6 is ill-suited to studying such cascaded optimizations. Furthermore, the multiple conditional constant propagation that transforms Figure 3.7b into Figure 3.7c is probably not implemented yet, since it doesn't make much sense in software. This evaluation is therefore left out of the scope of the present work.

In the following, we focus on FPGA-specific semantic-preserving optimizations which will not trigger further optimizations.

3.1.2 Integer multiplication by a constant

Multiplication by a constant has already been mentioned in introduction. We just refer to the rich existing literature on the subject [12, 46, 52, 47, 48, 49, 50, 53, 51, 14]. These are mostly academic works, but backend tools already embed some of it, so this optimization could be the first to arrive. An issue is that its relevance, in the big picture of a complete application, is not trivial: Replacing DSP resources with logic resources is an optimization only in a design that is more DSP-intensive than logic-intensive. Besides, as soon as a logic-based constant multiplier requires more than a handful of additions, it may entail more pressure on the routing resources as well. Discussing this trade-off in detail in the context of an application is out of scope of the present work.

3.1.3 Integer division by small constants

Integer division by a constant adds one more layer of optimization opportunities: In some cases, as illustrated by Figure 3.8 and Figure 3.9, a compiler is able to transform this division into a multiplication by a (suitably rounded) reciprocal. This then triggers the previous optimization of a constant multiplier. Actually, one may observe that on this example that the constant $1/7$ has the periodic pattern $1001001001001001001001001_2$ (hidden in the hexadecimal pattern 924_{16} in Figure 3.9). This enables a specific optimization of the shift-and-add constant multiplication algorithm [56].

Table 3.1 shows synthesis results on the two FPGA mainstream HLS flows. Intel HLS reports usage of MLABs which are the aggregation of several ALMs to emulate a larger LUT. The timing constraint was set to 100 MHz, however this factor is not important here as it does not change the structure of the generated operators. The goal here is to

Table 3.1: Synthesis results of 32-bit integer dividers with Vivado HLS for Kintex 7 (a), and Intel HLS for Arria 10 (b).

(a) Vivado HLS

Value	LUTs	Regs.	DSPs	SRLs
x	*235*	*295*	*0*	*1*
1	0	0	0	0
2	94	0	0	0
3	142	113	4	9
4	94	0	0	0
5	142	113	4	9
6	163	103	4	9
7	142	111	4	9
8	92	0	0	0
9	142	114	4	9

(b) Intel HLS

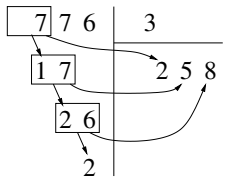
ALMs	FFs	RAMs	DSPs	MLABs
625	*638*	*4*	*10*	*9*
2	3	0	0	0
18	3	0	0	0
121	62	0	0	2
18	3	0	0	0
119.5	74	0	0	2
109.5	59	0	0	2
122	75	0	0	2
18	3	0	0	0
151.5	63	0	0	2

observe the optimizations performed (or not) by the tools. Here is what we can infer from this table:

- The generic divider (Value=x) is based on Xilinx on a shift-and-add algorithm, while on Intel a polynomial approach is used [57] that consumes multiplier and DSP resources.
- Both tools correctly optimize the division by a power of two, converting it into a shift.
- Division by non-power of two integers is implemented by a multiplication by the inverse on Xilinx (it consumes DSP blocks). On Intel, this multiplication is further optimized as a logic-only operation.

For the division of an integer by a very small constant, the best alternative is the algorithm described in [13]. It is based on the decimal *paper-and-pencil* algorithm illustrated in Figure 3.10. Figure 3.11 describes an unrolled architecture for a binary-friendly variant of this algorithm. There, the input X is written in hexadecimal (each 4-bit word X_i is an hexadecimal digit). The quotient bits come out in hexadecimal. The remainder of the division by 3 is always between 0 and 2, therefore fits on 2 bits. Each LUT on the Figure therefore stores the quotient Q_i (between 0_h and F_h) and the remainder R_i (between 0_h and 2_h) of the division by 3 of a number $R_{i+1}X_i$. On a recent LUT-based FPGA, each 6-input, 6-output LUT of Figure 3.11 consumes exactly 6 FPGA LUTs: This architecture is very well suited to FPGAs.

Table 3.2 compares the performance of the division of a 64-bit integer by a small constant, when left to Vivado HLS (left part), and when first replaced by an HLS description of the architecture of Figure 3.11 by a source-to-source transformation (right part of the table). The results were obtained using Vivado HLS 2016.3 targeting a Kintex 7 (part xc7k160tfbg484-1) at 330MHz. For constants smaller than 9, all the metrics (logic resources, DSP, latency and frequency) are improved by this transformation. As the constant grows larger, the latency degrades and the resource consumption increases:



We first compute the Euclidean division of 7 by 3. This gives the first digit of the quotient, here 2, and the remainder is 1. In other words $7 = 3 \times 2 + 1$. The second step divides 77 by 3 by first rewriting $77 = 70 + 7 = 3 \times 20 + 10 + 7$: dividing 17 by 3 gives $17 = 3 \times 5 + 2$. The third steps rewrites $776 = 770 + 6 = 250 + 20 + 6$ where $26 = 3 \times 8 + 2$, hence $776 = 3 \times 258 + 2$.
The only computation in each step is the Euclidean division by 3 of of a number between 0 and 29: it can be pre-computed for these 30 cases and stored in a LUT.

Figure 3.10: Illustrative example: division by 3 in decimal.

for the division by 9 we already have a worst latency and frequency than the default multiplication-based implementation, but still with much less resources.

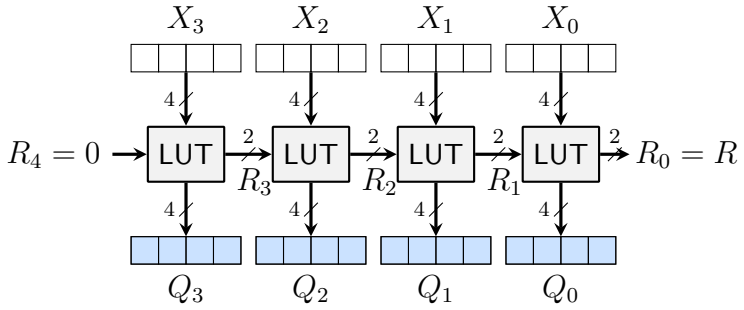


Figure 3.11: Unrolled architecture for a LUT-based division by a constant with a 16-bit input and LUTs with 4 input bits.

Table 3.2: Synthesis results of 64-bit integer constant divisors using Vivado HLS for Kintex 7.

	(a) C division				(b) [13] division in HLS		
Value	LUTs	reg.	DSPs	Cycles @ Freq	LUTs	reg.	Cycles @ Freq
x	*8831*	*8606*	*0*	*68@293MHz*	*NA*	*NA*	*NA*
2	194	193	0	3@467MHz	0	0	1 @ 1488MHz
3	931	966	16	23@373MHz	62	127	17 @ 403MHz
4	191	190	0	3@444MHz	0	0	1 @ 1488MHz
5	925	962	16	23@364MHz	107	152	23 @ 377MHz
6	927	964	16	23@363MHz	72	156	17 @ 380MHz
7	923	956	16	23@355MHz	107	152	23 @ 377MHz
8	189	187	0	3@449MHz	0	0	1 @ 1488MHz
9	929	961	16	23@356MHz	100	193	33 @ 351MHz

Table 3.3: Synthesis results of single-precision floating-point constant multipliers/dividers using Vivado HLS (Kintex 7) (a) and Intel HLS (Arria 10) (b) targeting 100 MHz.

(a) Vivado HLS

	Value	LUTs	Regs.	DSPs	SRLs
Mult	*x*	*86*	*99*	*3*	*0*
	1.0	0	0	0	0
	2.0	70	67	3	0
	3.0	67	70	3	0
	4.0	71	67	3	0
	5.0	71	67	3	0
Div	*x*	*780*	*392*	*0*	*25*
	1.0	0	0	0	0
	2.0	75	67	3	0
	3.0	740	250	0	25
	4.0	75	67	3	0
	5.0	739	250	0	25

(b) Intel HLS

	ALMs	FFs	RAMs	DSPs	MLABs
Mult	*43*	*36*	*0*	*1*	*2*
	2	3	0	0	0
	69.5	21	0	0	2
	102	20	0	0	2
	69	20	0	0	2
	108	21	0	0	2
Div	*311.5*	*634*	*3*	*4*	*7*
	2	3	0	0	0
	72	23	0	0	2
	331.5	500	3	4	9
	71.5	22	0	0	2
	322.5	504	3	4	9

Table 3.4: Synthesis results of floating-point constant divisors for single (a) and double (b) precision that implements [13] using Vivado HLS for Kintex 7.

(a) float

Value	LUTs	reg.	Cycles @ Freq
x	*784*	*1446*	*30 @ 330MHz*
2.0	34	0	1 @ 458MHz
3.0	152	130	10 @ 314MHz
4.0	35	0	1 @ 467MHz
5.0	149	151	12 @ 307MHz
6.0	126	126	10 @ 325MHz
7.0	151	151	12 @ 270MHz
8.0	55	0	1 @ 397MHz
9.0	180	161	17 @ 278MHz
10.0	261	162	13 @ 206MHz
11.0	189	161	17 @ 276MHz

(b) double

LUTs	reg.	Cycles @ Freq
3244	*3178*	*31 @ 188MHz*
77	68	2 @ 539MHz
608	310	17 @ 182MHz
179	70	2 @ 422MHz
606	319	22 @ 182MHz
604	311	17 @ 177MHz
624	319	22 @ 177MHz
208	68	2 @ 453MHz
628	333	32 @ 194MHz
609	320	22 @ 180MHz
636	333	32 @ 189MHz

3.1.4 Floating-point multiplications and divisions by small constants

As illustrated by Table 3.3, there are even fewer optimizations for floating-point multiplications and divisions by constants than when using integer arithmetic.

- Both Vivado HLS and Intel HLS are able to remove the constant multiplication and division by 1.0 (unsurprisingly, since it is a valid simplification in software compilers).
- Intel HLS seems to optimize constant multiplications (it never requires a DSP). Vivado HLS, on the other hand, doesn't even optimizes multiplications by 2 or a power of 2. This class of operations should resume to an addition on the exponents, and specific overflow/underflow logic.
- Both tools use a specific optimization when dividing by a power of two. This can easily be explained by looking at the assembly code generated by GCC or Clang/LLVM in such cases: both compilers will transform a division by 4.0 into a multiplication by 0.25, which is bit-for-bit equivalent, and much faster on most processors.
- Both tools use a standard divider for constants that are not a power of 2, with minor resource reductions thanks to the logic optimizer.

Again we may question the relevance of these choices on FPGAs. It is indeed possible to design floating-point versions of both constant multiplications [58] and constant divisions [13] that are bit-for-bit IEEE correctly rounded ones. For instance, in the case of the division, the remainder R that is output by Figure 3.11 can be used to determine the proper rounding of the significand quotient (for the full details, see [13]).

As we expect constant multiplications to be properly supported soon (it seems to be already the case on Intel HLS), we focus our evaluation on divisions by constants. Table 3.4 provides synthesis results of Vivado HLS C++ generated operators for floating-point divisions by small constants. The standard floating-point division is also given for comparison purposes, since Table 3.3 shows that it is the default architecture. All these operators can be more/less deeply pipelined to achieve higher/lower frequencies at the expense of latency and registers: we attempt to achieve a frequency comparable to that of the standard divider.

Each optimized constant divider uses fewer resources (up to 12 times) and has a lower latency (up to 3 times) for a comparable frequency. When dividing by a power of two, the cost of the custom divider is virtually nothing (again it resumes to an operation on the exponents).

3.1.5 Evaluation in context

We implemented a C-to-C source-to-source transformation that detects floating-point multiplications and divisions by constants in the source code, and replaces it by a custom operator that is bit-for-bit equivalent. This transformation was implemented as a plug-in within the open source source-to-source GeCoS compiler framework [54], as per Figure 3.6.

This work was then evaluated on the Polybench benchmark suite [59]. It contains several C programs that fit the polyhedral model. The focus here is on the stencil codes of this benchmark suite. Most of them contains a division by a small constant. Indeed, out of the 6 stencil codes, 5 were well suited for our transformations. The *Jacobi-1d* benchmark contains two divisions by 3; *Jacobi-2d* contains two divisions by 5; *Seidel-2d* contains a division by 9; *Fdtd-2d* contains two divisions by 2 and a multiplication by 0.7; finally, *Heat-3d* contains six divisions by 8 and six multiplications by 2.

Table 3.5 compares the synthesis results obtained from Vivado HLS.

Table 3.5: Synthesis results of benchmarks before and after transformations.

Benchmark	Type	LUTs	regs.	DSPs	Cycles @ Freq.
Fdtd-2d	Orig.	4741	6262	17	153G @ 320MHz
	Trans.	2819	4628	17	11G @ 345MHz
heat-3d	Orig.	3744	6118	31	193G @ 341MHz
	Trans.	4886	6984	17	147G @ 331MHz
Jacobi-1d	Orig.	4221	4985	3	185M @ 354MHz
	Trans.	2006	2971	3	137M @ 348MHz
Seidel-2d	Orig.	4514	5481	9	213G @ 358MHz
	Trans.	2328	3491	9	183G @ 337MHz
Jacobi-2d	Orig.	4335	5157	6	373G @ 355MHz
	Trans.	1806	2861	6	357G @ 336MHz

- using the original C code, targeting the maximum frequency achievable, and
- using the code after transformation by our GeCoS plug-in.

Each benchmark benefits from the transformations. Latency is improved up to 12 times for similar frequencies. The *Heat-3d* benchmark trades a bit more LUTs and registers for a lot less DSPs. In all other cases, LUTs, registers and DSPs usage is reduced.

The benefit of the transformations in terms of cycles differ from one benchmark to another. The best improvements are achieved when the transformed operator is in the critical path of an inner loop.

3.2 Optimization examples that change the program semantic

From a compiler point of view, the previous transformations were straightforward and semantic preserving.

The case study in this Section is a more complex program transformation that applies to floating-point reductions. The use of custom formats, driven by user-specified accuracy allows to tighten loop carried dependencies. The result of this complex sequence of optimizations cannot be obtained from an operator generator since it involves knowledge of the program behaviour in which the operator is to be instantiated. Before detailing it, we must digress a little on the subtleties of the management of floating-point arithmetic by compilers.

3.2.1 High-level synthesis (HLS) faithful to the floats

Most recent compilers, including HLS ones [60], attempt to follow established standards, in particular C11 and, for floating-point arithmetic, IEEE-754. This brings the huge

Table 3.6: Synthesis results of different accumulators using Vivado HLS for Kintex 7.

	Figure 3.12 (float)	Figure 3.13 (float)	Figure 3.12 (double)	Figure 3.13 (double)	Figure 3.16 (71 bits)	FloPoCo VHDL (71 bits)
LUTs	266	907	801	2193	736	719
DSPs	2	4	3	6	0	0
Latency	700K	142K	700K	142K	100K	100K
Accuracy	17 bits	17 bits	24 bits	24 bits	24 bits	24 bits

```
#define N 100000
float acc = 0;
for(int i=0; i<N; i++){
  acc+=in[i];
}
```

Figure 3.12: Naive reduction.

```
#define N 100000
float acc = 0, tmp1=0, ... , tmp10=0;
for(int i=0; i<N; i+=10){
  tmp1+=in[i];
  ...
  tmp10+=in[i+9];
}
acc=tmp1+...+tmp10;
```

Figure 3.13: Parallel reduction.

advantage of almost bit-exact reproducibility – the hardware will compute exactly the same results as the software. However, it also greatly reduces the freedom of optimization by the compiler. For instance, as floating-point addition is not associative, C11 mandates that code written `a+b+c+d` is executed as `((a+b)+c)+d`, although `(a+b)+(c+d)` have a shorter latency. This also prevents the parallelization of loops implementing reductions. A reduction is an associative computation which reduces a set of input values into a reduction location. Figure 3.12 provides the simplest example of reduction, where `acc` is the reduction location.

The first column of Table 3.6 shows how Vivado HLS synthesizes Figure 3.12 on a Kintex 7 FPGA. The floating-point addition takes 7 cycles, and the adder is only active one cycle out of 7 due to the loop-carried dependency. Figure 3.13 shows a different version of Figure 3.12 that we coded such that Vivado HLS expresses more parallelism. Vivado HLS will not transform Figure 3.12 into Figure 3.13, because they are not semantically equivalent[1] (the floating-point additions are reordered as if they were associative). However, the tool is able to exploit the parallelism in Figure 3.13 (second column of Table 3.6): The main adder is now active at each cycle on a different sub-sum.

Note that Figure 3.13 is only here as an example and might need more logic if N was not a multiple of 10.

3.2.2 Towards HLS faithful to the reals

In the remainder of this Chapter, we adopt a new, non-standard point of view. The latter is to assume that the floating-point C/C++ program is intended to describe a computation

[1] A parallel execution with the sequential semantics is also possible, but very expensive [61].

on real numbers when the user specifies it. In other words, the floats are interpreted as *real numbers* in the initial C/C++, thus recovering the freedom of associativity (among other). Indeed, most programmers will perform the kind of non-bit-exact optimizations illustrated by Figure 3.13 (sometimes assisted by source-to-source compilers or "unsafe" compiler optimizations). In a hardware context, we may also assume they wish they can tailor the precision (hence the cost) to the accuracy requirements of the application – a classical concern in HLS [62, 63]. In this case, a pragma should specify the accuracy of the computation with respect to the exact result. A high-level compiler is then in charge of determining the best way to ensure the prescribed accuracy.

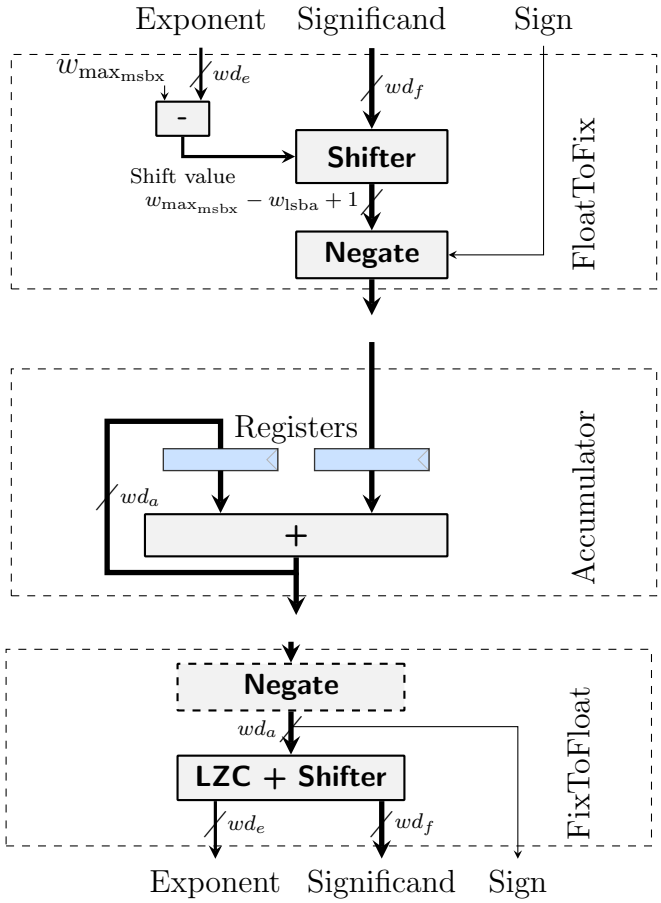


Figure 3.14: The conversion from floating-point to fixed-point (top), the fixed-point accumulation (middle) and the conversion from the fixed-point format to a floating-point (bottom).

3.2.3 The arithmetic side: application-specific accumulator support

The architecture used for this work is based on a more general idea developed by Kulisch. He advocated to augment processors with a very large fixed-point accumulator [64] whose 4288 bits would cover the entire range of double precision floating-point, and then some more: Such an accumulator would remove rounding errors from all the possible floating-point additions and sums-of-products. The added bonus of an operator that makes all

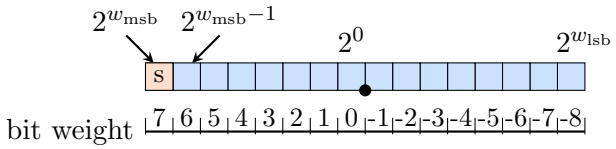


Figure 3.15: The bits of a fixed-point format, here $(w_{\text{msba}}, w_{\text{lsba}}) = (7, -8)$.

additions exact is that addition then becomes associative, since the loss of associativity in floating-point computations is due to rounding.

So far, Kulisch's complete accumulator has proven too costly to appear in mainstream processors. However, in the context of application acceleration with FPGAs, it can be tailored to the application accuracy requirements. Its cost then becomes comparable to classical floating-point operators, although it vastly improves accuracy [65]. This operator can be found in the FloPoCo [66] generator and in Intel DSP Builder Advanced. Its core idea, illustrated on Figure 3.14, is to use a large fixed-point register into which the significands of incoming floating-point summands are shifted (top) then accumulated (middle). A third component (bottom) converts the content of the accumulator back to the floating-point format. The sub-blocks visible on this Figure: shifter, adder, and leading zero counter (lzc); are essentially the building blocks of a classical floating-point adder.

The accumulator used here slightly improves the one offered by FloPoCo [65]:

- It supports subnormal numbers [23].
- In FloPoCo, FloatToFix and Accumulator form a single component, which restricts its application to simple accumulations similar to Figure 3.12. The decomposition in two components of Figure 3.14 enable a generalization to arbitrary summations within a loop, as Section 3.2.4 will show.

Note that we could have implemented any other non-standard operator performing a reduction such as [67, 68].

The parameters of a large accumulator

The main feature of this approach is that the internal fixed-point representation is configurable in order to control accuracy. It has two parameters:

- w_{msba} is the weight of the most significant bit of the accumulator. For example, if $w_{\text{msba}} = 20$, the accumulator can accommodate values up to a magnitude of $2^{20} \approx 10^6$.
- w_{lsba} is the weight of the least significant bit of the accumulator. For example, if $w_{\text{lsba}} = -50$, the accumulator can hold data accurate to $2^{-50} \approx 10^{-15}$.

Such a fixed-point format is illustrated in Figure 3.15.

The accumulator width wd_a is then computed as $w_{\text{msba}} - w_{\text{lsba}} + 1$, for instance 71 bits in the previous example. 71 bits represents a wide range and high accuracy, and still additions on this format will have one-cycle latency for practical frequencies on recent FPGAs. For comparison, for the same frequency, a floating-point adder has a latency of 7 to 10 cycles, depending on the target.

```
#define N 100000
float acc = 0;
ap_int<68> long_accumulator = 0;
for(int i = 0; i < N; i++) {
  long_accumulator += FloatToFix(in[i]);
}
acc = FixToFloat(long_accumulator);
```

Figure 3.16: Sum of `floats` using the large fixed-point accumulator.

As the accumulator width grows, the carry chain of the addition will limit the maximum frequency of the design. This can be improved thanks to partial carry save [65]. The extreme case would be to generate a complete Kulisch accumulator when the latter cannot be tailored using the partial carry save technique. Such a case is studied in details in Chapter 4.

Implementation within a HLS tool

This accumulator has been implemented in C/C++, using arbitrary-precision integer types (`ap_int`). The leading zero count, bit range selection and other operations are implemented using Vivado HLS built-in functions. For modularity purposes, the `FloatToFix` and `FixToFloat` are wrapped into C/C++ functions (respectively 28 and 22 lines of code). Their calls are inlined to enable HLS optimizations.

