High-performance coarse operators for FPGA-based computing

Matei Valentin Istoan

N°d'ordre NNT : 2017LYSEI030

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
(INSA de Lyon, CITI lab)

Ecole Doctorale : InfoMaths EDA 512
(Informatique Mathématique)

Spécialité/ discipline de doctorat : Informatique

Soutenue publiquement le 06/04/2017, par :
Matei Valentin Iştoan

High-Performance Coarse Operators for FPGA-based Computing

Devant le jury composé de :

IENNE, Paolo Professeur des Universités, EPFL, Lausanne	Rapporteur
CHOTIN-AVOT, Roselyne Maître de Conférences HDR, UPMC	Rapporteur
THOMAS, David Senior Lecturer Imperial College London, London	Examinateur
PETROT, Frédéric Professeur des Universités, ENSIMAG, Saint-Martin d'Heres	Président
SENTIEYS, Olivier Professeur des Universités, ENSSAT, Lannion	Examinateur
DE DINECHIN, Florent Professeur des Universités INSA-Lyon	Directeur de thèse

Thesis advisor: Professor Florent Dupont de Dinechin Matei Iştoan

High-performance Coarse Operators for FPGA-based Computing

Abstract

Field-Programmable Gate Arrays (FPGAs) have been shown to sometimes outperform mainstream microprocessors. The circuit paradigm enables efficient application-specific parallel computations. FPGAs also enable arithmetic efficiency: a bit is only computed if it is useful to the final result. To achieve this, FPGA arithmetic shouldn't be limited to basic arithmetic operations offered by microprocessors.

This thesis studies the implementation of coarser operations on FPGAs, in three main directions:

New FPGA-specific approaches for evaluating the sine, cosine and the arctangent have been developed. Each function is tuned for its context and is as versatile and flexible as possible. Arithmetic efficiency requires error analysis and parameter tuning, and a fine understanding of the algorithms used.

Digital filters are an important family of coarse operators resembling elementary functions: they can be specified at a high level as a transfer function with constraints on the signal/noise ratio, and then be implemented as an arithmetic datapath based on additions and multiplications. The main result is a method which transforms a high-level specification into a filter in an automated way. The first step is building an efficient method for computing sums of products by constants. Based on this, FIR and IIR filter generators are constructed.

For arithmetic operators to achieve maximum performance, context-specific pipelining is required. Even if the designer's knowledge is of great help when building and pipelining an arithmetic datapath, this remains complex and error-prone. A user-directed, automated method for pipelining has been developed.

This thesis provides a generator of high-quality, ready-made operators for coarse computing cores, which brings FPGA-based computing a step closer to mainstream adoption. The cores are part of an open-ended generator, where functions are described as high-level objects such as mathematical expressions.

Contents

Listing of figures

Dedicată familiei mele.

Acknowledgments

As everyone knows, behind every great *thésard* there is a great *encadrant*. As such, I would like to extend my gratitude to my supervisor and computer arithmetic mentor *Florent de Dinechin*. Thank you for making this thesis possible. Thank you for taking the time to share this great amount of knowledge that I have accumulated through my thesis. Thank you for always suggesting new and challenging ideas, and most of all for guiding me towards finding a solution for all of these challenges. Thank you for always providing a environment where I was happy and looking forward to come and do research in. It was a great pleasure to collaborate all these years.

As everyone knows, behind every great *doctorand* there is a great *familie*. I would have never made so far without your help and your support. Therefore, thank you *mama*, *tata* and my brother *Paul*. Thank you for all your love. Thank you for always being there. Thank you for believing in me and helping me get one step forward. Thank you for being a role-model and offering me someone to look-up to. The security you have offered me all these years is something I will never stop being grateful for. Thank you Paul for always paving the ways that I've taken, and for showing me what's possible when one sets his mind on an objective. Thank you for making possible my first interaction with research. You are and always will be the best brother a younger brother could wish for.

As everyone knows, behind every great *doutorando* there is a great *namorada*. Thank you from all my heart *Mirella* for all the love you have given me all this time. Thank you for being there for me, in a way I couldn't have imagined possible. Thank you for being calm and kind and patient, when I needed it most. Thank you for being my best friend and my confident. The excitement of doing research is further amplified when you are looking forward to going home each day. For this, I have only you to be grateful for.