Because the internal accumulation is performed on a fixed-point integer representation, the combinational delay between two accumulations is lower compared to a full floating-point addition. HLS tools can take advantage of this delay reduction by more aggressive loop pipelining (with a shorter initiation interval), resulting in a design with a shorter overall latency.

Validation

To evaluate and refine this implementation, we used the program of Figure 3.16, which we compared to Figures 3.12 and 3.13. In the latter, the loop was unrolled by a factor 7, as it is the latency of a floating-point adder on our target FPGA (Kintex 7).

For test data, we use as in Müller et al. [23] the input values `c[i] = (float) cos(i)`, where i is the input array's index. Therefore the accumulation computes $\sum_i$ `c[i]`.

The parameters chosen for the accumulator are:

- $w_{\text{msba}} = 17$. Indeed, as we are adding $\cos(i)$ 100K times, an upper bound is 100K, which can be encoded in 17 bits.
- $w_{\max_{\text{msbx}}} = 1$ as the maximum input value is 1.
- $w_{\text{lsba}} = -50$: the accumulator itself will be accurate to the 50th fractional bit. Note that a `float` input will see its significand rounded by FloatToFix only if its exponent is smaller than 2^{-25}, which is very rare. In other words, this accumulator is much more accurate than the data that is thrown to it.

The results are reported in Table 3.6 for simple and double precision. The Accuracy line of the table reports the number of correct bits of each implementation, after the result has been rounded to a `float`. All the data in this Table was obtained by generating `VHDL` from `C` synthesis using Vivado HLS followed by place and route from Vivado v2015.4, build 1412921. This Table also reports synthesis results for the corresponding FloPoCo-generated VHDL, which doesn't include the array management.

Vivado HLS uses DSPs to implement the shifts in its floating-point adders. Even if the shifts were implemented in LUTs, the first column would remain well below 500 LUTs: it has the best resource usage. However the latency of one iteration is 7 cycles, hence 100K iterations takes 700K cycles. When unrolling the loop, Vivado HLS is using almost 4 times more LUTs for floats, and 3 times more for doubles. The unrolled versions improves latency over naive versions. Nevertheless, the proposed approach gets even better latencies for a reasonable LUT usage. It also achieves maximum accuracy for the `float` format, which caps at 24 bits (the internal representations of the `double`, unrolled `double` and proposed approach have a higher accuracy than 24 bits, but their result is then rounded to a `float`). Finally, our results are very close to FloPoCo ones, both in terms of LUTs usage, DPSs and latency.

Exact floating-point multiplier Using this implementation method, we also created an exact floating-point multiplier with the final rounding removed. The product of two normal floating-point numbers can be written:

$$xy \;=\; (-1)^{s_x+s_y} \;\cdot\; 1.f_x \times 1.f_y \;\cdot\; 2^{e_x+e_y}$$

leading to the exact multiplier architecture of Figure 3.17. The significand product $1.f_x \times 1.f_y$ is a fixed-point number of $2wd_f + 2$ bits with two integer bits before the point. The exponent sum $e_x + e_y$ is an integer on $wd_e + 1$ bits.

As this product format is non-normalized, subnormal management [23] adds very little overhead: if the exponent field of one input is all zeroes, then the implicit bit in the significand is set to 0 (otherwise it is set to 1) and a correction of 1 is added to the exponent sum. This is implemented in the *subnormal detection* boxes of Figure 3.17.

A standard floating-point multiplier requires rounding and normalization logic to convert this exact product to the standard format, including a costly shifter in the case of subnormal input. The above exact multiplier doesn't, and is therefore simpler, faster and cheaper. However, its output is larger.

This function is called `ExactProduct` and represents 44 lines of code. The result significand is twice as large as the input significands (48 bits in single precision). To add it to the large accumulator, the Float-to-Fix block has to be adapted: in the sequel, it is called `ExactProductFloatToFix` (21 lines of code).

3.2.4 The compiler side: source-to-source transformation

The previous Section, as well as previous work [69] has shown that Vivado HLS can be used to synthesize very efficient specialized floating-point operators which rival in quality with those generated by FloPoCo. Our goal is now to study how such optimizations can be automated. More precisely, we aim at automatically optimize Figure 3.12 into Figure 3.16, and generalize this transformation to many more situations.

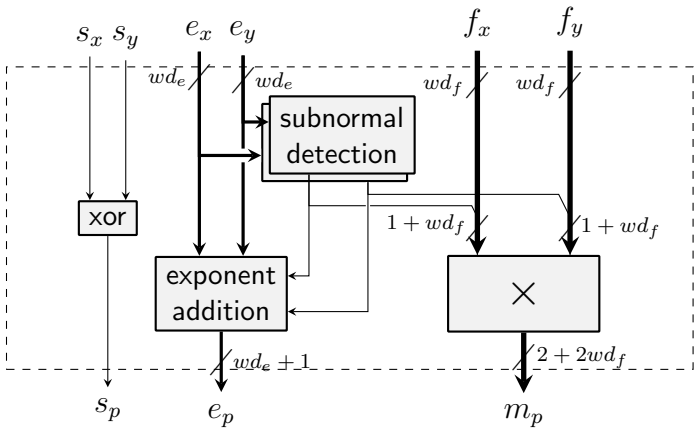


Figure 3.17: Exact floating-point multiplier.

For convenience, this optimization was also developed as a source-to-source transformation implemented within the GeCoS framework.

This part focuses on two computational patterns, namely the accumulation and the sum-of-product. Both are specific instances of the reduction pattern, which can be optimized by many compilers or parallel run-time environments. Reduction patterns are exposed to the compiler/runtime either though user directives (e.g `#pragma reduce` in OpenMP), or automatically inferred using static analysis techniques [70, 71].

As the problem of detecting reductions is not the main focus on this work, our tool uses a straightforward solution to the problem using a combination of user directive and (simple) program analysis. More specifically, the user must identify a target accumulation variable through a `pragma`, and provide additional information such as the dynamic range of the accumulated data along with the target accuracy. In the future, we expect to improve the program analysis, so that the two later parameters could be omitted in some situations. Chapter 4 studies the implementation of complete Kulisch accumulators in an FPGA context. We use complete accumulators as a fall-back strategy when no specification on the input data is given.

We found this approach easier, more general and less invasive than those attempting to convert a whole floating-point program into a fixed-point implementation [22].

Proposed compiler directive

In imperative languages such as `C`, reductions are implemented using `for` or `while` loop constructs. The proposed compiler directive must therefore appear right outside such a construct. Figure 3.18 illustrates its usage on the code of Figure 3.12.

The `pragma` must contain the following information:

- The keyword `FPacc`, which triggers the transformation.
- The name of the variable in which the accumulation is performed, preceded with the keyword `VAR`. In the example, the accumulation variable is `acc`.
- The maximum value that can be reached by the accumulator through the use of the `MaxAcc` keyword. This value is used to determine the weight w_{msba}.

- The desired accuracy of the accumulator using the `epsilon` keyword. This value is used to determine the weight w_{lsba}.
- Optional: The maximum value among all inputs of the accumulator in the `MaxInput` field. This value is used to determine the weight $w_{\text{max}_{\text{msbx}}}$. If this information is not provided, then $w_{\text{max}_{\text{msbx}}}$ is set to w_{msba}.

```
#define N 100000
float accumulation(float in[N]){
  float acc = 0;
  #pragma FPacc VAR=acc MaxAcc=100000.0 epsilon=1E-15 MaxInput=1.0
  for(int i=0; i<N; i++){
    acc+=in[i];
  }
  return acc;
}
```

Figure 3.18: Illustration of the use of a `pragma` for the naive accumulation.

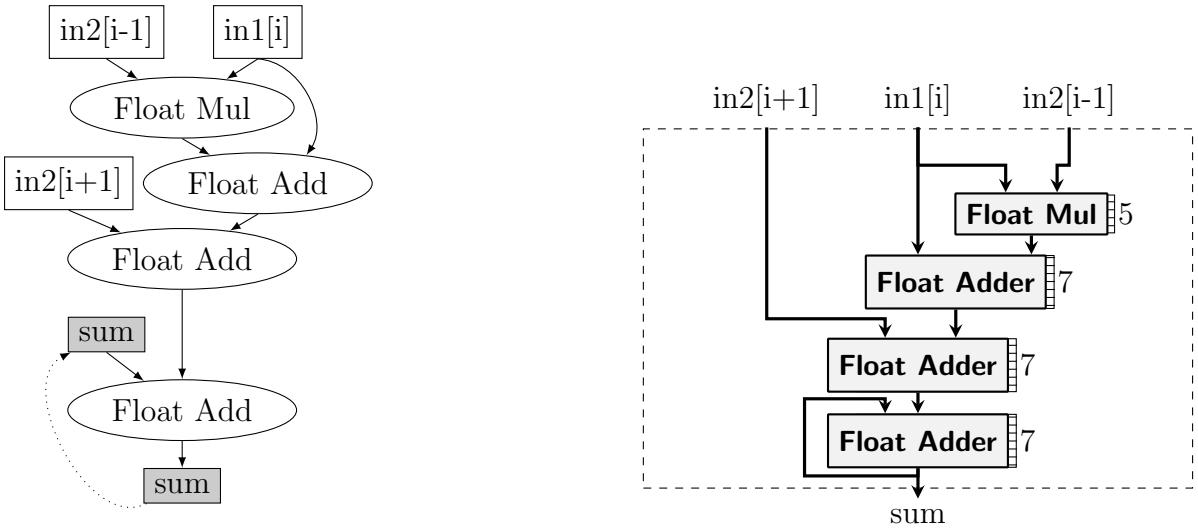
In the case when no size parameters are given, a complete Kulisch accumulator is currently produced. Note that the user can quietly overestimate the maximum value of the accumulator without major impact on area. For instance, overestimating `MaxAcc` by a factor 10 only adds 3 bits to the accumulator width.

```
#define N 100000
float computeSum(float in1[N], float in2[N]){
  float sum = 0;
  #pragma FPacc VAR=sum MaxAcc=300000.0 epsilon=1e-15 MaxInput=3.0
  for (int i=1; i<N-1; i++){
    sum+=in1[i]*in2[i-1];
    sum+=in1[i];
    sum+=in2[i+1];
  }
  return sum;
}
```

Figure 3.19: Simple reduction with multiple accumulation statements.

Proposed code transformation

The proposed transformation operates on the compiler program intermediate representation (IR), and rely on the ability to identify loops constructs and expose def/use relations between instructions of a same basic block in the form of an operation data-flow graph (DFG).



(a) Loop body dataflow graph

(b) Architecture

Figure 3.20: DFG of the loop body from Figure 3.19 (left) and its corresponding architecture (right). Keywords *float mul* and *float add* correspond to floating-point multipliers and adders respectively.

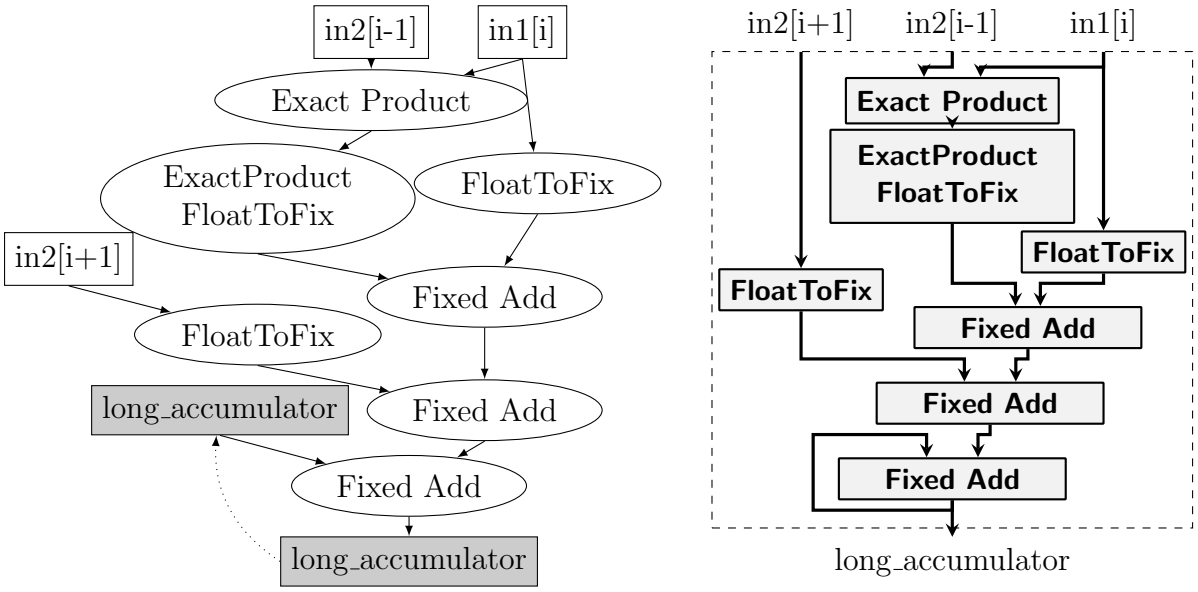
To illustrate the transformation, consider the toy but non-trivial program of Figure 3.19. This program performs a reduction into the variable `sum`, involving both sums and sums-of-product operations. Figure 3.20a shows the operation data-flow graph for the loop body of this program. In this Figure, dotted arrows represent loop-carried dependencies between operations belonging to distinct loop iterations. Such loop-carried dependencies have a very negative impact on the kernel latency as they prevent loop pipelining. For example, when using a pipelined floating-point adder with a seven cycle latency, the HLS tool will schedule a new iteration of the loop at best every seven cycles.

As illustrated in Figure 3.21a, the proposed transformation hoists the floating-point normalization step out of the loop, and performs the accumulation using fixed-point arithmetic. Since integer additions can be implemented with a 1-cycle delay at our target frequency, the HLS tool may now be able to initiate a new iteration every cycle, improving the overall latency by a factor of 7.

The code transformation first identifies all relevant basic blocks (i.e those associated to the `pragma` directive). It then performs a backward traversal of the data-flow graph, starting from a `Float Add` node that writes to the accumulation variable identified by the `#pragma`.

During this traversal, the following actions are performed depending on the visited nodes:

- A node with the summation variable is ignored.
- A `Float Add` node is transformed to an accurate fixed-point adder. The analysis is then recursively launched on that node.
- A `Float Mul` node is replaced with a call to the `ExactProduct` function followed by a call to `ExactProdFloatToFix`.
- Any other node has a call to `FloatToFix` inserted.



(a) Loop body data-flow graph

(b) Architecture

Figure 3.21: DFG of the loop body from Figure 3.19 (left) and its corresponding architecture (right) after transformations.

This algorithm rewrites the DFG from Figure 3.20a into the new DFG shown on Figure 3.21a. In addition, a new basic block containing a call to `FixToFloat` is inserted immediately after the transformed loop, in order to expose the floating-point representation of the results to the remainder of the program.

From there, it is then possible to regenerate the corresponding `C` code (shown in Figure 3.22). As an illustration of the whole process, Figures 3.20b and 3.21b describe the architectures corresponding to the code before and and after the transformation.

Evaluation of the toy example of Figure 3.19

The proposed transformations work on non-trivial examples such as the one represented in Figure 3.19. Table 3.7 shows how resource consumption depends on `epsilon`, all the other parameters being those given in the `pragma` of Figure 3.19. All these versions where synthesized for 100 MHz.

Table 3.7: Comparison between the naive code from Figure 3.19 and its transformed equivalent. All these versions run at 100MHz.

	Naive	Transformed $w_{\text{lsba}} = -14$	Transformed $w_{\text{lsba}} = -20$	Transformed $w_{\text{lsba}} = -50$
LUTs	538	693	824	1400
DSPs	5	2	2	2
Latency	2000K	100 K	100K	100K

```
#include <ap_int.h>
#define N 100000

float computeSum(float in1[N], float in2[N]) {
  float sum = 0;
  ap_int<70> long_accumulator_generated;

  long_accumulator_generated = 0;
  for(int i=1; i < N-1; i++) {
    #pragma HLS PIPELINE II=1
    long_accumulator_generated += FloatToFix(in1[i])
                                + FloatToFix(in1[i] * in2[i - 1])
                                + FloatToFix(in2[i + 1]);
  }
  sum = FixToFloat(long_accumulator_generated);
  return sum;
}
```

Figure 3.22: Transformed code from Figure 3.19.

Compared to the classical IEEE-754 implementation, the transformed code uses more LUTs for fewer DSPs. This is due to Vivado implementing shifters using DSPs within the floating-point IP, but not in the transformed code. In all cases, on this example, the transformed code has its latency reduced by a factor 20.

3.2.5 Evaluation in context

In order to evaluate the relevance of the proposed transformations on real-life programs, we used the EEMBC FPMark benchmark suite [72]. This suite consists of 10 programs. A first result is that half of these programs contain visible accumulations:

- Enhanced Livermore Loops (1/16 kernels contains one accumulation).
- LU Decomposition (multiple accumulations).
- Neural Net (multiple accumulations).
- Fourier Coefficients (one accumulation).
- Black Scholes (one accumulation).

The following of this Chapter focuses on these, and ignores the other half (Fast Fourier Transform, Horner's method, Linpack, ArcTan, Ray Tracer).

Most benchmarks come in single-precision and double-precision versions. We focus here on the single-precision. Double-precision benchmarks lead to the same conclusions.

Benchmarks and accuracy: methodology

Each benchmark comes with a golden reference against which the computed results are compared. As the proposed transformations are controlled by the accuracy, it may happen that the transformed benchmark is less accurate than the original. In this case, it will not pass the benchmark verification test, and rightly so.

A problem is that the transformed code will also fail the test if it is *more* accurate than the original. Indeed, the golden reference is the result of a certain combination of rounding errors using the standard FP formats, which we do not attempt to replicate.

To work around this problem, each benchmark was first transformed into a high-precision version where the accumulation variable is a 10,000-bit floating-point numbers using the MPFR library [73]. We used the result of this highly-accurate version as a "platinum" reference, against which we could measure the accuracy of the benchmark's golden reference. This allowed us to choose our `epsilon` parameter such that the transformed code would be at least as accurate as the golden reference. This way, the `epsilon` of the following results is obtained through profiling. The accuracy of the obtained results are computed as the number of correct bits of the result.

We first present the benchmarks that are improved by our approach before discussing the reasons why we can't prove that the others are.

Table 3.8: Synthesis results of benchmarks before and after transformations.

Benchmark	Type	LUTs	DSPs	Latency	Accuracy
Livermore	Orig.	384	5	80K	11 bits
	Trans.	576	2	20K	13 bits
LU-8	Orig.	809	5	82	8-23 bits
	Trans.	1007	2	17	23 bits
LU-45	Orig.	819	5	452	8-23 bits
	Trans.	1034	2	54	23 bits
Scholes	Orig.	15640	175	N/A	19 bits
	Trans.	15923	175	N/A	23 bits
Fourier	Orig.	34596	64	N/A	6 bits
	Trans.	34681	59	N/A	11 bits

Enhanced Livermore Loops This program contains 16 kernels of loops that compute numerical equations. Among these kernels, there is one that performs a sum-of-product (banded linear equations). This kernel computes 20000 sums-of-products. The values accumulated are pre-computed. This is a perfect candidate for the proposed transformations.

For this benchmark, the optimal accumulation parameters were found as:

```
MaxAcc=50000.0 epsilon=1e-5 MaxInput=22000.0
```

Synthesis results of both codes (before and after transformation) are given in Table 3.8. As in the previous toy examples, latency and accuracy are vastly improved for comparable area.

LU Decomposition and Neural Net Both the LU decomposition and the neural net programs contain multiple nested small accumulations. In the LU decomposition program, an inner loop accumulates between 8 and 45 values. Such accumulations are performed more than 7M times. In the neural net program, inner loops accumulate between 8 and 35 values, and such accumulations are performed more than 5K times.

Both of these programs accumulate values from registers or memory that are already computed. It makes these programs good candidates for the proposed transformations.

Vivado HLS is unable to predict a latency for these designs due to their non-constant loop trip counts. As a consequence, instead of presenting results for the complete benchmark, we restrict ourselves to the LU innermost loops. Table 3.8 shows the results obtained for the smallest (8 terms) and the largest (45 terms) sums-of-products in lines LU-8 and LU-45 respectively. The latency is vastly improved even for the smallest one. The accuracy results of the original code here varies from 8 to 23 bits between different instances of the loops. To have a fair comparison, we generated a conservative design that performs 23 bits of accuracy on all loops, using a sub-optimal amount of resources.

Black Scholes This program contains an accumulation that sums 200 terms. The result of this computation is divided by a constant (that could be optimized by using transformations from Section 3.1). This process is performed 5000 times.

Here the optimal accumulator parameters are the following:

```
MaxAcc=245000.0 epsilon=1e-4 MaxInput=278.0
```

This gives us an accumulator that uses 19 bits for the integer part and 10 bits for the fractional part. The result of the synthesis are provided in Table 3.8.

For comparable area, accuracy is vastly improved but latency could not be obtained statically from Vivado HLS. Indeed, the Black Scholes algorithm uses the mathematical function *power*. Such a function is not natively supported by Vivado HLS, and was therefore implemented by hand using a data dependent trip count loop. Because of this, the tool cannot statically derive the execution latency of the benchmark making the overall latency data dependent. One could use cosimulation to obtain the latency of a specific set of inputs.

Fourier Coefficients The Fourier coefficients program, which computes the coefficients of a Fourier series, contains an accumulation which is performed in single precision. This program comes in three different configurations: small, medium and big. Each of them computes the same algorithm but with a different amount of iterations. The big version is supposed to compute the most accurate answer. We obtain similar results for the three versions of this program, as a consequence we only present the big version here. In this version, there are multiple instances of 2K terms accumulations. The accumulator is reset at every call.

The parameters determined for this benchmark were the following:

```
MaxAcc=6000.0 epsilon=1e-7 MaxInput=10.0
```

This results in an accumulator using 14 bits for the integer part and 24 bits for the fractional part. The synthesis results obtained for the original and transformed codes are given in Table 3.8.

Here again, area cost is comparable, while accuracy is improved by 5 bits (which represents one order of magnitude). As for Black Scholes, Vivado HLS cannot compute the overall latency due to the *power* function. However, since our operators have a shorter latency by design, we expect the circuits to also have a shorter latency.

3.3 Discussion

This Chapter demonstrates how today's HLS tools fail at exploiting full FPGAs potential when dealing with floating-point numbers. The historic nature of x86 backends compiler is embedded in these hardware compilers. CPU specific optimizations are then followed in a custom hardware context. This choice is questionable knowing FPGAs best assets are custom datapath that differs from CPUs'.

Well known low-level arithmetic optimizations can still be applied to a high-level input source, as showcased in this study. The benefit in terms of resource usage and latency makes these optimizations a *must do* to close the gap between HLS and RTL design. Furthermore, the behavioural description of a program, as seen per the compilers, allows for further optimizations than what can be applied in RTL design. Indeed, the HLS compiler can extract information from the context in which the operator is used.

We provided a tool that automatically transforms a Vivado HLS compliant C/C++ code to a transformed equivalent (available at `gitlab.inria.fr/gecos/gecos-arith`). This transformed code got its floating-point accumulations; divisions and multiplications by small constants enhanced using application-specific arithmetic. The goal of this tool is not to be used before using a HLS tool but to show that HLS tools should implement these transformations.

A very little number of operators were studied in this work. These were examples to showcase that HLS tools are capable of highly effective arithmetic optimizations. The greater goal of this work is to gather two communities: arithmeticians and compiler designers. Therefore, it would allow to integrate and enhance a lot more operators within HLS tools.

Chapter 4

Architecture exploration of exact floating-point accumulators

Chapter 3 demonstrated the benefits of using a tailored fixed-point accumulator for floating-point accumulations. However, the parameters of the fixed-point format depends on the nature of the manipulated data. The fallback strategy chosen in this work is then to use a large fixed-point format. Specifically, the present work builds upon a proposal by Capello and Miranker [74] refined in the 90s by Kulisch [75, 76, 77] and others [78, 79]: a fixed-point accumulator that holds enough bits to ensure that sums and sums-of-products of floating-point data can be performed without any rounding error. Designing such an *exact accumulator* is quite area-demanding [77], but the recent availability of dark silicon [80] has sparked renewed interest in this proposal when targeting ASICs [81, 82, 83]. Its implementation using the large registers of existing vector units has also been studied [84]. A recent article [85] suggests that an exact accumulator is a very competitive way of implementing a 16-bit floating-point (half-precision) unit.

In this last work, the accumulator itself remains quite small (80 bits), and the author considers two alternatives: storing it in a sign+magnitude comparable to standard floating-point, or storing it in 2's complement. Its conclusion is that (in terms of cost) *"The 2's complement accumulator is a little ahead of the sign+magnitude. Nonetheless the sign-magnitude conversion (to standard floating-point) would require less hardware"*.

For larger accumulators, 2's complement has been used since the beginning [79], essentially for the reasons that have made 2's complement the format of choice for most fixed-point applications: there is only one zero, and the same hardware can perform both addition and subtraction.

In an exact fixed-point accumulator, a positive summand may trigger the propagation of a carry bit all the way up the accumulator, possibly even changing its sign. Similarly, a negative summand may trigger the propagation of a negative carry, also called a borrow bit. A central feature of 2's complement is that carries and borrows are two faces of the same coin. However, at least two independent recent work [82, 83] describe architectures that explicit distinct bits for carries and borrows. Indeed, the reference textbook [77] is itself not perfectly clear in this respect. The present Chapter therefore attempts to survey issues of sign management in more details than in the existing literature. It systematically explores exact accumulator variants, including some that are novel to the best of our knowledge, and more efficient than the state of the art. This exploration is performed while targeting FPGAs, where the accumulator is tailored to a custom size depending on

the application needs. This work provides implementations of full sized accumulators for when the accumulator can't be tailored.

Section 4.1 presents in more details the concept of a large accumulator for exact sums and sums-of-products. Sections 4.2 and 4.3 demonstrate that for each implementation variant, the proper use of 2's complement for sign management leads to simpler architectures and, more importantly, reduces the critical path of the accumulator loop. Section 4.4 discusses the final conversion of the large accumulator back to a floating-point format. All these variants have been implemented as a templated C++ library, compliant with Vivado HLS, that is used in Section 4.5 to compare them.

4.1 Parameters of an accumulator for exact sums and sums-of-products

The main idea here is to remove all sources of rounding errors in a dot product by using exact multipliers and exact accumulators. The block diagram of Figure 4.1 illustrates the architecture envisioned.

As shown in Chapter 3, the product of two normal floating-point numbers can be written:

$$xy \;=\; (-1)^{s_x+s_y} \;\cdot\; 1.f_x \times 1.f_y \;\cdot\; 2^{e_x+e_y}$$

The significand product $1.f_x \times 1.f_y$ is a fixed-point number of $wd'_f = 2wd_f + 2$ bits with two integer bits before the point. The exponent sum $e_x + e_y$ is an integer on $wd'_e = wd_e + 1$ bits. This defines a non-standard, non-normalized (wd'_e, wd'_f) floating-point format for the exact product.

An exact fixed-point accumulator should be large enough such that the bits of its fixed-point format cover all the possible bit weights of summands, whatever the exponents. The minimum size wd_a of such an accumulator can be deduced from the exponent width wd_e and significand width wd_f of the input format. Hence $wd_a = 2^{wd'_e} + wd'_f - 1$ (wd_a for

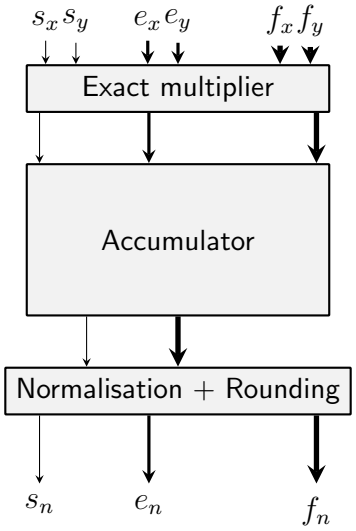


Figure 4.1: Block diagram of a large accumulator for exact sums and sums-of-products.

Table 4.1: Accumulator sizes for products of IEEE-754-2008 data.

format			significand weights		product weights		wd_a
name	wd_e	wd_f	min	max	min	max	(bits)
half	5	10	2^{-24}	2^{15}	2^{-51}	2^{31}	85
float	8	23	2^{-149}	2^{127}	2^{-301}	2^{255}	559
double	11	52	2^{-1074}	2^{1023}	2^{-2151}	2^{2047}	4201

standard formats are given in Table 4.1). The one bit deduced is the implicit bit of the representation, already encoded in the exponent range. Kulisch suggested to add even more bits to absorb possible temporary overflows. Therefore, we do not remove the bit that encodes ∞.

It is important in practice that a Kulisch accumulator may also be used to compute simple sums. Here, we must make the distinction between two usage contexts.

In the (co)processor hardware context envisioned in most previous works, simple sums use the same exact accumulator as sums-of-products. In the case of simple sums, standard floating-point inputs must be converted from the standard format (wd_e, wd_f) into the extended non-normalized format (wd'_e, wd'_f). As the latter format is a superset of the former, this simply requires a few additional multiplexers from the exact multiplier, so that the input fraction f_x has its leading bit explicit (with the same subnormal management as in the exact multiplier) before being right-padded with zeroes, and transferred directly on the output without going through the multiplier (hence replacing the multiplier from Figure 4.1).

In an HLS context, in the case of a simple sum, the compiler will not generate the multiplier hardware, and the exact accumulator may be smaller, as shown in next Section. In this case, the input to the accumulator is a non-standard format with explicit leading significand bit $(wd'_e, wd'_f) = (wd_e, 1 + wd_f)$. For this HLS context, in all the following, the term "summand" should be understood as referring either to a standard summand, or an exact product. In both cases, the exponent is noted e_p and the significand m_p. Similarly, we will use the term *exact accumulator* for accumulators of both exact sums and sums-of-products.

In the case of a simple sum, the value of wd_a will be roughly halved compared to the value of Table 4.1 for a sum-of-products. We therefore keep wd_a as a parameter, and the values given in Table 4.1, although used in the rest of this Chapter, should be considered worst-case (or fall-back) values.

In his book [77], Kulish suggested three ways of implementing the long accumulator, which we review now.

4.2 Kulisch-1: Long adder and long shift

4.2.1 Kulisch-1 with the accumulator in sign+magnitude

A first idea is to use for the accumulator a sign-magnitude representation similar to that of floating-point: A register of wd_a bits contains the absolute value a of the accumulator, and the sign is stored in a separate bit s_a. A summand is first shifted to the correct

place, according to its exponent. This requires a large shifter. The shifted summand p and the accumulator are then either added, or subtracted, depending on their respective signs. This costs an adder/subtracter of wd_a bits (implemented as a row of XORs and the leftmost adder on Figure 4.2).

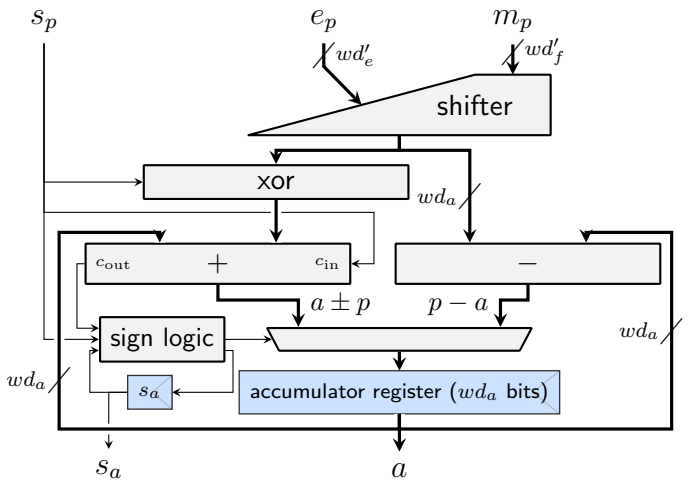


Figure 4.2: Kulisch-1 with sign-magnitude accumulator: low-latency architecture.

In the case of an effective subtraction, the result may become negative. This case must be detected, and the result negated before being stored in the accumulator register. This negation costs a second adder, with a second carry propagation, then a multiplexer. A classical improvement [23], implemented in Figure 4.2, is to compute in parallel $a \pm p$ and $p - a$. This solution has roughly the same hardware cost (two adders, one multiplier) but improves the critical path, since the two carry propagations are in parallel.

However, there remains a wd_a-bit addition in the critical path. It will be either slow due to carry propagation, or expensive if fast adder architectures are used. In the FPGA experiments detailed in Section 4.5, the operating frequency of Kulisch-1 architectures is acceptable for 16-bit floating-point inputs, but not for the larger formats which are the most widely used. Still, it may be the best choice for very small formats [85], or if the context enables small exponent ranges such as in Chapter 3 or [65].

4.2.2 Kulisch-1 with the accumulator in 2's complement

The architecture of Figure 4.3 was introduced in [65]. It makes the sign management explicit if the exact accumulator is kept in 2's complement. Here, a negative summand is converted to 2's complement after the shift using a wide XOR and a carry-in to the adder that implement the well-known equation $-x = \overline{x} + 1$.

Figure 4.3 is obviously simpler than Figure 4.2, and (more importantly) its critical path is a little shorter.

Besides, it enables another improvement: it is possible to XOR the significand before the shift, leading to a smaller XOR (as illustrated by Figure 4.4). In this case, the shifter must pad left and right with s_p instead of padding with 0s. This is mostly for free: the overall fanout on s_p remains close to wd_a.

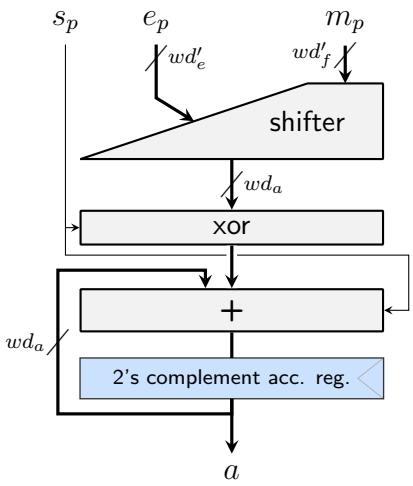


Figure 4.3: Kulisch-1 with 2's complement accumulator.

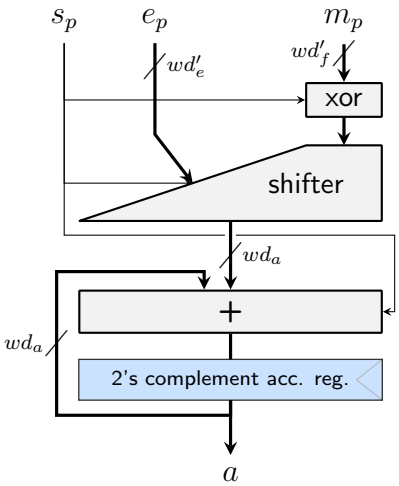


Figure 4.4: Alternative Kulisch-1 with 2's complement accumulator.

4.2.3 Kulisch-1 high-radix carry-save architecture

To increase frequency, the large input shifter, as well as the Normalisation+Rounding step from Figure 4.1 (which contains a lzc and a shifter) can be pipelined arbitrarily (they are combinatorial). However, the data-dependent loop around the accumulator still has the delay of a wd_a-bit addition. One way to reduce it is to register the carry propagation every b bits [79, 65]. The accumulated sum is still stored exactly, but in a radix-2^b carry-save representation (Figure 4.5 for Kulisch-1 with the accumulator in 2's complement). This architecture enables single-cycle accumulation at an arbitrary frequency (the smaller b, the higher the frequency), at the cost of an additional 1-bit register every b bits. Typical values for b should range from 16 to 64 bits, therefore this hardware overhead will be limited to a few percent.

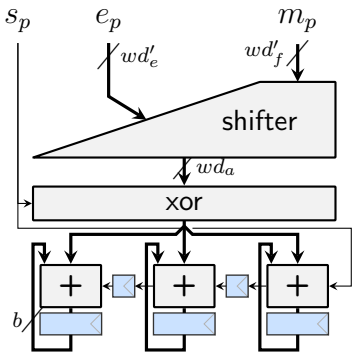


Figure 4.5: 2's complement Kulisch-1 with radix-2^b carry-save.

There will also be some overhead (in time or area) in converting the radix-2^b carry save accumulator back to floating-point. This will be discussed in Section 4.4.

This last Kulisch-1 architecture is very similar to the segmented architectures which we review now.

4.3 Segmenting the accumulator into words

All the other architectures segment the long accumulator in $N = \lceil wd_a/b \rceil$ words of b bits. The main motivation is that contrary to Kulisch-1, a summand then only needs to be added to a small number S of words, not to the full accumulator.

The input shift operation is therefore decomposed into word selection, and intra-word shift. If $b = 2^k$, the intra-word shift distance is simply obtained as the k lower bits of the exponent, while the word address is obtained as the $wd'_e - k$ leading bits. This is described in Figure 4.6.

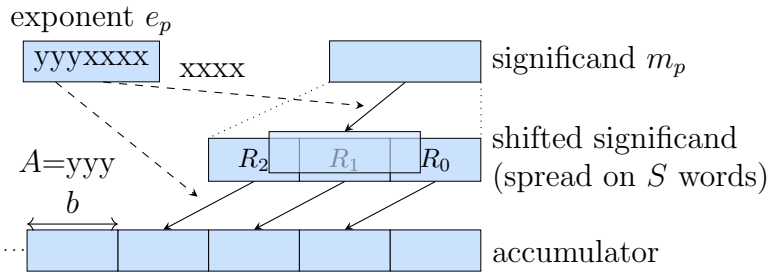


Figure 4.6: Shifting a significand into S sub-words (here $S = 3$).

A significand will typically be spread across multiple words: at least two, but possibly more. Precisely, after a shift of maximum size $b - 1$, the shifted significand is of size $wd'_f + b - 1$, and is spread over $S = \left\lceil \frac{wd'_f+b-1}{b} \right\rceil$ words [79].

Such a segmentation has two advantages: 1/ the two steps of the shift can be executed in a pipelined fashion, and 2/ the required addition is smaller: only S words of shifted significand need to be added to the corresponding words of the accumulator. However, there are two issues to address. The first is *carry propagation* from one accumulator word to the next. This potentially requires to update all the words above the S target words. If done naively, this may either require a long delay, or a variable number of cycles, during which the accumulator is not available for further inputs. The second is *sign management*: the shifted significand needs to be either added or subtracted to the accumulator, depending on its sign. In case of subtraction, the first issue becomes an issue of *borrow propagation*.

The reader should now convince himself that a sign+magnitude representation of the accumulator requires a specific mechanism. One problem would be that the operation to be performed (effective addition or effective subtraction) depends on the sign of the accumulator (see Figure 4.2). But the sign itself depends on the carry propagation through the whole accumulator: in case of effective subtraction, the sign of the result cannot be predicted. And when such a subtraction changes the sign of the accumulator, all its words are impacted, not only the S words facing the shifted significand.

Next Sections discuss the proposed solutions to handle signed accumulations and motivate the use of 2's complement.

4.3.1 Kulisch-2: Segmented accumulator in RAM

In this variant, the words of the exact accumulator are stored in an on-chip RAM. To address the carry/borrow propagation issue, Müller *et al.* introduced [79] an array that

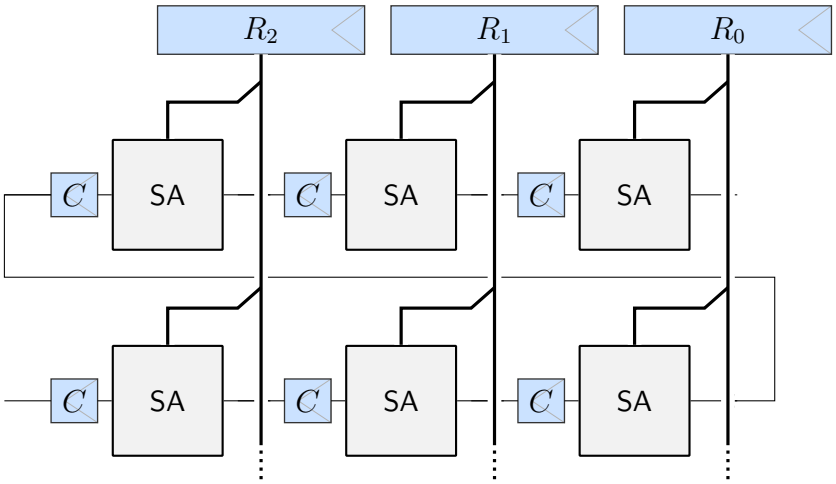


Figure 4.7: Kulisch-3 accumulator, here for $S = 3$.

stores the state of each accumulator word. The state of a word can be "all ones", "all zeroes", or "a mix of ones and zeroes". When a significand is added to word number M, the architecture uses the state array to also load from memory the first word $M' > M$ not containing only ones. If a carry is produced, M' will receive it, and the state of all the words between M and M' switch from "all ones" to "all zeroes".

Subtractions are handled in a similar way: First, the shifted significand on S words (Figure 4.6) is converted to 2's complement. Note that this conversion is incomplete: if the summand was negative, it should have its sign replicated all the way up to the most significant bit of the accumulator. However, the numerical value of this sign extension is identical to single negative bit in the carry-out position of the shifted significand: this is what is called a borrow bit. A borrow bit is therefore always produced for a negative summand, and then sometimes absorbed by the (positive) effective carry out of the addition (of the negated summand). If it is not absorbed, the borrow bit propagates up the accumulator similarly to a carry bit: words containing all 0s are transformed into all 1s, until a word absorbs the borrow. The main idea here is that the same state array works both for propagating carries and borrows [79].

The main problem with the segmented accumulator in RAM is its latency. It is impossible to pipeline it: the RAM reads cannot be overlapped with the RAM writes if they happen to touch the same locations, which cannot be predicted either since it depends on the carry resolution. This architecture is therefore not suitable for single-cycle accumulation. A recent article [83] presents a pipelined version of this architecture and claims single-cycle accumulation, but it is unclear how pipeline stalls are avoided.

4.3.2 Kulisch-3: Sub-adders with delayed carry propagation

This architecture, depicted on Figure 4.7, contains one sub-adder (SA) for each accumulator word. These N SAs can work in parallel. Besides, a register receives the carry and borrow values produced by each SA, and transmits it to the next SA, to be consumed at the next cycle. Thus, the carry/borrow propagation is delayed: carries and borrows are partly propagated at each cycle, but it takes N cycles at the end of an accumulation (inputting zeroes) to complete the propagation. This will be discussed in Section 4.4.

In practice, each significand part, tagged with its destination address i, is sent on an accumulator bus (see Figure 4.7). This bus spans all the SAs and is composed of S lanes. Each SA listens to this bus, and the matching SAs perform their part of the summation in parallel.

This architecture enables accumulation with 1-cycle latency. The loop critical path that limits the frequency is the sub-word accumulation. The two next sections detail alternatives for this.

Sub-adder with summand in sign+magnitude

In this version, used in [82] and [83], and depicted in Figure 4.8, the shifted significand is sent on the bus in sign-magnitude format. To accumulate it, the SA must either add it or subtract it, and also add the carry and subtract the borrow.

In Kulisch's book [77], the SA adder is depicted as an adder/subtracter that also inputs a carry. This is not possible with a single adder, as an adder/subtracter uses its carry-in bit to select between addition and subtraction. A possible architecture with two adders is shown Figure 4.8.

Here the borrow propagation line is not redundant with the carry line: it is used to transmit a *negative* bit. Therefore, on Figure 4.7, each C register is a 2-bit register. Normally, a subtracter is an adder with one input complemented and its carry-in set to one. This does not allow for another 1-bit borrow summand.

Sub-adder with summand in 2's complement

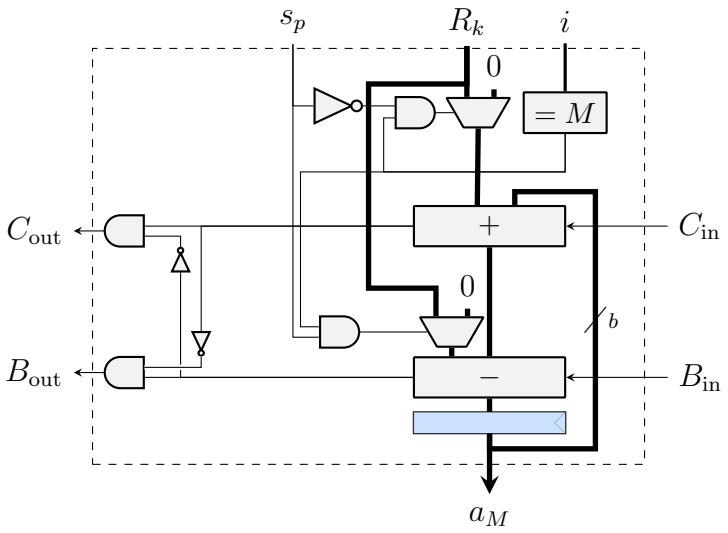


Figure 4.8: Kulisch-3 SA with summands in sign-magnitude.

Sending the summands in 2's complement simplifies the Kulisch-3 SA (Figure 4.9): it now requires only one binary adder, and on Figure 4.7, each C register is a 1-bit register.

Compared to the sign-magnitude Kulisch-3 architecture, sign management has been transferred from the SAs (where it was replicated N times) to a single adder of size $b \cdot S$ and to XOR the S words of the shifted significand. Compared to the segmented Kulisch-1 of Figure 4.5, this architecture requires a smaller XOR (it was wd_a bits on Figure 4.3).

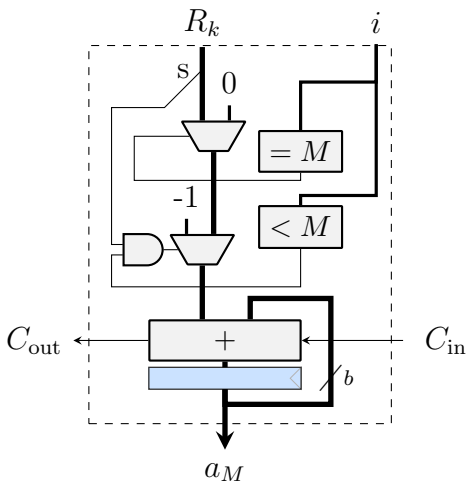


Figure 4.9: Kulisch-3 SA with summands in 2's complement.

However, in 2's complement, a negative number must have its sign extended all the way to the most significant bit of the accumulator. Therefore, when a negative significand is sent on the bus, the S SA's with the matching addresses add it to their accumulator sub-word, and all the SA's with higher addresses add -1 (a string of ones) to their sub-word. This performs the sign extension of the 2's complement significand across the entire floating-point range. It costs one extra comparator, but on a few bits only (precisely $\lceil \log_2 N \rceil$, for instance 6 bits for double precision (N=64) when $b = 64$).

4.4 Recovering the accumulator values

We now discuss the conversion of the accumulator value back to floating-point. In his book, Kulisch states that a correct rounding of the accumulator is of importance. Therefore, the accumulators proposed here perform the *round-to-nearest, ties to even* rounding policy.

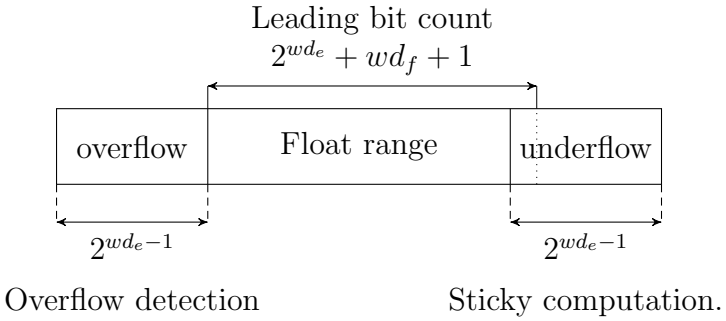


Figure 4.10: Structure of an exact accumulator.

Figure 4.10 illustrates how the complete accumulator is split in three parts. The size of the exact accumulator is $wd_a = 2^{wd'_e} + wd'_f - 1$. Hence $wd_a = 2^{wd_e+1} + 2wd_f + 1$. However, as the accumulator holds the exact sum-of-product, the complete range can't be represented in the output format (wd_e, wd_f). Therefore, the bits of the accumulator that have a weight greater than 2^{wd_e} will only be used to detect overflows. Similarly, the bits with a weight lower than 2^{-wd_f} will be used to compute a sticky, further used to perform

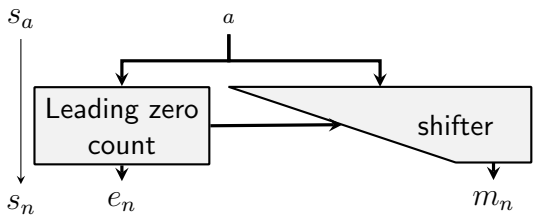


Figure 4.11: Normalisation of the large accumulator in sign magnitude.

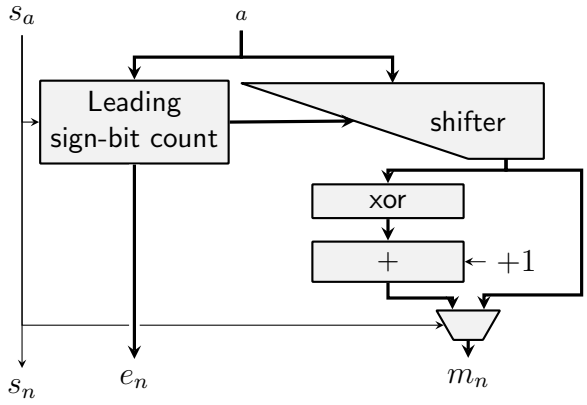


Figure 4.12: Normalisation of the large accumulator in 2's complement.

the correct rounding of the result. In the end, the reduced accumulator to normalize fits in $2^{wd_e} + wd_f + 1$ bits.

In the case of a sign+magnitude accumulator (Figure 4.11), a leading zero count determines the most significant bit, hence the shift to apply to the exact accumulator value.

If the accumulator is in 2's complement, then it needs to be negated when it is negative. This is best performed on the significand result after normalisation (Figure 4.12). To do so, the leading zero count has to be replaced with a leading sign-bit counter (same cost as the leading zero count), and then the result significand may be negated (one $(wd_f + 1)$-bit adder). The saving of one wd_a-bit addition in the accumulation itself is worth this overhead in the normalisation.

Kulisch claims that the state array simplifies the normalisation. However, our experiments with the state array in Kulisch-2 question this claim. Indeed, the overflow and underflow limits from Figure 4.10 are not multiples of the word size. Hence it still requires a significant amount logic to handle such special cases. Plus, with the state array management comes with a hardware overhead and increases the length of the accumulation critical path. However, the state array has the benefit of resolving the carry propagation in one cycle. In this work, the state array is completely removed in order to reduce the loop's critical path. In this case, all segmented accumulators have to propagate their carries completely, in N cycles inputting zeroes before normalisation can start. This choice is motivated by the fact that such an accumulator should be used only when performing a large sum-of-product, hence mitigating the carry propagation cost.

As a final remark, an alternative is to replace the rounding policy by a truncation, thus

saving all the hardware rounding logic. Indeed, as the accumulator's value is not IEEE-754 compliant any more (while improving by nature the accuracy of the computation), the use of this IEEE-754 rounding policy can be questioned.

4.5 Evaluation of Kulisch accumulators

All these architectures have been implemented in an open-source templated C++ library that is compliant with Vivado HLS. They are parametrized in wd_e, wd_f, and the SA size b. For wd_a we use the values from Table 4.1, rounded up to the next larger multiple of b. We note SMKn for $n \in \{1, 3\}$ the versions using sign-magnitude, and 2CKn the variants using 2's complement. Synthesis results in the literature, when available, target VLSI technologies, which would make a direct comparison difficult. Therefore, we also implemented the original Kulisch versions. Latencies reported are of a Vivado HLS program that computes the dot product of vectors of size 10000, then converts the final value back to floating-point. Reported costs include that of this final conversion. All the numbers presented here were obtained with Vivado 2018.3 after place-and-route, on a Kintex 7 FPGA (xc7k160tfbg484-1).

The use of HLS to design low-level arithmetic components could be questioned. To check that this tool was reliable in this context, we first compared synthesis of 2CK1 operators developed here with the corresponding VHDL operators available in the FloPoCo tool [65]. After very few changes to the design style, we reached comparable frequency and logic usage. This is consistent with a body of recent work demonstrated that HLS tools are capable of generating efficient low-level design [69, 86].

Table 4.2: Dot-product architectures optimized for frequency.
*: Uses 5 BRAM

float	variant	LUTs	reg.	DSPs	cycles	@frequency
half	SMK1	701	760	1	10,017	@232MHz
	2CK1	761	565	1	10,020	@240MHz
	2CK3, b=32	625	762	1	10,027	@368MHz
single	*IEEE-754*	*481*	*965*	*3*	*130,010*	*@463MHz*
	SMK1	3,778	3,737	2	10,008	@76MHz
	2CK1	4,340	1,685	2	10,008	@84MHz
	2CK1, b=64	5,084	5,200	2	10,041	@340MHz
	SMK3, b=64	5,394	8,598	2	10,065	@222MHz
	2CK3, b=64	3,575	4,854	2	10,042	@348MHz
double	*IEEE-754*	*997*	*2,007*	*11*	*120,021*	*@370MHz*
	SMK1	39,664	12,692	9	10,004	@11MHz
	2CK1	37,898	4,771	9	10,003	@11MHz
	2CK1, b=64	52.027	69.911	9	10,184	@227MHz
	K2*, b=64	4,083	5,542	9	1,010,161	@253MHz
	SMK3, b=64	44.398	84.342	9	10,320	@205MHz
	2CK3, b=64	27,544	43,228	9	10,118	@252MHz
	2CK3, b=128	28,487	39,962	9	10,072	@218MHz

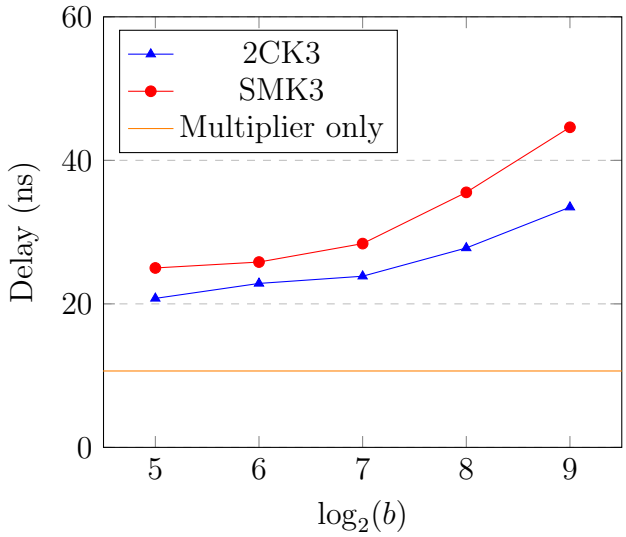


Figure 4.13: Combinational delays of FP64 dot-product architectures depending on SA size.

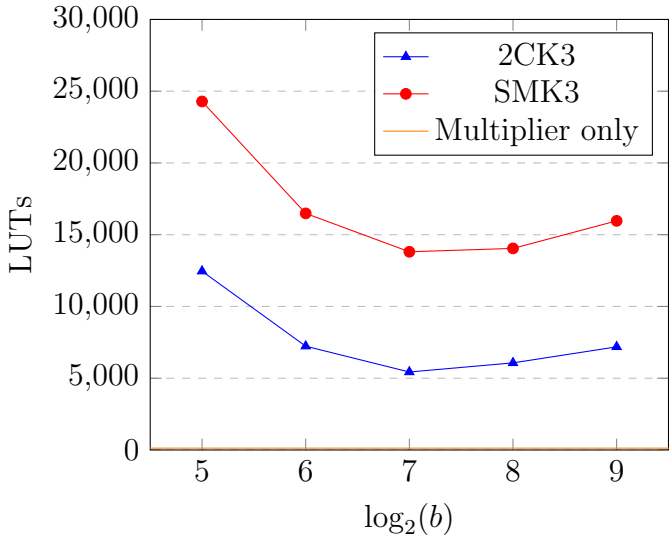


Figure 4.14: LUT usage of FP64 dot-product architectures depending on SA size.

4.5.1 Cost/frequency/latency trade-off

Table 4.2 presents results of accumulators taking a new input every cycle while targeting maximum frequency achievable. As expected, for SMK1 and 2CK1, half precision is fast enough but single and double precisions cannot achieve high frequencies due to the long carry propagation. The use of radix-2^b carry-save from Section 4.2.3 (noted 2CK1, b=64) allows for higher frequencies for both the single and double precision variants. In the case of single precision, the frequency in increased by a factor 4 at a 14% increase in LUT count. This speed-up is even higher in the case of double precision, with a factor of 14 for a 19% LUT cost increase.

As expected, SMK2 trades logic resources for BRAM, but it has a very long latency.

In all of these comparisons, the 2's complement versions perform the best in terms of frequency for little to no resources consumption increase. All other things being equal, Kulisch-3, unsurprisingly, also outperforms Kulisch-1.

Figure 4.13 shows the combinational delays of a single addition for both SMK3 and 2CK3 when varying the SA size b for double precision inputs (without final normalisation). In both cases the delay is composed of the multiplication followed by the SA delay. As the multiplication has the same delay for both architectures, the difference only comes from the SAs. It is also interesting to look at Figure 4.14 that reports the associated LUT count, demonstrating the increased complexity of the sign+magnitude architecture.

The targeted FPGA can perform a carry propagation of a 64-bit adder at about 340MHz. In the case of single precision, the maximum frequency achievable for 2CK1 (b=64) and 2CK3 (b=64) is limited by the size of the segment. However, the maximum frequency of the SMK3 (b=64) is only of 210MHz. This longer critical path is held by the accumulation loop (one 64-bit subtraction and one 64-bit addition). Indeed, this accumulation loop cannot be easily pipelined by inserting a register in between the addition and subtraction. Doing so would require having two accumulators (unsynchronized) living in parallel. Handling such a complex logic would require even more resources and make the cost even higher than the 2's complement alternatives. This SMK3 limitation is still true for double precision.

The maximum frequency of 2CK3 (b=64) for double precision does not meet the one obtained for single precision. Indeed, the critical path of the double precision architectures is in the exact multiplier. The product of two 53-bit significands result in a 106-bit addition. It is this final addition that limits the overall maximum frequency. This limitation comes from Vivado HLS integer multipliers implementation. Future work could involve the building custom Vivado HLS compliant integer multipliers with pipelined final additions to improve that metric. It is then interesting to note that the 2CK3 (b=128) allows to reduce the overall latency by having fewer words for only a slight reduction in frequency (now constrained by the 128-bit carry propagation, only slightly larger than the previous 106-bit addition).

Furthermore, all the variants that achieve single-cycle accumulation require $10,000 + n$ cycles to complete an exact dot product of size $10,000$. About half of this latency n is spent in the input shift, and the other half in the final normalisation (lzc+shift).

4.5.2 Comparisons with plain floating-point accumulation

Finally, it is interesting to compare the best accumulator for single precision with the naive alternative of performing the accumulation in floating-point. If the accumulation is performed in single precision, a Kulisch accumulator requires more than 10x the resources, but brings in a 13x latency improvement (13 is the latency of the floating-point adder). Performance-wise, it may seem a tie, but the Kulisch accumulator offers perfect accuracy. Note that techniques from the literature that attempt to reduce the latency [87] also require orders of magnitude more resource.

A simple and effective alternative to improve the accuracy of a dot-product of single precision data is to use double precision operators: conversion of single to double precision is error-less, and then the products become exact, and the accumulation much more accurate. Compared to this, Kulisch accumulators require 5x more logic resource, but much fewer DSPs. They provide a 14x latency improvement as well as guaranteed perfect

accuracy. This makes them a valid alternative to consider.

4.6 Discussion

This Chapter introduced more efficient variants of the exact accumulator, an expensive but low-latency and highly accurate implementation of sums and sums-of-products. A general conclusion is that 2's complement allows for higher frequencies for a lower resource usage. This contribution is validated on FPGAs but should also improve VLSI implementations designed for general purpose processors such as [82]. The templated C++ operators are available in the *Modern Arithmetic Tools (marto)* open-source library at `gitlab.inria.fr/lforget/marto`.

The idea of having an exact accumulator gained so much traction that a newly proposed format for encoding real numbers, called posits [88], mandates the use of an exact accumulator. The next Chapter studies in details the hardware implementation of operators for this number system, and compares is to floating-point numbers augmented with a Kulisch accumulator.

Chapter 5

Architecture exploration of operators for the posit number system

This Chapter results from a joint collaboration with Luc Forget.

The posit number system [88] (detailed in Section 5.1) is an emerging machine representation of real numbers that aims at replacing IEEE-754 floating-points. The first posit claim is that the floating-point representation is inefficient. When the exponent can be encoded on only a few bits, the rest of the bits should be used to extend the precision. The second claim, adopted from Kulisch [77], is that the sum of many products is a pervasive operation, justifying specific hardware to compute it exactly. To this purpose, the draft posit standard [89] mandates a *quire*, a variant of the exact Kulisch accumulator [77] for the posit number system. Such an operator has a hardware cost, as discussed in Chapter 4.

Most current evaluations of posits in applications are performed through software simulations [90, 91, 92, 93]. The C/C++ SoftPosit library [1] (among others [2]) implements the latest posit standard and allows for direct comparison with floating-point numbers in terms of accuracy.

However, the hardware cost of posits is not yet completely known. Hardware posit adders and multipliers have been written in HDL [94, 95] or using Intel OpenCL SDK compliant templated C++ [96]. Posits have been evaluated on applications such as machine learning [90, 91] or matrix multiply [92]. Among these works, only [90] is open-source and partially supports the quire, but only for 8-bit posits. [96] and [94] are parametric designs but are not open-source and do not support the quire. The work presented in this Chapter, although similar in spirit, refines the architectures in [96], attempting to use the same datapath optimization tricks that are used in the floating-point operators it compares to [23]. Conversely, [94] compares a posit implementation to a floating-point implementation that is 3x larger than the state-of-the-art.

This Chapter presents an improvement to the implementation of posit hardware with respect to all the previous works, and enables a comparison with state-of-the-art floating-point. It is parametric, open-source, and it is the first implementation to include a standard-compliant, parametric quire.

The proposed implementation is a templated C++ library compliant with Vivado HLS.

[1] `gitlab.com/cerlane/SoftPosit`

[2] `posithub.org/docs/PDS/PositEffortsSurvey.html` as of march 6, 2019

It currently offers standalone posit adders, subtracters and multipliers, with overloading of the C++ operators `+`, `-` and `*` for posit datatypes. Alternatively, the quire can add or subtract posits, or posit products, without rounding error. This open-source library[3] is built on a custom internal representation and extensible to other operators. The longer-term objective is really to make it possible for designers to easily switch an HLS design between floating-point and posit arithmetic, in order to compare their respective accuracy/cost/performance trade-offs.

The analysis is focused on 32- and 64-bit posits as 16-bit posits are, by nature, cheap and better than IEEE 16-bit floats. Still, the proposed library can be used for 16-bit posits too. For 16-bit formats, posits should be compared to other 16-bit formats alternative such as Bfloat used by Intel [97] and Arm [98] or DLFloat by IBM [99]. These formats are highly motivated by artificial intelligence algorithms, which does not require high accuracy.

Section 5.1 introduces in more details the posit number system, the algorithms for decoding and encoding them, and the datapath parameters entailed by these algorithms. Section 5.2 provides details on the architectural improvements implemented in the proposed library. Section 5.3 compares the performance and cost of the proposed posit operators, first to the state-of-the-art, then to floating-point operators. It also evaluates the quire in accumulation loops against IEEE floating-point and custom floating-point Kulisch accumulators.

5.1 Posit representation

The posit number system [88] is a floating-point encoding scheme with tapered precision. A posit format is defined by its size in bits (N) and its exponent field width (wd_{es}), which are the two parameters of the proposed templated implementation.

5.1.1 Decoding posits

The value of Figure 5.1 will be used as an illustrative example of how posits work.

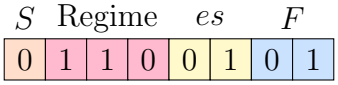


Figure 5.1: Posit decomposition example ($N = 8$, $wd_{es} = 2$).

The first bit S of the posit encodes its sign. Here the value is positive as $S = 0$. The exponent E of the number is split in two parts. The first part is computed out of the (variable-size) regime field, defined by a sequence of l identical bits ended by the opposite bit. The encoded range k is $-l$ if the bits of this sequence are equal to S, otherwise $l - 1$. In this example, the sequence consists in two ones: $l = 2$, therefore $k = 1$. The wd_{es} following bits are xored with S to obtain the lower exponents bits es: the exponent E is the concatenation of k and es. In our example, $E = 101$ as $es = 01$.

The remaining bits encode the fractional part F of the significand. An implicit leading bit I is obtained by negating S, here $I = 1$. Finally, the value of the posit can be defined as:

$$2^E \times (I.F - 2 \times S) \tag{5.1}$$

[3] `gitlab.inria.fr/lforget/marto`

The value represented by the example is

$$2^{101_2} \times (1.01_2 - 2 \times 0) = 2^5 \times 1.25 = 40.$$

Note that the regime can extend to the point where there is no room for F or es. In this case, the bits shifted out are assumed to be zeros.

Posit formats admit two special values, 0 and Not a Real (NaR). For encoding 0, all the posit fields are null, including the implicit bit. NaR is the equivalent of IEEE-754 NaN (Not a Number). Its encoding only has the sign bit set. There is no special encodings for infinity: posit arithmetic saturates instead. This is a disputable choice but disputing it is out of scope of this work.

5.1.2 Posits bounds and sizes

Due to the run-length encoding of the range k, posits with low magnitude exponents have more significand bits. The maximum precision wd_F is obtained for the minimum length of the regime (2), therefore

$$wd_F = N - (3 + wd_{es})$$

The maximal exponent is obtained when the regime running length is $N - 1$. In this case, all the es and F bits are pushed out by the regime. Hence the maximum exponent value is $E_{Max} = (N-2)2^{wd_{es}}$. The number of bits needed to store the exponent in 2's complement format is therefore

$$wd_E = 1 + \lceil \log_2 \left((N-2)2^{wd_{es}} \right) \rceil = 1 + wd_{es} + \lceil \log_2(N-2) \rceil$$

The wd_{es} parameter allows trading between the range of the format and its precision. The posit standard [89] defines four formats with an encoding size N of 8, 16, 32 and 64 bits respectively. The exponent field size wd_{es} of standard formats follows the relation $wd_{es} = \log_2(N) - 3$. These formats are used for evaluation in this Chapter, although the library is fully parametrized in N and wd_{es}.

A posit-compliant environment must also provide a *quire*. This latter allows for the exact accumulation of posit products. It is based on the floating-point Kulisch accumulator. The maximum exponent of a standard posit is $(N-2)2^{wd_{es}} = (N-2)2^{\log_2(N)-3} = N(N-2)2^{-3} = \frac{N^2-2N}{8}$. In that case, wd_F is reduced to 0 and only the implicit bit is set. For the standard formats, product magnitudes then range from $2^{-\frac{N^2-2N}{4}}$ to $2^{\frac{N^2-2N}{4}}$. Hence, $\frac{N^2}{2} - N + 1$ bits are required to store any such product in fixed-point representation. The standard motivates that the quire should easily be transferred to and from memory. To do so, it should have a size which is a multiple of 8. The addition of $N - 2$ carry bits and one sign bit fulfil that goal, hence the width of standard format quires is

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

The different sizes and bounds for standard posit formats are reported in Table 5.1.

The next Section introduces a custom internal representation for posits, based on previously shown sizes. This internal representation is used inside the arithmetic operators described further.

Table 5.1: Dimensions and bounds of standard posits ($wd_{es} = \log_2(N) - 3$).

N	wd_{es}	wd_E	wd_F	E_{Max}	wd_q	wd_{pif}
8	0	4	5	6	32	14
16	1	6	12	28	128	23
32	2	8	27	120	512	40
64	3	10	58	496	2048	73

5.2 Architecture

The variable-length fields of the posit encodings are not well suited to efficient computation on bit-parallel hardware. As all previous implementations, we first convert posits to a more hardware-friendly representation. A contribution of this work is to formally define this intermediate format.

5.2.1 Posit intermediate format (PIF)

The *posit intermediate format* (PIF) is a custom floating-point format used to represent a posit value with fixed size fields. Its main difference with standard floating-point is that the significand is stored in 2's complement just like the posit significand. This simplifies decoding, but also slightly simplifies the addition of two posits.

The significand is composed of three fields S, I and F, where S is a sign bit, I is the explicit leading bit of the posit significand, and F is its fraction field, on wd_F bits in order to accommodate the most accurate posits of the format (less accurate ones are right-padded with zeroes). For the example of Figure 5.1, $S = 0$, $I = 1$ and $F = 010$ ($wd_F = 3$ so the posit fraction is padded with one zero in this case).

The exponent is stored as the offset from the minimum posit exponent, on wd_E bits. This is similar to the biased exponents of IEEE floating-points, and motivated by the same reasons: it simplifies the critical path of the operators, at the cost of small additions in posit decoding/encoding, whose latency is hidden by the longer latency of significand processing.

Posit numbers with maximum magnitude exponents have their fraction bits completely pushed out ($F = 0$). For them, Equation 5.1 becomes

$$\begin{cases} 2^E \times 1, & \text{for positive numbers} \\ -2 \times 2^E = -2^{E+1}, & \text{for negative numbers} \end{cases}$$

Hence, the minimal exponent expressed in *posit intermediate format* is for $-2^{-E_{Max}}$. In this case, in order to verify $E + 1 = -E_{Max}$, the exponent value is $E = -E_{Max} - 1$. This leads to a bias value $Bias = (N - 2)2^{wd_{es}} + 1$.

Finally, three extra bits are added to the format. The *isNaR* bit is used to signal NaR. It avoids the necessity of checking for NaR in arithmetic operators. The Round and Sticky bits capture the necessary and sufficient information that must be kept after an operation on PIF values to correctly round the result back to posit.

To summarize, a *posit intermediate format* contains the following fields:

- A NaR flag $isNaR$ on 1 bit

- A sign S on 1 bit
- An exponent E on wd_E bits
- An implicit bit I on 1 bit
- A significand F on wd_F bits
- A round bit *round* on 1 bit
- A sticky bit *sticky* on 1 bit

The total width of the posit intermediate format is therefore $wd_{\text{pif}} = wd_F + wd_E + 5$ bits. Posit intermediate format sizes for standard posit formats are reported in Table 5.1 as wd_{pif}.

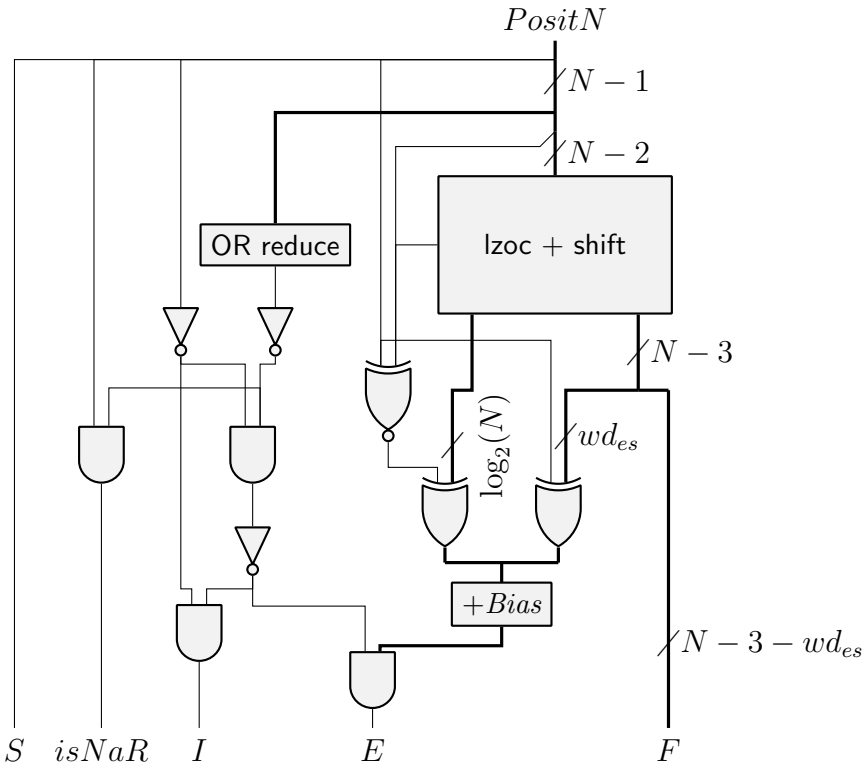


Figure 5.2: Architecture of a posit decoder.

5.2.2 Posit to PIF decoder

The proposed posit decoder is described in Figure 5.2.

The exponent of the posit is the combination of es and k, which is computed from the run-length l of the leading bit. Indeed, if the leading bit is 0, then $k = -l$ $(= \bar{l} + 1)$; if it is 1, then $k = l - 1$. By skipping a bit at the start of the sequence, the count returns $l' = l - 1$. Therefore $k = \overline{l' + 1} + 1$, hence $k = \bar{l'}$ if the leading bit is 0 or $k = l'$ if the leading bit is 1. The same method can be applied for negative numbers by computing

$k = l'$ when the leading bit is 1 and $k = \bar{l'}$ when the leading bit is 0. This method is different from the literature and allows for saving an addition when computing $-l$.

The most expensive part of this architecture are (a) the `OR reduce` over $N-1$ bits to detect NaR numbers and (b) the leading zero or one count (`lzoc + shift`) that consumes the regime bits while aligning the significand. The $+Bias$ aligns the exponents to simplify following operators. This decoding cannot be compared to an IEEE floating-point equivalent as no decoding is needed.

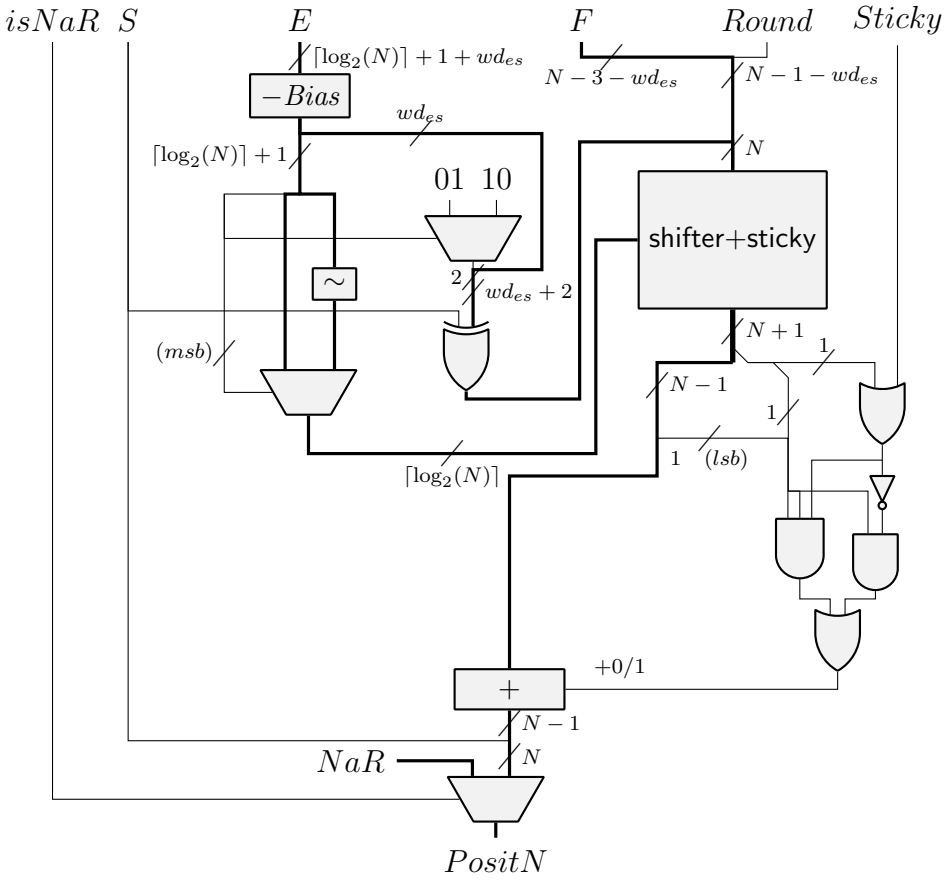


Figure 5.3: Architecture of a posit encoder.

5.2.3 PIF to posit encoder

Due to the variable-length encoding of posits, the position to which a PIF value must be rounded is known only when performing this conversion. The Round and Sticky bits carry synthetic information about the bits of the infinitely accurate result beyond the bits of F, but the encoder (depicted in Figure 5.3) still embeds quite some logic.

The fraction is first shifted to include the regime bits and es. Once shifted, the first $N-1$ bits represent the unrounded posit without sign. The remaining bits of the shifted fraction are used to extract the actual round bit and compute the final sticky bit. This information is used to compute the rounding to the nearest with tie to even.

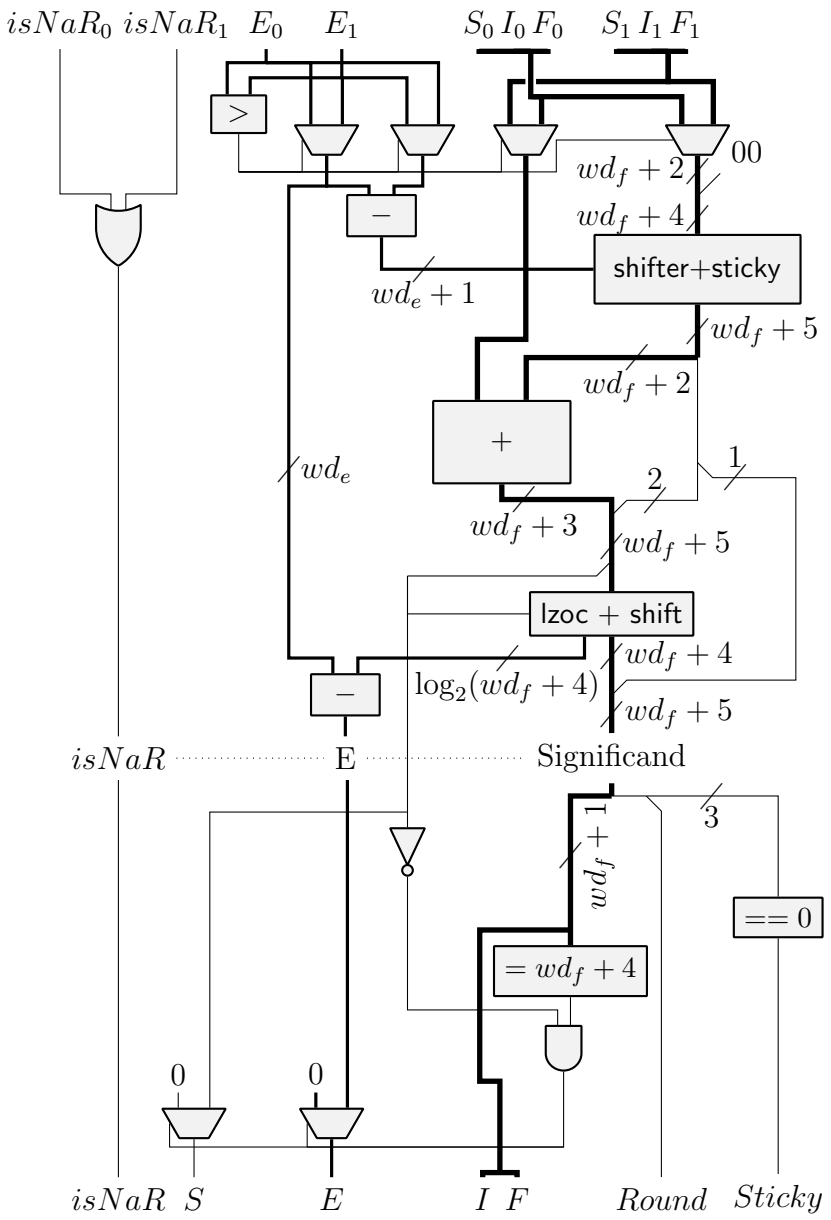


Figure 5.4: Architecture of a PIF adder.

5.2.4 PIF adder/subtracter and multiplier

The architectures of the PIF adder/subtracter (Figure 5.4) and multiplier (Figure 5.5) 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 [23], 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 5.4 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

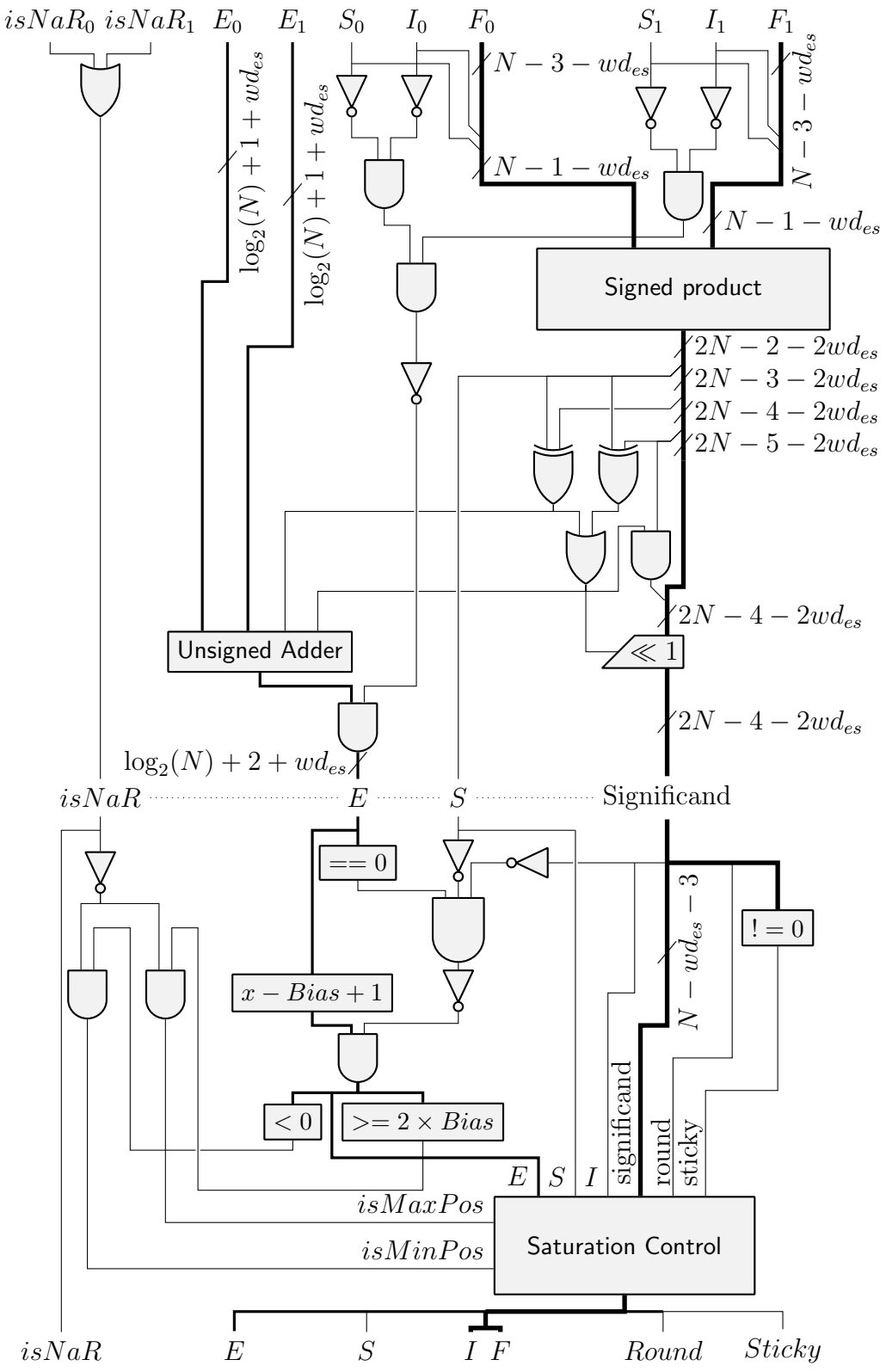


Figure 5.5: Architecture of a PIF multiplier.

be shifted back by the second lzoc+shift. All this allows us to keep most intermediate

signals on $wd_f + 2$ to $wd_f + 6$ bits, where previous work [96, 94] seem to use datapaths that are twice as large.

The bottom part of Figures 5.4 and 5.5 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.

5.2.5 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 Figure 5.5 (top).

To add a simple posit to the quire, it is first converted to PIF, then the PIF value is converted to the same exact multiplier format, which is straightforward.

The posit standard [89] specifies NaR as a special quire value. Testing this special value at each new quire operation is then expensive. Instead, this work proposes to add a flag bit that signals that the value held in the quire is NaR. This bit is set when NaR is added to the quire and stays set until the end of the computation. This extra bit can replace one of the quire carry bits. A slightly more expensive alternative would be to encode and decode NaR value when transferring quire to/from memory.

The proposed quire architecture is depicted in Figure 5.6.

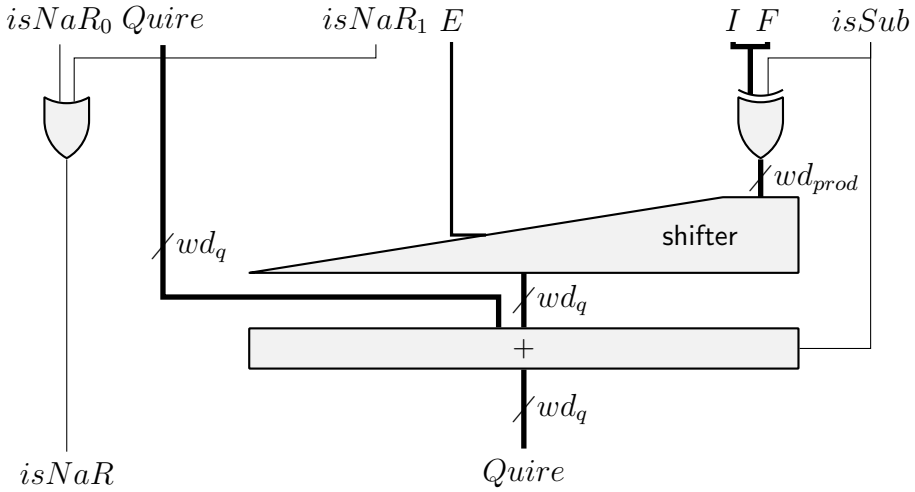


Figure 5.6: Architecture of a posit quire addition/subtraction.

Addition of products to the quire

The simplest implementation of the quire addition/subtraction is depicted in Figure 5.6 where the quire data structure is as depicted in Figure 5.7. An exact posit product fraction 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 subtraction is performed at very little cost using the same method as in the posit adder/subtracter.

There are several techniques to implement the quire, some of which allow for pipelined accumulations of products with one-cycle latency at arbitrary frequency such as discussed in Chapter 4. The most cost-effective way of achieving 1-cycle accumulation at high

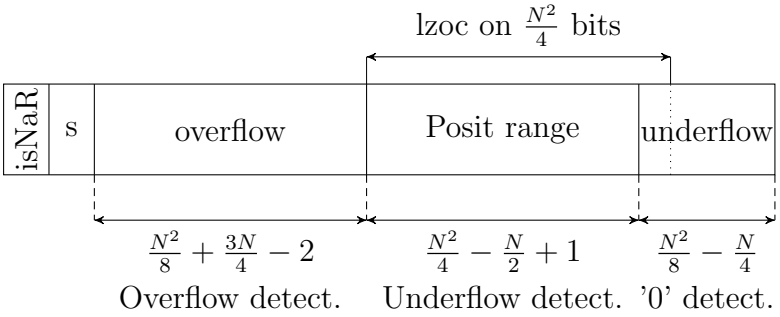


Figure 5.7: Quire conversion to *posit intermediate format*.

frequency is to keep the quire in a redundant format with delayed carries. The conversion of such a format to a non-redundant format (completing carry propagation) will incur additional overhead. The architecture chosen in this work is a custom 2's complement segmented kulisch accumulator. The impact of this choice on cost and performance is evaluated in Section 5.3.

Conversion from quire to posit

The conversion of the quire value to a posit is divided in two steps. The quire is first converted to a PIF value (architecture depicted in Figure 5.8) before the latter is encoded to a posit (Section 5.2.1).

The main performance challenge is the latency of converting a quire to a posit. Indeed, the first conversion requires a large lzoc of $\frac{N^2}{4}$ bits and two large XOR reduce to detect overflows and underflows/compute the sticky bit as shown in Figure 5.7. The sticky is required for correct rounding. The *overflow* and *underflow* regions holds the bits of exact products. Therefore, their range is higher than what a posit can encode. The *carry* region (from standard to absorb temporary overflows) of the quire is embed in the *overflow* region here.

5.3 Evaluation

All the designs presented here have been tested exhaustively for 8-bit and 16-bit standard posits against the reference SoftPosit implementation. They have also been tested extensively for other sizes.

The presented posit architectures are first shown to improve the state-of-the-art in 5.3.1. This ensures fair comparisons, with state-of-the-art floating-point operators in 5.3.2, and of exact acumulators in 5.3.3.

5.3.1 Comparison with state-of-the-art posit implementations

Results are reported in [94] for a Xilinx Zynq-7000, and in [95] for a Virtex 7. We chose for our comparison the simpler setting of [94] (Zynq-7000, no pipeline) and synthesized both

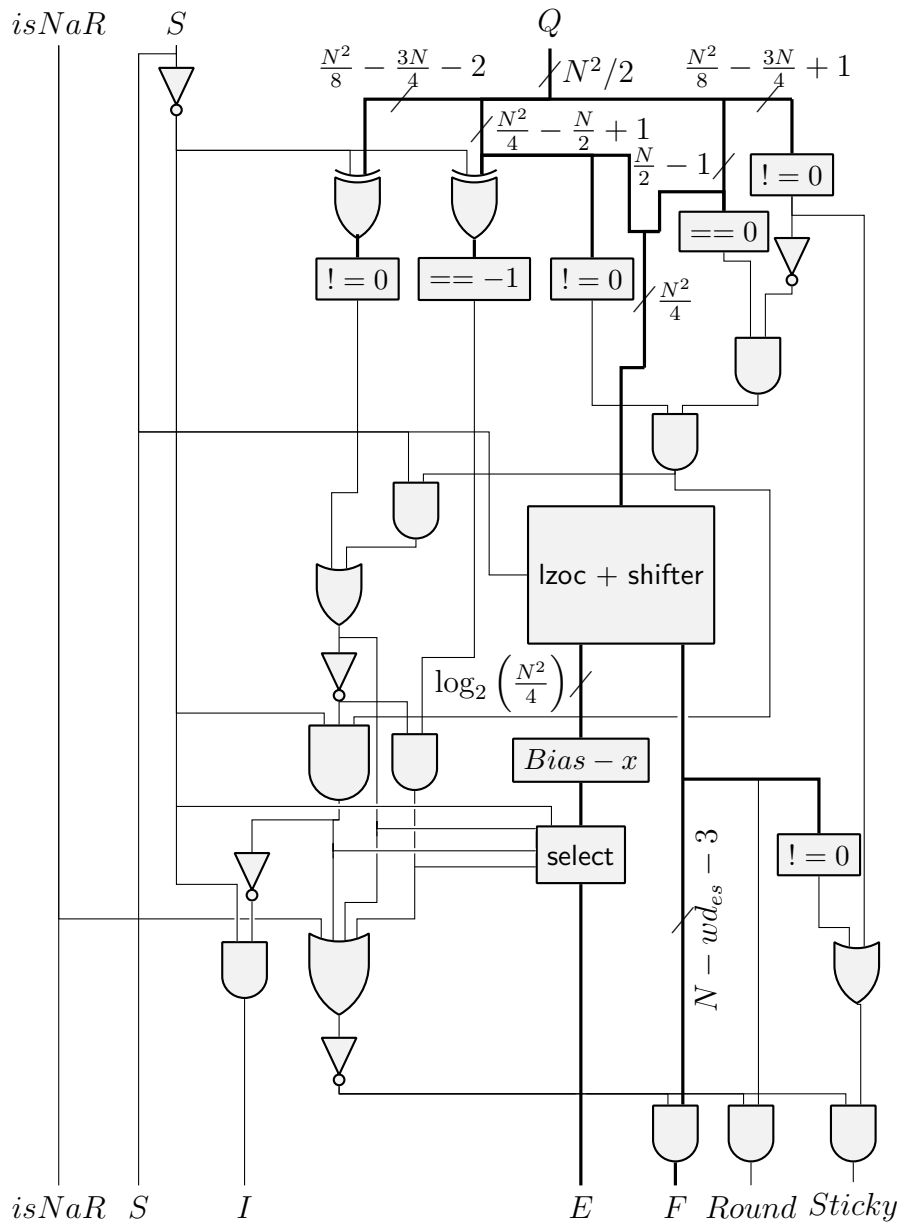


Figure 5.8: Architecture of the conversion from the quire to a *posit intermediate format*.

our library and that of [95] [4] for this setting using VivadoHLS 2018.3. Results are given in Table 5.2. The present work improves 32-bit operators in all metrics. In the 16-bit cases, the delays are always improved. There is only one case (the 16-bit multiplier) where [94] has better resource consumption. Still, in this case, the area.time (AT) and AT^2 of the proposed approach are better.

In [96], results are given for a Stratix V FPGA. Their adder operator is actually an adder/subtracter. The corresponding comparison is in Table 5.3. For this table, we used VivadoHLS 2018.3 to generate VHDL files which were then synthesized using Quartus 18.1. This worked without problem for our designs, at the cost of sub-optimal quality of

[4] source code accessed on June 26th, 2019 at
`github.com/manish-kj/Posit-HDL-Arithmetic`

Table 5.2: Comparison with [94] and [95] targeting Zynq (combinatorial components).

(a) Posit Adder

	N	LUT	DSP	Delay (ns)
[94]	16	320	0	23
	32	981	0	40
[95]	16	460	0	21
	32	1115	0	29
This work	16	320	0	21
	32	745	0	24

(b) Posit Multiplier

	N	LUTs	DSPs	Delay (ns)
[94]	16	218	1	24
	32	572	4	33
[95]	16	271	1	19
	32	648	4	27
This work	16	253	1	18
	32	469	4	27

Table 5.3: Comparison with [96] targeting Stratix V.

(a) Posit Add/Sub

	N	ALM	DSP	Cycles	FMax (MHz)
[96]	16	∼500	0	∼49	∼550
	32	∼1000	0	∼51	∼520
This work	16	327	0	19	584
	32	636	0	24	539

(b) Posit Multiplier

	N	ALM	DSP	Cycles	FMax (MHz)
[96]	16	∼330	1	∼35	∼600
	32	∼600	1	∼38	∼550
This work	16	199	1	16	600
	32	452	2	21	445

results. We report approximate data for [96] since it is read from graphical plots.

In general, the operators developed in this work require fewer resources and have shorter critical paths. This is mainly due to rigorous implementation of each component (shifters, lzoc, etc.) and improvements over existing architectures (addition saved in the encoder, contraction of the adder similar to state-of-the-art floating-point adders, etc.). There is a discrepancy in the 32-bit multiplication in Table 5.3: the 29x29 multiplier is implemented as two DSP blocks in 36x36 mode [100] in our case, while it is implemented in [96] as one DSP block in 27x27 mode, plus some logic. The slower frequency of our library in this case is not surprising, as we synthesize for an Intel FPGA the VHDL generated for a Xilinx FPGA. It will be solved in the near future by a portable HLS library instead of the current Vivado-specific one. Such a portable library is further detailed in Chapter 6.

Table 5.4: Synthesis results of posit and IEEE floating-point adders and multipliers.

(a) Adder

	N	LUT	Reg.	DSP	Cycles	Delay (ns)
Posit	16	383	358	0	18	2.702
	32	738	811	0	22	2.659
	64	1660	2579	0	33	2.609
IEEE	16	216	205	0	12	2.331
	32	425	375	0	14	2.690
	64	918	792	0	17	2.737
Float	32	341	467	0	9	2.529
	64	641	1098	0	11	2.562
Soft FP [69]	16	205	228	0	10	2.453
	32	416	527	0	13	2.239
	64	1237	1545	0	19	2.702

(b) Multiplier

	N	LUT	Reg.	DSP	Cycles	Delay (ns)
Posit	16	269	292	1	16	2.361
	32	544	710	4	21	2.421
	64	1501	2410	16	42	2.816
Float	32	80	193	3	7	2.201
	64	196	636	11	17	2.568
Soft FP [69]	16	38	127	1	8	1.825
	32	67	228	2	9	2.193
	64	259	651	9	10	3.299

5.3.2 Comparison with floating-point operators

All the remaining results given in this Chapter are obtained using Vivado HLS and Vivado 2018.3 targeting 3ns delay for a Kintex 7 FPGA (xc7k160tfbg484-1). Table 5.4 compares posits and floats of the same size on addition and multiplication.

On the addition side, we have a perfectly fair comparison between the results labelled "Posit" and the results labelled "IEEE": this latter line describes a fully compliant IEEE adder, with subnormal support, implemented with the same care as the posit operators and using the same parametric subcomponents. We observe that the posit adder is almost twice as large and twice as slow as the IEEE adder. Some of it is due to the variable-length field encoding and decoding (Figures 5.2 and 5.3). Some of it is due to the slightly extended internal precision of posits.

We also give results for two other mainstream floating-point implementations. The line labelled Float corresponds to the IP used by Vivado HLS when using the `float` and `double` C datatypes (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 of floating-point for Xilinx FPGAs. However, it is not IEEE-compliant: although the memory format is that of IEEE floats, subnormals are flushed to zero to save resources. The line labelled Soft FP reports a recent HLS-oriented templated floating-point library [69] which is not IEEE-compliant either.

The comparison on multiplication is less definitive, as it lacks a fully compliant IEEE multiplier implementation with subnormal support. Still, the posit multiplier is much more expensive and slower than the industry standard floating-point. Supporting subnormal adds an overhead roughly corresponding to one posit decoder (one lzoc and one shifter), and is not expected to overturn the game.

In absolute terms, there may be some overhead due to HLS tools, but recent works [86, 69], as well as the the comparison between the "Soft FP" and the "Float" hard IP, suggests that it is becoming negligible.

Table 5.5: Synthesis results for a sum of 1000 products
(U: Unsegmented, S32 and S64: word sizes of 32 and 64 bits).

		LUT	Reg.	DSP	Cycles	Delay (ns)
Quire 16	U	1409	1763	1	1028	3.215
	S32	1239	1431	1	1031	2.643
	S64	1185	1555	1	1030	2.756
Quire 32 (512 bits)	U	5068	6256	4	1040	8.850
	S32	4394	4779	4	1055	2.854
	S64	3783	4564	4	1047	2.961
Kulisch 32 (559 bits)	S32	4446	5290	2	1050	2.875
	S64	4365	5276	2	1041	2.854
Float 32		460	806	3	10011	2.676
Float 64		892	1999	11	12021	2.737

Table 5.6: Detailed synthesis results of hardware posit quire.

(a) Posit 16

		LUT	Reg.	DSP	Cycles	Delay (ns)
Decoding		59	64	0	4	1.986
Product		50	113	1	7	1.832
Quire addition	U	499	1078	0	5	2.681
	S32	459	357	0	4	2.628
	S64	432	543	0	5	2.437
Carry propagation	S32	108	137	0	5	2.548
	S64	71	134	0	3	2.545
Quire to posit		560	480	0	10	2.609

(b) Posit 32

		LUT	Reg.	DSP	Cycles	Delay (ns)
Decoding		137	142	0	5	2.158
Product		93	277	4	10	2.143
Quire addition	U	2384	4712	0	7	5.050
	S32	1424	984	0	5	2.679
	S64	1148	1066	0	4	2.488
Carry propagation	S32	519	535	0	17	2.549
	S64	480	531	0	9	2.945
Quire to posit		2534	2439	0	17	2.878

5.3.3 Quire evaluation

The synthesis results for the quire are given in Table 5.5 where 1000 sums-of-product are performed and the result is returned as a posit. They are compared to a floating-point Kulisch accumulator and to regular floating-point hardware. Kulisch and quire are presented in unsegmented (U) version along with two segmented versions (S32 and S64 for word sizes of 32 or 64 bits). The unsegmented versions are not able to achieve 3ns due to the long carry propagation. The Kulisch accumulator used in this paper is the 2's complement Kulisch 3 variant architecture from Chapter 4. Classically, using an exact accumulator consumes roughly 10x more resources but reduces the latency by 10x, while making the computation exact.

Here the cost and performance of a quire for 32-bit posits and a Kulisch accumulator for 32-bit floats are almost identical.

Detailed synthesis results of all the subcomponents are given in Table 5.6. The accumulation loop is the *Quire addition* component. It can be pipelined with an initiation interval of one cycle. During synthesis, the *Carry propagation* component will be merged

Table 5.7: Architectural parameters of N-bit posits.

	range $(\min, \max)$	custom FP (wd_E, wd_F)
8-bit posits, es=0	$[2^{-6}, 2^{6}]$	(4, 5)
16-bit posits, es=1	$[2^{-28}, 2^{28}]$	(6, 12)
32-bit posits, es=2	$[2^{-120}, 2^{120}]$	(8, 27)
64-bit posits, es=3	$[2^{-496}, 2^{496}]$	(10, 58)

with the *Quire addition*, reducing its cost. However, there is an irreducible latency for the final carry propagation once the accumulation is over.

The *Decoding* and *Product* components can be pushed out of the accumulation loop and pipelined to feed the *Quire addition* component. Conversely, carries must be propagated before the conversion *Quire to posit* can occur. Therefore, the total latency of the design is approximately the sum of the combined *Decoding*, *Product* and *Quire addition* pipeline depths; the *Quire addition* initiation interval, times the number of products to add; the *Carry propagation* pipeline depth; and the *Quire to posit* pipeline depth.

This latency is amortized for large sums. However, it has to be taken into account when considering the quire to add a few values, e.g. to emulate an FMA or a fused dot product.

5.4 Using posits as a storage format only

Table 5.7 shows that 8-bit and 16-bit posits can be cast errorlessly in 32-bit floating-points, while 32-bit posits can be cast errorlessly in 64-bit floating-points. One viable alternative to replacing IEEE floats by posits is to only use posits as a memory storage format, and to compute on the standard IEEE floats.

This solution has many advantages:

- It offers some of the benefits of the quire (internal extended precision) with more generality (e.g. you can divide in this extended precision), and at a lower cost;
- the better usage of memory bits offered by the posit format is exploited where relevant (to reduce main memory bandwidth);
- The latency overhead of posit decoding is paid only when transferring from/to memory;
- Where it is needed, we still have floating-point arithmetic and its constant error;
- Well established floating-point libraries can be mixed and matched with emerging posit developments;

It has one drawback that should not be overlooked: attempting to use this approach to implement standard posit operations may incur wrong results due to double rounding. This remains to be studied.

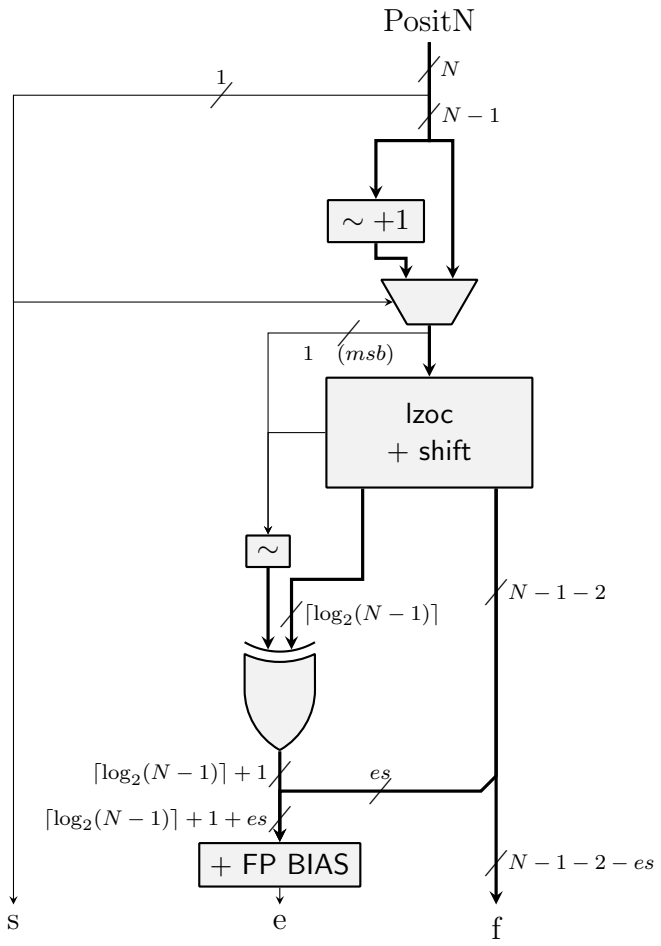


Figure 5.9: Architecture of a N-bit posit to floating-point decoder.

Note that a posit will never be converted to a subnormal, and that subnormals will all be converted to the smallest posit. This could be an argument for designing FPUs without hardware subnormal support (i.e. flushing to zero or trapping to software on subnormals).

The required conversion operators, described Figures 5.9 and 5.10 are similar to Figures 5.2 and 5.3. The `FP BIAS` corresponds to the floating-point bias.

This approach has been tested in an HLS context: we only implemented the converters of Figures 5.9 and 5.10, and relied on standard floating-point operators integrated in Vivado HLS.

Some synthesis results are given for a Kintex 7 FPGA (xc7k160tfbg484-1) using Vivado HLS and Vivado v2016.3 in Table 5.8. The conclusion is that posit decoders and encoders remains small compared to the operators being wrapped, all the more as the goal is to wrap complete floating-point pipelines, i.e. several operations, between posit decoders and encoders. The latency also remains smaller than that of the operators being wrapped.

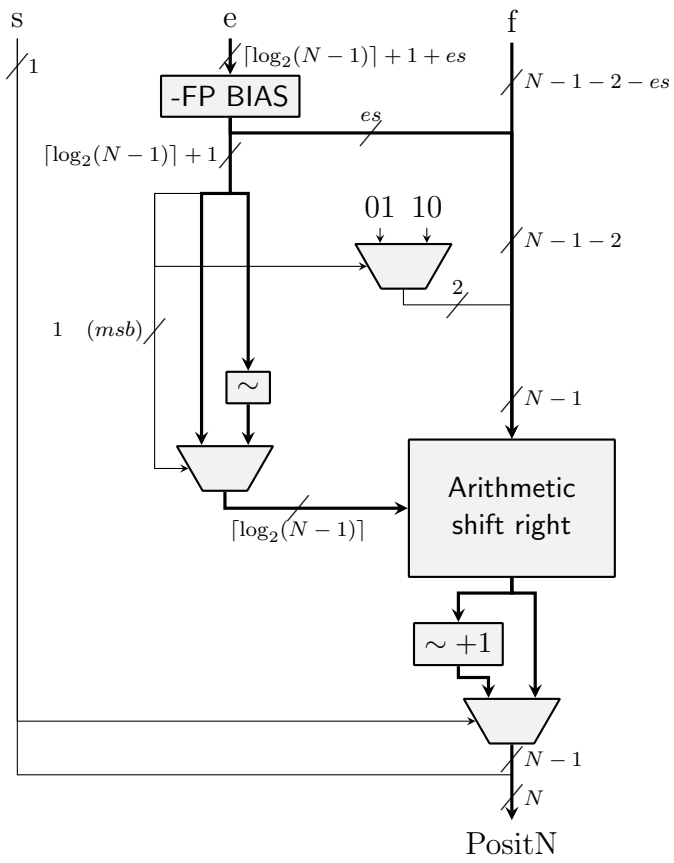


Figure 5.10: Architecture of a floating-point to N-bit posit encoder.

Table 5.8: Synthesis results of the posit encoders and decoders, compared with the operators they wrap.

posit size	float size	operation	LUTs	Reg.	DSPs	Cycles	Delay (ns)
8 bits	32 bits	posit to float	25	28	0	5	1.831
8 bits	32 bits	float to posit	20	30	0	3	1.968
16 bits	32 bits	posit to float	90	72	0	6	2.008
16 bits	32 bits	float to posit	63	56	0	4	1.347
	32 bits	ADD	367	618	0	12	1.956
	32 bits	MUL	83	193	3	8	1.984
	32 bits	DIV	798	1446	0	30	2.283
	32 bits	SQRT	458	810	0	28	2.331
32 bits	64 bits	posit to float	192	174	0	8	2.114
32 bits	64 bits	float to posit	139	106	0	4	2.008
	64 bits	ADD	654	1035	3	15	2.754
	64 bits	MUL	203	636	11	18	2.890
	64 bits	DIV	3283	6167	0	59	2.724
	64 bits	SQRT	1771	3353	0	59	2.577

5.5 Discussion

The purpose of this work is to enable the evaluation of the cost of converting a floating-point application to posits. To that end, a Vivado HLS templated C++ library[5] implements the posit number system, including the quire. This library has been implemented with the same care as state-of-the-art floating-point, with several improvements in the datapath that translate to greatly improved performance compared to previous posit implementations. Posit hardware is found to be more expensive than floating-point hardware. However, for applications where posits are more accurate than floats of the same size, the real use case should be to vary the parameters, so as to find which arithmetic provides the required application-level accuracy at the minimal cost. We hope that this work enables such studies.

This work also demonstrates that posits can be used as a storage format, relying on underlying floating-point hardware for little cost.

Future work includes completing the library with missing standalone operations (division, square root), and making it portable to a broader range of HLS tools.

In the context where one can vary the parameters of the posits to evaluate the cost/accuracy/performance ratio, it would be fair to also vary the parameters of the floats. An independent study shows [6] shows that the examples used by Gustafson exploit the few extra bits of accuracy sometimes offered by posits. A few extra significand bits to a floating-point format can make up for the extra accuracy of the equivalent posit at a lower resource cost and latency. Furthermore, a Kulisch accumulator can also be used to perform exact sum-of-products. In such a context, the accumulator could also be tailored to the application to save latency and resources such as described in Chapter 3.

This work revealed difficult comparisons of the proposed HLS components to other work targeting different vendors FPGAs, hence that use different HLS tools. The lack of compatibility between codes is mainly due (in our case) to each tool using its own custom size integer library. To address this problem, next Chapter introduces a header-only compatibility layer offering a consistent and comprehensive interface to signed and unsigned arbitrary-sized integers.

[5] `gitlab.inria.fr/lforget/marto`

[6] `marc-b-reynolds.github.io/math/2019/02/06/Posit1.html`

Chapter 6

A type-safe arbitrary precision arithmetic portability layer for HLS tools

This Chapter results from a joint collaboration with Luc Forget and David Thomas.

A body of recent work has shown that HLS tools are mature enough to implement advanced arithmetic components such as floating-point [69, 86], the emerging posit competition ([96], Chapter 5), or non-standard application-specific operators (Chapter 3 and 4). When compared to a more classical HDL-based approach such as VFLOAT [101] or FloPoCo [34], this approach of implementing operators in HLS means that new operators can be added via libraries, and instantiated operators can benefit from all the high-level scheduling optimizations performed by HLS compilers.

A key advantage of implementing operators using HLS is that the operators can be both platform-independent and efficient, providing open-source and debuggable operators with similar performance to vendor IP cores. However, this performance relies on the use of highly-optimized vendor-specific libraries for custom-width integers, such as `ap_int` and `ac_int`. Floating-point and posit operators also rely on some less common integer operations, such as leading zero counts and shifting while normalizing. These specific operations have efficient hardware implementations, but the interface as well as the implementation may vary for each target FPGA family and HLS tool. A naive implementation can increase area and reduce performance.

The application programming interface (API) of vendor-specific libraries tends to be designed for ease-of-use, with features such as automatic sign-extension of types, and implicit conversions. Such features are useful for writing DSP applications, where padding only reduces performance. However, when implementing a floating-point operator, each bit matters, so silent padding or conversion of types may mask a logic error or un-handled corner case.

To support the development of truly cross-platform open-source custom operators for HLS, this work introduces a new library called Hint – "hardware integer" – which provides a lightweight type-safe abstraction over vendor libraries. As well as basic integer operations, it also provides optimized implementations of the more obscure integer operations needed for implementing floating-point and posit operators, taking advantage of template techniques to construct optimized datapaths and provide information to the HLS compiler. The main

contributions are:

- A new open-source API called Hint, which provides a platform-independent API and strongly-typed semantics for defining custom-width integer datapaths.
- An extension API for adding new backends by defining a small number of shared operations, and a set of backends for widely used HLS tools.
- New compile-time optimized operators for: shifting and computing stickies; performing leading zero counts (lzc); computing combined shifts and lzc. All these operators use C++ templates to optimize each instance for the exact operand widths and shift values requested.
- An evaluation of the library using two different HLS tools (Vivado HLS and Intel HLS) with the above mentioned operators.
- The implementation of a complete posit adder (from Chapter 5) using Hint.

The proposed API and vendor-specific backends are available as an open-source library at:

`github.com/yuguen/hint`

6.1 Background and motivation

6.1.1 Arbitrary-sized integers for HLS

The support of arbitrary-sized bit vectors is not standard in HLS tools. The nearest to a common standard is the `ac_int` templated C++ library developed at Mentor Graphics [102]. It is supported by the commercial tools Intel HLS and CatapultC, and the academic tool GAUT. However, the HLS tool with the most traction in reconfigurable computing is probably Xilinx Vivado HLS, and it uses a proprietary library called `ap_int` [103]. While `ap_int` and `ac_int` provide almost functionally equivalent support for basic arbitrary-sized signed and unsigned integers, their interfaces are different 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 [39] defines 64 new non-standard base types `int1` to `int64`. Two other academic tools, LegUp [36] and Panda/Bambu [35] 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-bit numbers and bit-masks – hopefully the compiler will truncate it to 17-bits during optimization, but this may not occur until late in the synthesis process, so it will be scheduled as if it were the full 32-bits.

Arbitrary-sized integers are extremely useful when designing custom operators: for example, for double-precision floating-point operators we have 11-bit exponents and 52-bit fractions at the inputs and outputs. Then inside an adder datapath we find a 53-bit explicit fraction and a 56-bit data after effective addition, while for multipliers, we have an intermediate significand product of 106 bits. This was for the standard 64-bit floating-point format, but reconfigurable computing can also take advantage of non-standard formats. Such non-standard floating-point formats have already been expressed using C++ templates [69], but these must be adapted for each HLS vendor's integer library.

6.1.2 Type safety

Another issue is to define the exact meaning of compound expressions involving implicit intermediate types and implicit type conversions. For instance, in the expression `(a+b)+c`, the type of the intermediate result `(a+b)` is implicit and implementation-defined. In addition, libraries such as `ap_int` and `ac_int` use operator overloading to define the types and semantics, so the behavior of the addition is defined in the library implementation, which may silently involve implicit casts if `a` and `b` have different types.

Mainstream tools such as Vivado HLS or CatapultC tend to chose 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}.

Things are a bit trickier with shifts: left shifts may or may not lose the shifted out bits. Right shifts always lose the shifted-out bits, but in the signed case they may perform a sign extension, where the size of the intermediate format will matter. The interested reader is invited to compile and run program from Figure 6.1.

```
#include <iostream>
int main() {
  int a,b,s;
  a = 255;

  s = 31;
  b = (a<<s) >>s;
  std::cout << b << std::endl;

  s = 33;
  b = (a<<s) >>s;
  std::cout << b << std::endl;
}
```

Figure 6.1: C undefined behaviours.

At time of writing, on a Linux 64-bit PC, using Clang or GCC, the first assignment to b computes -1: this can be explained by the fact that all arithmetic is performed in unsigned 32 bits. The second assignment may compute 255 (with optimization level -O0) or 0 (with -O2). 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). We leave it to the reader to check the generated assembly code: our main message is that the ease of use of HLS may hide subtleties that incur bizarre behaviors, but also hidden hardware or latency overheads. The "type-safe" part in the title of this Chapter really means to give back to the designer some control of what is happening in a HLS tool.

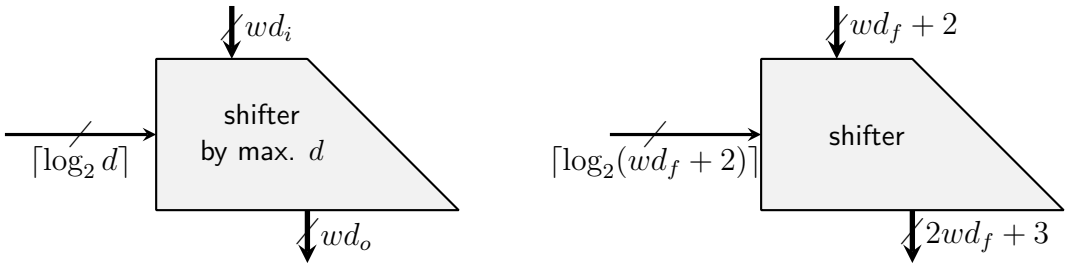


Figure 6.2: A generic shifter (left) instantiated (right) in a floating-point adder with wd_f fraction bits

6.1.3 Core arithmetic primitives for floating-point and posits

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 additions, 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 backend tools that know the target. Division and square root can be implemented either out of additions and tabulation, or out of multiplications. Whether or not this algorithmic choice should be left to the backend tools is out of the scope of this work.

- **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 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 6.2: a shifter may be defined by an input width wd_i, a maximum shift distance d, and an output width wd_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 $wd_o \geq wd_i + d$.

- **Arbitrary precision leading zero counter, leading one counter, and leading bit counter**. The latter counts ones if the leading (leftmost) bit is a one, and counts zeroes if the leading bit is a zero.

6.1.4 Fused arithmetic primitives

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

- A **multiply-add** computes $A \times B + C$.

- **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. This operator incrementally computes the logical OR of all the shifted-out bits (historically called sticky bit). This slightly saves on the latency, and more importantly saves the hardware that would have shifted further all these bits.

- 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.

6.1.5 Support of these primitives in HLS tools

All tools support the primitives which are part of C (addition, multiplication, shifts), although the actual behavior may be implementation-defined in some cases. 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.

6.2 Type safety for arbitrary-precision integers in HLS

Current HLS integer libraries perform very little compile-time sanity checks. This Section describes a Hint variable declaration and its elementary methods along with their type-checking. These basic methods are used to build more complex operators.

6.2.1 Variable declaration

The Hint library is a templated wrapper above underlying integer libraries. A user must write it's function as a template function to be able target all the backends. In that case, it's function will have as template parameter `template<unsigned int , bool> class HintWrapper`. This defines a type that a user can use to declare it's variable: `HintWrapper<W, s> var;` defines a variable `var` on `W` bits and has a sign `s`.

6.2.2 Variable assignment

In vendor tools, assignment to a variable of mismatched size is a cause of silent truncation and potentially unused bits. The proposed library diverges from `ac_int` and `ap_int` as it only allows assignments of identical size variables. Furthermore, it only allows for assignments with matching signedness. The variable assignment is performed by using the usual `"="` operator.

This restriction requires the programmer to explicitly truncate their variables when a part is not needed. A compiler error will be thrown if these properties are not ensured. Enforcing this behavior helped us discover several bugs in our operators.

6.2.3 Slicing

As a preliminary note, it is interesting to remark that the 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 `l` of the variable `var` is written `var.slc<S>(l)`. Conversely, the `ap_int` slicing method is `var.range(h,l)` where `h` and `l` 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. Therefore, a slice whose size varies in a loop will compile using `ap_int` but not using `ac_int`.

Neither of these two libraries is able to check at compile time if the slice is out of bounds. By having `S` as a template parameter, `ac_int` ensures that the size of the output is known at compile time. However, if the user assigns the result of the slice to a larger or smaller variable, the result might be truncated without warning. When using `ap_int`, no compile-time checks can be performed. The result of an out-of-bound slice will have some of its bits set to 0.

For Hint, we choose to be even more restrictive in order to allow checks at compile-time that slices are in range. A `Hint` slice is of the form `var.slice<h,l>()` where `h` and `l` are the weights of the msb and lsb of the slice of `var`. As `h` and `l` are known at compile time, sanity checks are performed and the output size is known. Therefore the program won't compile if `h<l` or `l<0` or if `h>=W` with `W` being the size of `var`. The size of the returned integer is `h-l+1`. It cannot be truncated implicitly, as we have seen that assignments are only allowed between matching sizes variables.

6.2.4 Concatenation

Both libraries are able to know at compile time the size of the result of a concatenation. However, there may still be a silent truncation when assigning this result to a smaller variable. The proposed concatenation method is `var1.concatenate(var2);` where the result size is the sum of the sizes of `var1` and `var2`.

6.2.5 Others

The Hint API can be extended with any other methods with the same spirit that all types must be checked at compile time. For example, the current implementation contains:

- bitwise operations such as and, or, xor that from two identical W-bit width variables returns a W-bit variable containing the corresponding bitwise operation:

  e.g. `var1.bitwise_and(var2)`

- and/or reductions that return a 1-bit result:

  e.g. `var1.and_reduction()`

- a signal inverter that transforms each bit to its opposite:

  `var1.invert()`

- an operator that computes the reverse of a variable (the lsb takes the msb and so on):

```
var1.backwards()
```

- a padding operator that performs the extension of a Hint variable to a larger one; only available if the result size is larger than the original size:

```
var1.leftpad<newsize>()
```

- a generator of a sequence of a given length containing identical bits:

e.g. `HintWrapper<W, s>::generateSequence(bit)`

  where `bit` is the single bit to be replicated.

- a sign inverter:

```
var1.invert()
```

- an equality operator `"=="` that only compares identical width and sign Hint variables
- a multiplexer operator that takes a control bit and two identical width variables:

```
mux(control, var1, var2)
```

- a modular addition (`var1.modularAdd(var2)`) that performs the sum on two `W`-bit variables and return the sum of these two values on `W` bits when the user knows the addition won't overflow. Similarly, Hint also provides a modular subtraction (`var1.modularSub(var2)`)
- an adder with carry that takes two `W`-bit variables with a 1-bit carry-in and returns a sum of the three on `W+1` bits:

```
var1.addWithCarry(var2, carry)
```

All the types are safely deduced by the compiler using the `auto` keyword as each operator returns a specific type depending on its inputs.

This list is subject to grow, following applications needs, hence requiring each backend to implement these basic functionalities. However, this set of basic functionalities allows for building higher-level operators.

6.3 Portability to mainstream HLS tools

There are several ways to implement vendor-specific backends for the proposed interface. This section presents three approaches, each with their pros and cons.

6.3.1 Class duplication for each backend

In order to have compile-time decision of which backend to implement, one can simply enable/disable the Hint class definitions depending on environment variables (using `#if`-based conditional compilation) . This implementation is the less elegant way of implementing such a portability layer. It is also the most error prone: bit-for-bit portability relies on each backend implementing the same methods and the same static verification semantics, despite sharing no code and compiling to different libraries.

This poor engineering approach was nevertheless used to check the feasibility of a portability layer. A complete posit adder implementation initially written for `ap_int` was ported to Hint (from Chapter 5). When compiled with Vivado HLS, no degradation of the quality of results was observed. Meanwhile, the posit operator could now be compiled with Intel HLS.

6.3.2 A shared class interface

The approach of using a conventional interface that each backend follows is a bit more elegant. There is an implicit interface that each backend must implement in order for the operators built upon to compile. This is also true regarding the static verifications which are replicated in each backend. Thus two backends might not perform the same static verifications.

6.3.3 Curiously recurring template pattern (CRTP)

An elegant way of centralizing the static checks is to use a CRTP class [104]. A front class is provided to the user for instantiating a custom Hint. It is templated by a width, a sign and, a backend. A given backend inherits from the associated specialized Hint class. Therefore the Hint class is the frontend of the library. It implements all the API methods and is in charge of performing the static verifications. If such verifications are satisfied, a call to the underlying backend implementing the same method is issued.

This approach allowed for correct software simulations for both Vivado HLS and Intel HLS. Indeed, this approach only uses features from C++11. Unfortunately, synthesis results showed here that both tools were unable to pipeline complex operators in this case. Further investigations showed that a templated Hint function, when implemented as a CRTP, results in a monolithic block that cannot be pipelined. The recursive template calls insert registers that the optimizer is unable to remove, resulting in a high latency, resource hungry operator.

We expect that future versions of the vendor tools will catch up. In the meantime, the second approach will be used. There is also some longer-term hope that “concepts” introduced in C++20 will make CRTP useless.

6.4 Evaluation

All the presented results are given after place-and-route. Vivado HLS 2016.3[1] was used when targeting Kintex 7; Intel HLS 19.1 when targeting Arria 10.

The evaluation is divided in four parts: ensuring that no overhead is generated, implementing combined operators that reduce resource consumption, then latency, and demonstrate the Hint library a larger project.

Table 6.1: Synthesis of lzc and shifters on Arria 10 (achieved clock target of 240MHz).

	N	ALMs	FFs	MLABs	cycles
native type lzc	26	32.5	32	0	1
	55	86.5	91	1	5
	256	465.5	710	1	8
hint type lzc	26	32.5	32	0	1
	55	86.5	91	1	5
	256	465.5	710	1	8
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 lzc + shift	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

The overhead evaluation is performed on the implementation of a lzc. The lzc algorithm chosen in this paper has been implemented using `ac_int`, `ap_int` and Hint. The synthesis results are given in Tables 6.1 (top) and 6.2 (top) for Intel and Xilinx respectively. Vivado HLS provides a builtin lzc, which is also presented here. The sizes (N) of the inputs corresponds to real world examples. Indeed, 26 and 55 bits are the width of the lzc needed in single and double precision floating-point adders while a 256-bit lzc is needed for a 32-bit posit quire.

The comparison between the native type implementation and the Hint type implementation shows no overhead when using Hint. Furthermore, the implemented lzc algorithm outperforms the Vivado HLS builtin lzc both in term of resources and latency.

The first combined operator presented is the shifter+sticky; reducing resource consumption. The shifted out bits are not discarded, but ORed in a "sticky" bit. The fused operator attempts to OR these bits inside of the shifter, before they are shifted out. This saves the logic that otherwise shift these bits to their final place before the final wide OR.

[1]This older version of Vivado HLS is used because 2018.3, the latest version, at the time of writing, proved very unstable, with numerous crashes and sometimes silent production of incorrect hardware. Vivado HLS 2016.3 is the best compromise between stability and quality of results in our case.

Table 6.2: Synthesis of lzc and shifters on Kintex 7 (achieved target delay of 3ns).

	N	LUTs	FFs	SRLs	cycles
builtin lzc	26	50	81	0	4
	55	85	111	0	4
	256	475	559	11	9
native type lzc	26	26	57	1	4
	55	68	87	10	5
	256	233	516	5	7
hint type lzc	26	26	57	1	4
	55	69	87	11	5
	256	234	516	5	7
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 lzc + shift	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

Figure 6.3 illustrates the not combined operator with a toy shifter that can shift up to 7 bits. In a first stage, the input register (here on 4 bits) can be shifted by 0 or 4. A second stage then takes that new register (now on 8 bits) and shift it by 0 or 2, etc. The final value is obtained by keeping the 4 most significant bits of the result and computing a sticky on the discarded bits. Here the total register footprint of the operator is $4+8+10+11 = 33$ bits. In reality, when using the standard C "`>>`" for computing the shifter followed by a sticky, the initial 4-bit input must be extended to the exact output format (here 11 bits). The shift stages then all operates on 11 bits, making the register footprint even higher (here $11+11+11+11 = 44$ bits).

The fused shifter and sticky used in this work is depicted by Figure 6.4. After the first shift by 4, a partial sticky is computed. Therefore, the next stage only needs to shift a 4-bit register and compute the following partial sticky. By doing so, the register footprint of the architecture is in this case $4+8+6+5 = 23$ bits, plus the partial stickies for a total of 25 bits. A very little cost is added in OR gates to compute the partial stickies, but it is highly mitigated.

The synthesis results of the shifter+sticky are presented in Tables 6.3 and 6.4. The sizes (N) of the operators comes from the floating-point adder in single (27 bits) and double (56 bits) precision. As both tables show, this optimization saves a considerable amount of logic on both targets. Table 6.3 does not report the number of cycles required for said operators. Indeed, Intel HLS was giving untrustworthy latency results. However, the circuits were cosimulated to ensure that they produced the correct mathematical results

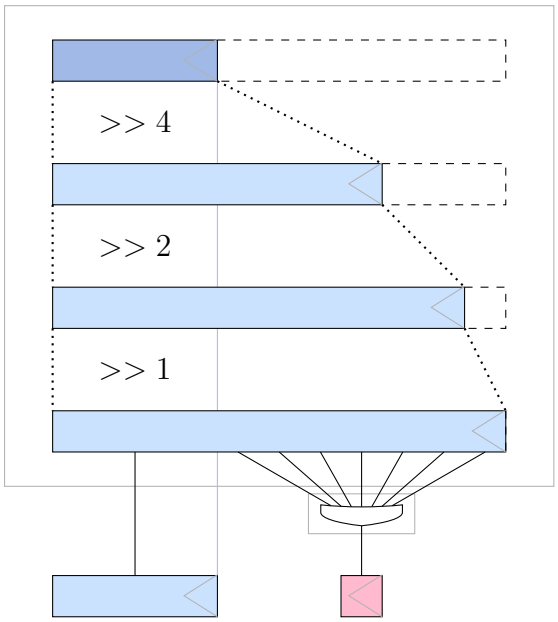


Figure 6.3: Scheme of the architecture of a shifter followed by a sticky (4-bit register shifted up to 7 bits).

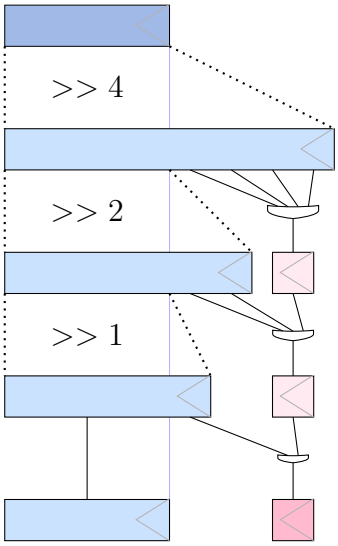


Figure 6.4: Scheme of the architecture of a combined shifter and sticky (4-bit register shifted up to 7 bits).

Table 6.3: Synthesis of shifters+stickies on Arria 10 (achieved clock target of 240MHz).

	N	ALMs	FFs	MLABs
native shift + sticky	27	134	63	2
	56	277	212	3
hint shift + sticky	27	82	40	0
	56	179.5	128	0

using ModelSim.

The second operator is a combined lzc+shifter that reduces latency compared to a serial implementation. A lzc is usually followed by a shift as mostly used in floating-point or posit normalizers. Combining these two operators allows to reduce the latency of the design. Indeed, both the shift and the lzc are divided in stages where each stage can perform a lzc step as well as a shift step. This removes the data dependency of the complete lzc computation before issuing the shift at the expense of more logic.

To evaluate such an operator, a combined Hint lzc+shift is compared to a Hint lzc followed by a native shift (>>). Tables 6.1 (bottom) and 6.2 (bottom) provide synthesis results of these implementations for both Intel and Xilinx FPGAs. In addition to sizes previously presented for the lzc, we added the sizes of a 16-bit quire normalizer (64 bits), a 32-bit quire normalizer (256 bits) and of a 32-bit Kulisch accumulator normalizer (279 bits).

For both Intel and Xilinx, the latency of the combined lzc+shift is improved compared to a serial implementation. In some cases, the tools are even able to reduce resource consumption. This might be due to them being able to compress multiple stages of

Table 6.4: Synthesis of shifters+stickies on Kintex 7 (achieved target of 3ns).

	N	LUTs	FFs	Cycles
native shift + sticky	27	113	110	3
	56	309	234	4
hint shift + sticky	27	84	65	3
	56	203	133	3

lzc+shift in a better way than with a separated design.

Finally, the complete posit adder from Chapter 5 has been rewritten using Hint without overhead compared to the original version written using `ap_int`. The architecture involved the use of two lzc+shift and two shifters+stickies as long as additions, comparisons, logic functions, concatenation, etc.

6.5 Discussion

This Chapter provides an open-source portability layer for custom-size integer datatypes called Hint. It is strongly typed: no information is lost or useless bits appended when performing operations on Hints without an explicit programmer request. This is enforced at compile time with static type checks from the C++ compiler. These static checks follow a well defined semantic, making every operation explicit about the types it manipulates. This may sound very restrictive, but ultimately, most Hint variables can safely use the `auto` C++ type, as all the widths and signedness are derived and checked from the inputs.

The Hint library allows one to write a single operator that can be synthetized using different vendor platforms. The component can also be efficiently simulated using a GMP backend. Using Hint, the computation results are guaranteed to be identical on every platform.

Hint does not induce any overhead when using mainstream HLS tools compared to their native types implementations.

In general, work on this Chapter has been considerably slowed down by vendor tools limitations. They are currently unable to equate identical variable through the template layers involved by the best implementation, although the programs are accepted and properly simulated within the tools. In addition, depending on the C++ construction used, two functionally equivalent programs can result in drastically different synthesized hardware.

These issues are being understood and ironed out. The wider goal is to build a larger collection of sophisticated arithmetic operators upon this set of trusted operations. The first step towards this will be to complete the port of existing open-source HLS works such as floating-points [69] or posit operators (from Chapter 5). The next step will then be to apply the same concepts to fixed-point formats (wrapping `ac_fixed` and `ap_fixed`).

A longer-term objective is to build on the clear semantic of every basic operation to build formal proofs of the correct behavior of the hardware.

Chapter 7

Conclusion and future work

7.1 Conclusion

Ten years ago, high-level synthesis tools would not have been considered as an alternative to hand-writing HDL for designing optimized operators. Throughout this thesis, we proved that they are now mature enough to compete. Furthermore, their high-level/behavioural view of a component enables compiler optimizations that were not possible when only using synthesis tools.

Due to there relatively young nature, high-level synthesis tools still lack some arithmetic optimizations. Indeed, they mostly rely on their underlying x86 compiler backend that is written for optimizing CPU compute speed. Chapter 3 showcases this sub-optimal implementation. It demonstrates safe arithmetic optimizations that should be applied in every scenario, such as application-specific floating-point multiplication and division by small constants. It then argues that by considering floating-points as real numbers, the generated circuits can greatly improve the accuracy/cost/performance ratio. To demonstrate this claim, the sum-of-product of floating-point numbers is used as a case study. The accumulation is performed on a custom fixed-point format that is tailored to the context by the use of compiler directives.

When the application context does not allow to tailor the fixed-point format, a large accumulator covering the entire floating-point range is generated. The latter still improves both the accuracy and the performance metrics compared to the regular floating-point accumulation. This idea is not new and was already well studied in the 90's by Kulisch. However, the systematic use of sign+magnitude encoding in the recent literature to implement such an accumulator was questionable. Therefore, Chapter 4 studied in details different implementations of the Kulisch accumulator. The use of 2's complement is showed to reduce its cost while increasing the maximum frequency at which it can operate.

Based on a tapered precision scheme and an exact accumulator, the posit number systems claims to be a candidate to replace the IEEE floating-point format. The increased precision for small exponents and this exact arithmetic opportunity could increase the accuracy some applications. However, the hardware cost evaluation of such a number system was too little to enable fair comparisons with floating-points. In Chapter 5, a throughout analysis of posit operators is performed, using the same level of hardware optimization as state-of-the-art floating-point operators. Their cost is showed to remain much higher than their floating-point counterparts in terms of resource usage and performance. Furthermore, the posit quire is shown to have a similar cost as the floating-point Kulisch accumulator.

Hence, using a floating-point format, augmented with a Kulisch accumulator, might be sufficient for most applications, while being faster than posits for regular operators.

The presented HLS operators all rely on custom size integer libraries that allow for custom width data-path. However, each HLS tool uses a specific integer library.An experiment in Chapter 5 showed that using a vendor HLS tool to generate an HDL description before using a synthesis tool from another vendor results in suboptimal designs. Furthermore, the direct translation from one library to another is difficult as they do not follow the same semantic. Plus, some base operators are available in some HLS tools but not others. To enable easy comparisons on HLS operators on any FPGA target, Chapter 6 presented a custom size integer library for HLS tools that allows one code to be deployed on multiple tools. In addition to the enhanced compatibility, the library is strongly typed. Indeed, in a hardware context where every bit matters, silent bit truncations or undefined behaviours should be avoided.

All the presented work is open-source:

- Chapter 3: the source-to-source transformations are available in the `gecos-arith` sub-project of the GeCoS compiler (`gitlab.inria.fr/gecos`)
- Chapter 4 and 5: the parametric architectures of the different variants of the Kulisch accumulators and the posit operators are available as C/C++ libraries in the *Modern arithmetic tools (Marto)* project (`gitlab.inria.fr/lforget/marto`)
- Chapter 6: the Hint strongly typed custom size integer library and the optimized operators are available at `github.com/yuguen/hint`

7.2 Future work

More arithmetic optimizations in HLS tools In this work, we used a source-to-source approach which enables optimizations on top of closed source HLS tools. One could also integrate the proposed transformations directly in HLS tools underlying compilers. Implementing optimizations for application-specific operators only require compilers to select the correct operator at instruction selection. However, designing a compiler pass (e.g. for LLVM) would only make the optimization available for HLS tools built on that specific compiler framework. Nevertheless, building compiler passes and a compiler backend designed from the ground up for HLS is probably the best solution regarding the quality of the designs.

Question the use of IEEE floating-points in HLS tools The IEEE-754 standard, followed by these high-level languages, was designed for general purpose hardware. As demonstrated in Chapter 3, not following that standard can lead to faster/cheaper/more accurate hardware. The approach followed in this thesis was to consider `float` variables as real numbers, hence not following the high-level language original semantic, for ease-of-use. However, one could consider building a new type for HLS, such as `real`, that would be interpreted by HLS tools as a different type than floating-points. This custom type could embed range and accuracy specifications to enable further compiler optimizations. Alternatively, the high-level language could be dropped completely for another high-level domain-specific language.

Exploit exact floating-point accumulators in non-traditional scenarios Now that exact floating-point accumulators described in Chapter 4 are able to fit and operate at high clock speed in FPGAs, more complex designs around their use can be considered. In a recent work, [105] proposed to accelerate convolution layers in convolutional neural networks by overclocking the FPGA. A lightweight fault tolerance mechanism is used to identify and recover from a fault due to too tight timings. This mechanism is based on integer checksums, and cannot be performed directly using floating-point arithmetic. The use of custom accumulators could enable such techniques for floating-point computations.

Explore number representations using the C++ template engine The C++ template engine used to explore posit operators in Chapter 5 proved very useful for (a) fast architecture exploration and (b) making the parametric operators available through a library. The latter can be used to evaluate the cost of the many recently proposed alternatives to standard floating-points [97, 98, 99, 106].

Add new operators to the Hint library All the presented work can be made available for multiple HLS tools by using the Hint library from Chapter 6. This will also help identify new primitives that are not yet implemented in Hint.

The work presented in this thesis aims at improving the HLS generated designs that involves arithmetic computations. This can only be done by bridging the gap between the compiler designers and the arithmeticians community. This thesis is a step towards that direction.

This thesis led to several publications:

- Chapter 3: This work was first presented as a poster at the Compas'2016 french conference [107], then in 2017 as a poster at the FCCM conference [108] before being presented as a full paper at the 2017 FPL conference [109]. It is now under review for the TACO journal.
- Chapter 4: This work was presented as a full paper at the Compas'2017 french conference [110]. A journal version is yet to be submitted to TRETS.
- Chapter 5: This work is the aggregation of two full papers, one at the 2019 CoNGA conference [93] and the other at the 2019 FPL conference [111]. It has also been presented to the Compas'2019 french conference [112].
- Chapter 6: This work was presented at the 2019 HEART conference [113].
- During this thesis, and not presented in this document, a collaboration with the Intel Exascale Computer Research laboratory led to a publication at the 2018 AI-SEPS workshop [114].

Bibliography

[1] *IEEE standard for floating-point arithmetic*. 2008.

[2] Fons Lange, Alfred van der Hoeven, E.F. Deprettere, and J Bu. "An optimal floating-point pipeline CMOS CORDIC processor". In: *International symposium on circuits and systems*. IEEE, 1988, pp. 2043–2047.

[3] Robert K. Montoye, Erdem Hokenek, and Stephen L. Runyon. "Design of the IBM RISC System/6000 floating-point execution unit". In: *Journal of research and development* 34.1 (1990), pp. 59–70.

[4] Andrew Beaumont-Smith, Neil Burgess, S Lefrere, and Cheng-Chew Lim. "Reduced latency IEEE floating-point standard adder architectures". In: *14th symposium on computer arithmetic*. IEEE, 1999, pp. 35–42.

[5] Guy Even, Silvia M Mueller, and Peter-Michael Seidel. "A dual precision IEEE floating-point multiplier". In: *Integration, the vlsi journal* 29.2 (2000), pp. 167–180.

[6] Libo Huang, Li Shen, Kui Dai, and Zhiying Wang. "A new architecture for multiple-precision floating-point multiply-add fused unit design". In: *18th symposium on computer arithmetic*. IEEE, 2007, pp. 69–76.

[7] Jongwook Sohn and Earl E Swartzlander. "Improved architectures for a fused floating-point add-subtract unit". In: *Transactions on circuits and systems i* 59.10 (2012), pp. 2285–2291.

[8] Konstantinos Manolopoulos, D Reisis, and Vassilios A Chouliaras. "An efficient multiple precision floating-point multiply-add fused unit". In: *Microelectronics journal* 49 (2016), pp. 10–18.

[9] David R. Lutz. "ARM floating-point 2019: latency, area, power". In: *26nd symposium on computer arithmetic*. IEEE, 2019, pp. 69–76.

[10] Nabeel Shirazi, Al Walters, and Peter Athanas. "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines". In: *International symposium on fpgas for custom computing machines*. IEEE, 1995, pp. 155–162.

[11] Jian Liang, Russell Tessier, and Oskar Mencer. "Floating point unit generation and evaluation for FPGAs". In: *11th symposium on field-programmable custom computing machines*. IEEE, 2003, pp. 185–194.

[12] Kenneth David Chapman. "Fast integer multipliers fit in FPGAs". In: *Edn magazine* 39.10 (1994), pp. 80–80.

[13] H Fatih Ugurdag, Florent De Dinechin, Y Serhan Gener, Sezer Goren, and Laurent-Stéphane Didier. "Hardware division by small integer constants". In: *Transactions on computers* 66.12 (2017), pp. 2097–2110.

[14] Florent de Dinechin, Silviu-Ioan Filip, Luc Forget, and Martin Kumm. "Table-based versus shift-and-add constant multipliers for FPGAs". In: *26nd symposium on computer arithmetic*. IEEE, 2019, pp. 151–158.

[15] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. "Ternary neural networks for resource-efficient AI applications". In: *International joint conference on neural networks*. IEEE, 2017, pp. 2547–2554.

[16] Anastasia Volkova, Matei Istoan, Florent De Dinechin, and Thibault Hilaire. "Towards hardware IIR filters computing just right: direct form I case study". In: *Transactions on computers* 68.4 (2018), pp. 597–608.

[17] Torbjorn Granlund. *GNU MP: the GNU multiple precision arithmetic library 6.1.2*. 2016.

[18] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Symposium on principles of programming languages*. ACM, 1977, pp. 238–252.

[19] Seehyun Kim and Wonyong Sung. "A floating-point to fixed-point assembly program translator for the TMS 320C25". In: *Transactions on circuits and systems II* 41.11 (1994), pp. 730–739.

[20] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. "Fixed-point optimization utility for C and C++ based digital signal processing programs". In: *Transactions on circuits and systems II* 45.11 (1998), pp. 1455–1464.

[21] Matthieu Martel. "Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics". In: *Formal methods in system design* 35.3 (2009), pp. 265–278.

[22] Olivier Sentieys, Daniel Menard, David Novo, and Karthick Parashar. "Automatic fixed-point conversion: a gateway to high-level power optimization". In: *Tutorial at ieee/acm design automation and test in europe*. 2014.

[23] Jean-Michel Müller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of floating-point arithmetic, 2nd edition*. Birkhauser Boston, 2018.

[24] John Hauser. "Berkeley softfloat". In: *Private communications* (2002).

[25] Stephen Trimberger. "Effects of FPGA architecture on FPGA routing". In: *Design automation conference*. IEEE, 1995, pp. 574–578.

[26] Michael J Alexander and Gabriel Robins. "New performance-driven FPGA routing algorithms". In: *Transactions on computer-aided design of integrated circuits and systems* 15.12 (1996), pp. 1505–1517.

[27] Vaughn Betz and Jonathan Rose. "VPR: a new packing, placement and routing tool for FPGA research". In: *International workshop on field programmable logic and applications*. Springer, 1997, pp. 213–222.

[28] Martin Langhammer and Bogdan Pasca. *Floating-point adder circuitry with sub-normal support*. US Patent App. 15/704,313. 2019.

[29] Jan Decaluwe. "MyHDL: a python-based hardware description language." In: *Linux journal* 127 (2004), pp. 84–87.

[30] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: constructing hardware in a scala embedded language". In: *Design automation conference*. IEEE, 2012, pp. 1212–1221.

[31] Charles E Leiserson and James B Saxe. "Retiming synchronous circuitry". In: *Algorithmica* 6.1-6 (1991), pp. 5–35.

[32] Tamotsu Akashi. *Retiming circuit and method for performing retiming*. US Patent 6,178,212. 2001.

[33] Girish Venkataramani and Yongfeng Gu. "System-level retiming and pipelining". In: *22nd international symposium on field-programmable custom computing machines*. IEEE, 2014, pp. 80–87.

[34] Florent De Dinechin and Bogdan Pasca. "Designing custom arithmetic data paths with FloPoCo". In: *Design & test of computers* 28.4 (2011), pp. 18–27.

[35] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. "A survey and evaluation of FPGA high-level synthesis tools". In: *Transactions on computer-aided design of integrated circuits and systems* 35.10 (2015), pp. 1591–1604.

[36] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. "LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems". In: *Transactions on embedded computing systems* 13.2 (2013), p. 24.

[37] Christian Pilato and Fabrizio Ferrandi. "Bambu: a modular framework for the high level synthesis of memory-intensive applications". In: *23rd international conference on field programmable logic and applications*. IEEE, 2013, pp. 1–4.

[38] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. "GAUT: a high-level synthesis tool for DSP applications". In: *High-level synthesis: from algorithm to digital circuit*. Springer, 2008, pp. 147–169.

[39] Adrien Prost-Boucle, Olivier Müller, and Frédéric Rousseau. "A fast and stand-alone HLS methodology for hardware accelerator generation under resource constraints". In: *Journal of systems architecture* (2014).

[40] Michael Fingeroff. *High-level synthesis: blue book*. Xlibris Corporation, 2010.

[41] Sumit Gupta, Rajesh Gupta, Nikil D Dutt, and Alexandru Nicolau. *SPARK: a parallelizing approach to the high-level synthesis of digital circuits*. Springer, 2007.

[42] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. "Parallel programming for FPGAs". In: *Arxiv e-prints* (2018). eprint: 1805.03648.

[43] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[44] Robert Morgan. *Building an optimizing compiler*. Digital Press, 1998.

[45] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Vol. 289. Morgan Kaufmann, 2002.

[46] Andrew G Dempster and Malcolm D Macleod. "Constant integer multiplication using minimum adders". In: *Iee proceedings-circuits, devices and systems* 141.5 (1994), pp. 407–413.

[47] Oscar Gustafsson. "Lower bounds for constant multiplication problems". In: *Transactions on circuits and systems ii* 54.11 (2007), pp. 974–978.

[48] Yevgen Voronenko and Markus Püschel. "Multiplierless multiple constant multiplication". In: *Transactions on algorithms* 3.2 (2007), p. 11.

[49] Levent Aksoy, Eduardo Costa, Paulo Flores, and Jose Monteiro. "Optimization of area in digital FIR filters using gate-level metrics". In: *44th design automation conference*. IEEE, 2007, pp. 420–423.

[50] Jason Thong and Nicola Nicolici. "An optimal and practical approach to single constant multiplication". In: *Transactions on computer-aided design of integrated circuits and systems* 30.9 (2011), pp. 1373–1386.

[51] Martin Kumm, Oscar Gustafsson, Mario Garrido, and Peter Zipf. "Optimal single constant multiplication using ternary adders". In: *Transactions on circuits and systems ii* 65.7 (2018), pp. 928–932.

[52] Michael J Wirthlin. "Constant coefficient multiplication using look-up tables". In: *Journal of VLSI signal processing systems for signal, image and video technology* 36.1 (2004), pp. 7–15.

[53] E. George Walters. "Reduced-area constant-coefficient and multiple-constant multipliers for xilinx FPGAs with 6-input LUTs". In: *Electronics* 6.4 (2017), p. 101.

[54] Antoine Floc'h, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, Mythri Alle, Ludovic L'Hours, Nicolas Simon, Steven Derrien, et al. "GeCoS: a framework for prototyping custom hardware design flows". In: *13th international working conference on source code analysis and manipulation*. IEEE, 2013, pp. 100–105.

[55] ISO. *C11 standard*. ISO/IEC 9899:2011. 2011. URL: `/bib/iso/C11/n1570.pdf`.

[56] Florent de Dinechin. "Multiplication by rational constants". In: *Transactions on circuits and systems ii* 59.2 (2012), pp. 98–102.

[57] Bogdan Pasca. "Correctly rounded floating-point division for DSP-enabled FPGAs". In: *22nd international conference on field programmable logic and applications*. IEEE, 2012, pp. 249–254.

[58] Nicolas Brisebarre, Florent De Dinechin, and Jean-Michel Müller. "Integer and floating-point constant multipliers for fpgas". In: *International conference on application-specific systems, architectures and processors*. IEEE, 2008, pp. 239–244.

[59] Louis-Noël Pouchet. *Polybench: the polyhedral benchmark suite*. 2012. URL: `www.cs.ucla.edu/pouchet/software/polybench`.

[60] James Hrica. *Floating-point design with vivado HLS*. Xilinx Application Note. 2012.

[61] Nachiket Kapre and André DeHon. "Optimistic parallelization of floating-point accumulation". In: *18th symposium on computer arithmetic*. IEEE, 2007, pp. 205–216.

[62] Marcel Gort and Jason H Anderson. "Range and bitmask analysis for hardware optimization in high-level synthesis". In: *18th asia and south pacific design automation conference*. IEEE, 2013, pp. 773–779.

[63] Gabriel Caffarena, Juan A Lopez, Carlos Carreras, and Octavio Nieto-Taladriz. "High-level synthesis of multiple word-length DSP algorithms using heterogeneous-resource FPGAs". In: *International conference on field programmable logic and applications*. IEEE, 2006, pp. 1–4.

[64] Ulrich Kulisch and Van Snyder. "The exact dot product as basic tool for long interval arithmetic". In: *Computing* 91.3 (2011), pp. 307–313.

[65] Florent de Dinechin, Bogdan Pasca, Octavian Cret, and Radu Tudoran. "An FPGA-specific approach to floating-point accumulation and sum-of-products". In: *International conference on field-programmable technology*. IEEE, 2008, pp. 33–40.

[66] Florent de Dinechin and Bogdan Pasca. "Reconfigurable arithmetic for high-performance computing". In: *High-performance computing using FPGAs*. Springer, 2013. Chap. Reconfigurable Arithmetic for High-Performance Computing, pp. 631–663.

[67] Zhen Luo and Margaret Martonosi. "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques". In: *Transactions on computers* 49.3 (2000), pp. 208–218.

[68] Edin Kadric, Paul Gurniak, and André DeHon. "Accurate parallel floating-point accumulation". In: *Transactions on computers* 65.11 (2016), pp. 3224–3238.

[69] David Thomas. "Templatised soft floating-point for high-level synthesis". In: *27th international symposium on field-programmable custom computing machines*. IEEE, 2019.

[70] Xavier Redon and Paul Feautrier. "Detection of scans". In: *Parallel algorithms and applictations* 15.3-4 (2000), pp. 229–263.

[71] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. "Polly's polyhedral scheduling in the presence of reductions". In: *International workshop on polyhedral compilation techniques* (2015).

[72] *EEMBC - the embedded microprocessor benchmark consortium*. `http://www.eembc.org/`.

[73] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. "MPFR: a multiple-precision binary floating-point library with correct rounding". In: *Transactions on mathematical software* 33.2 (2007), p. 13.

[74] PR Capello and Willard L. Miranker. "Systolic super summation". In: *Transactions on computers* 37.6 (1988), pp. 657–677.

[75] Reinhard Kirchner and Ulrich Kulisch. "Accurate arithmetic for vector processors". In: *Journal of parallel and distributed computing* 5.3 (1988), pp. 250–270.

[76] Juergen Kernhof, Christoph Baumhof, Bernd Hofflinger, Ulrich Kulisch, Steve Kwee, Peter Schramm, Manfred Selzer, and Thomas Teufel. "A CMOS floating-point processing chip for verified exact vector arithmetic". In: *European solid-state circuits conference*. IEEE, 1994, pp. 196–199.

[77] Ulrich Kulisch. *Computer arithmetic and validity: theory, implementation, and applications*. Walter de Gruyter, 2013.

[78] Andreas Knofel. "Fast hardware units for the computation of accurate dot products". In: *10th symposium on computer arithmetic*. IEEE, 1991, pp. 70–74.

[79] Michael Müller, Christine Rub, and W Rulling. "Exact accumulation of floating-point numbers". In: *10th symposium on computer arithmetic*. IEEE, 1991, pp. 64–69.

[80] Michael B Taylor. "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse". In: *Design automation conference*. IEEE, 2012, pp. 1131–1136.

[81] Fabian Nowak and Rainer Buchty. "A tightly coupled accelerator infrastructure for exact arithmetics". In: *International conference on architecture of computing systems*. Springer, 2010, pp. 222–233.

[82] Jack Koenig, David Biancolin, Jonathan Bachrach, and Krste Asanovic. "A hardware accelerator for computing an exact dot product". In: *24th symposium on computer arithmetic*. IEEE, 2017, pp. 114–121.

[83] Luis Fiolhais and Horacio Neto. "An efficient exact fused dot product processor in FPGA". In: *28th international conference on field programmable logic and applications*. IEEE, 2018, pp. 327–3273.

[84] Neil Burgess, Chris Goodyer, Chris Hinds, and David Lutz. "High-precision anchored accumulators for reproducible floating-point summation". In: *Transactions on computers* (2018).

[85] Nicolas Brunie. "Modified fused multiply and add for exact low precision product accumulation". In: *24th symposium on computer arithmetic*. IEEE, 2017, pp. 106–113.

[86] Samridhi Bansal, Hsuan Hsiao, Tomasz Czajkowski, and Jason H Anderson. "High-level synthesis of software-customizable floating-point cores". In: *Design, automation & test in europe conference & exhibition*. IEEE, 2018, pp. 37–42.

[87] David Wilson and Greg Stitt. "The unified accumulator architecture: a configurable, portable, and extensible floating-point accumulator". In: *Transactions on reconfigurable technology and systems* 9.3 (2016), p. 21.

[88] John L Gustafson and Isaac T Yonemoto. "Beating floating point at its own game: posit arithmetic". In: *Supercomputing frontiers and innovations* 4.2 (2017), pp. 71–86.

[89] Posit Working Group. *Posit standard documentation*. Release 3.2-draft. 2018.

[90] Jeff Johnson. "Rethinking floating point for deep learning". In: *Arxiv preprint 1811.01721* (2018).

[91] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. "Performance-efficiency trade-off of low-precision numerical formats in deep neural networks". In: *Conference for next generation arithmetic*. ACM, 2019, 3:1–3:9.

[92] Jianyu Chen, Zaid Al-Ars, and H Peter Hofstee. "A matrix-multiply unit for posits in reconfigurable logic leveraging (open) CAPI". In: *Conference for next generation arithmetic*. ACM, 2018, p. 1.

[93] Florent De Dinechin, Luc Forget, Jean-Michel Müller, and Yohann Uguen. "Posits: the good, the bad and the ugly". In: *Proceedings of the conference for next generation arithmetic*. ACM, 2019, p. 6.

[94] Rohit Chaurasiya, John Gustafson, Rahul Shrestha, Jonathan Neudorfer, Sangeeth Nambiar, Kaustav Niyogi, Farhad Merchant, and Rainer Leupers. "Parameterized posit arithmetic hardware generator". In: *36th international conference on computer design*. IEEE, 2018, pp. 334–341.

[95] Manish Kumar Jaiswal and Hayden K-H So. "PACoGen: a hardware posit arithmetic core generator". In: *Ieee access* 7 (2019), pp. 74586–74601.

[96] Artur Podobas and Satoshi Matsuoka. "Hardware implementation of POSITs and their application in FPGAs". In: *International parallel and distributed processing symposium workshops*. IEEE, 2018, pp. 138–145.

[97] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. "Leveraging the Bfloat16 artificial intelligence datatype for higher-precision computations". In: *26nd symposium on computer arithmetic*. 2019, pp. 97–98.

[98] Neil Burgess, Nigel Stephens, Jelena Milanovic, and Konstantinos Monachopolous. "Bfloat16 processing for neural networks". In: *26nd symposium on computer arithmetic*. IEEE, 2019, pp. 88–91.

[99] Ankur Agraval, Silvia M. Mueller, Bruce M. Fleischer, Jungwook Choi, Naigang Wang, Xiao Sun, and Kailash Gopalakrishman. "DLFloat: a 16-b floating point format designed for deep learning training and inference". In: *26nd symposium on computer arithmetic*. IEEE, 2019, pp. 92–95.

[100] *Stratix-V device handbook*. Altera Corporation. 2013.

[101] Xiaojun Wang, Sherman Braganza, and Miriam Leeser. "Advanced components in the variable precision floating-point library". In: *14th symposium on field-programmable custom computing machines*. IEEE, 2006, pp. 249–258.

[102] Andres Takach. *Algorithm c (AC) datatypes*. 2018.

[103] *Vivado design suite user guide: high-level synthesis (UG902)*. 2018.

[104] James O Coplien. "Curiously recurring template patterns". In: *C++ report* 7.2 (1995), pp. 24–27.

[105] Thibaut Marty, Tomofumi Yuki, and Steven Derrien. "Enabling overclocking through algorithm-level error detection". In: *International conference on field-programmable technology*. IEEE, 2018, pp. 174–181.

[106] David Defour. "FP-ANR: a representation format to handle floating-point cancellation at run-time". In: *25th symposium on computer arithmetic*. IEEE, 2018, pp. 76–83.

[107] Yohann Uguen, Florent de Dinechin, and Steven Derrien. "High-level synthesis and arithmetic optimizations". In: *Compas'2016- conférence d'informatique en parallélisme, architecture et système*. Poster. 2016.

[108] Yohann Uguen, Florent de Dinechin, and Steven Derrien. “A high-level synthesis approach optimizing accumulations in floating-point programs using custom formats and operators”. In: *International symposium on field-programmable custom computing machines*. IEEE, 2017, pp. 80–80.

[109] Yohann Uguen, Florent de Dinechin, and Steven Derrien. “Bridging high-level synthesis and application-specific arithmetic: the case study of floating-point summations”. In: *27th international conference on field programmable logic and applications*. IEEE, 2017, pp. 1–8.

[110] Yohann Uguen and Florent de Dinechin. “Exploration architecturale de l’accumulateur de kulisch”. In: *Compas’2017 - conférence d’informatique en parallélisme, architecture et système*. 2017, pp. 1–8.

[111] Yohann Uguen, Luc Forget, and Florent de Dinechin. “Evaluating the hardware cost of the posit number system”. In: *29th international conference on field-programmable logic and applications*. IEEE, 2019.

[112] Luc Forget, Yohann Uguen, and Florent De Dinechin. “Hardware cost evaluation of the posit number system”. In: *Compas’2019 - Conférence d’informatique en Parallélisme, Architecture et Système*. 2019, pp. 1–7.

[113] Luc Forget, Yohann Uguen, Florent de Dinechin, and David Thomas. “A type-safe arbitrary precision arithmetic portability layer for HLS tools”. In: *International symposium on highly efficient accelerators and reconfigurable technologies*. 2019, pp. 1–6.

[114] Yohann Uguen and Eric Petit. “PyGA: a Python to FPGA compiler prototype”. In: *International workshop on artificial intelligence and empirical methods for software engineering and parallel computing systems*. ACM, 2018, pp. 11–15.

FOLIO ADMINISTRATIF

THESE DE L'UNIVERSITE DE LYON OPEREE AU SEIN DE L'INSA LYON

NOM : Uguen

DATE de SOUTENANCE : 13/11/2019

Prénoms : Yohann, Yves

TITRE : High-level synthesis and arithmetic optimizations

NATURE : Doctorat

Numéro d'ordre : 2019LYSEI099

Ecole doctorale : Infomaths

Spécialité : Informatique

RESUME : À cause de la nature relativement jeune des outils de synthèse de haut-niveau (HLS), de nombreuses optimisations arithmétiques n'y sont pas encore implémentées. Cette thèse propose des optimisations arithmétiques se servant du contexte spécifique dans lequel les opérateurs sont instanciés, avec par exemple des divisions par des constantes.
Elle propose ensuite de s'éloigner de la sémantique des langages supportés par les outils de HLS, améliorant ainsi le compromis précision/coût/performance. Cette proposition est démontrée sur des sommes-de-produits de nombres flottants. La somme est réalisée dans un format en virgule-fixe défini par une directive de compilation.
Quand trop peu d'informations sont disponibles ce format en virgule-fixe, une stratégie est de générer un accumulateur couvrant l'intégralité du format flottant. Cette thèse explore plusieurs implémentations d'un tel accumulateur. L'utilisation d'une représentation en complément à deux permet de réduire le chemin critique de la boucle d'accumulation, ainsi que la quantité de ressources utilisées. L'intérêt d'un accumulateur exact est tel qu'il est proposé comme brique de base à un format alternatif aux nombres flottants, appelé posits, utilisant une précision variable. Pour évaluer précisément le coût matériel de ce format, cette thèse présente des architectures d'opérateurs posits, implémentés avec le même degré d'optimisation que celui de l'état de l'art des opérateurs flottants. Une analyse détaillée montre que le coût des opérateurs posits est malgré tout bien
plus élevé que celui de leurs équivalents flottants. Enfin, cette thèse présente une couche de compatibilité entre outils de HLS, permettant de viser plusieurs outils avec un seul code. Cette bibliothèque implémente un type d'entiers de taille variable, avec de plus une sémantique strictement typée, ainsi qu'un ensemble d'opérateurs ad-hoc optimisés.

MOTS-CLÉS : Synthèse de haut niveau, arithmétique des ordinateurs

Laboratoire (s) de recherche : CITILab

Directeur de thèse: Florent de Dinechin

Président de jury : Frédéric Pétrot

Composition du jury :
Frédéric Pétrot, Professeur des Universités, TIMA, Grenoble, France
Philippe Coussy, Professeur des Universités, UBS, Lorient, France
Olivier Sentieys, Professeur des Universités, Univ. Rennes, Inria, IRISA, Rennes, France
Laure Gonnord, Maître de conférence, Université Lyon 1, France
Martin Kumm, Professeur des Universités, Université de Fulda, Allemagne
Florent de Dinechin, Professeur des Universités, INSA Lyon, France