I would like to extend my gratitude to all my collaborators, *Thibault Hilaire* and *Anastasia Volkova*, *Nicolas Brisebarre* and *Silviu-Ioan Filip*, *Bogdan Pașca* for all the exciting scientific exchanges. Thank you for making parts of this thesis possible and for growing the reach of my understanding and for teaching me new and exciting things.

I am grateful to the members of the defense jury for accepting to take on such a task. I would like to thank *Roselyne Chotin-Avot* and *Paolo Ienne* for reviewing the thesis manuscript. Thank you for the very insightful feedback, the thorough remarks and invaluable corrections. I would like to thank *David Thomas*, *Olivier Sentieys* and *Frédéric Petrot* for accepting to be part of my defense jury, for their challenging remarks and the opportunity of having a new and different view on the thesis.

As everyone knows, behind every great *PhD. candidate* there is a great *group of friends*. Doing your PhD. is a country abroad is a amazing experience, but can have it's thorny moments as well. Therefore, I would like to thank all of *RoMafia* for welcoming in their midsts on my arrival in Lyon. Thank you for the endless discussions in the coffee breaks, for the movies, the picnics, the sport and the parties, for the great advice and the challenging views. Thank you *Bogdan* and *Mioara*, *Saşa* and *Andreea*, *Adrian*, *Cristi*, *Oana*, *Claudiu*, *Ioana C.*, *Ioana*, *Sorana*, *Adela*. And when RoMafia was no more, thank you to those that kept a Romanian touch in my French experience and graced me

with their friendships. Thank you *Silviu*, *Magda* and *Ana*, *Ruxandra*, *Mihai* and *Valentina*.

It is a great thing to collaborate with other researchers. This experience is even greater when these people go from being collaborators to being great friends. Thank you *Anastasia*, *Silviu* and *Bogdan* for making this possible.

Thank you *Thomas* and *Vincent* for making the Masters easier and keeping this friendship alive ever since.

Thank you *Quentin* and *Valentin* for making living in Lyon that bit more enjoyable. Thank you *Jacky* and *Nicolas* for being great friends ever since the early days in Lyon.

Thank you *Daniel* for all the Tuesday and Wednesday nights of football, the talks, the laughs and the comforting discussions on how PhD. life is complicated.

Thank you *Julien*, *Fosca*, *Natalia*, *David* and *Tatjana*, *Catherine* and *Osvanny*, *Quentin* and *Mari* for making sure that life outside the laboratory is just as much as enjoyable as the one inside. Thank you *Glenn* for being a great friend throughout, and an even better one since the defense.

Thank you *Mickael* and *Matthieu* for being the best office mates one could want, and great friends outside the laboratory. Thank you *Ştefan*, *Leo*, *Samir*, *Victor* and *David* for making life at the lab that much more interesting. Thank you *Gaelle*. And I am grateful to all my colleagues at CITI laboratory for making the PhD. an enjoyable experience.

I would also like to thank my great friends *Mihai*, *Paul* and *Mircea* for being there all this time. Thank you *Vlad*, *Claudiu*, *Zsolt* and *Robert* for you friendship.

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

1 Introduction

Context: Computer Arithmetic for Reconfigurable Circuits

Computations are at the heart of computer science. Either fast or slow, exact or approximate, and of various degrees of complexity, they are part of every computing or information processing system. As computer science developed and consolidated itself as a discipline, increasing interest developed around the efficiency of the computations. This is partly due to an effort to overcome the limitations of evolving hardware resources, and partly due to computer science making its way as a ubiquitous tool for other domains, where it was pushed to its limits. The common ground between the two is a wish to better take advantage of the available resources.

This underlying quest for better performance has driven computing systems to specialization. A graphics processor (GPU) can generally do a better job of graphics-related tasks than a generalized processor (CPU), while an application-specific circuit (ASIC) can do an even more limited set of tasks, but even more efficiently. In this setting, reconfigurable circuits try to strike a balance between efficiency and re-usability, an aspect which is sacrificed by application-specific circuits.

Some of these devices have proven to be even more versatile than originally designed. GPUs have proven their prowess in scientific and financial computations, to mention only a few domains. It is also the case for reconfigurable devices. Originally designed for testing and emulation, this class of devices has shown its versatility, in fields such as networking, signal processing, databases, neural networks, scientific computing, or for being used as accelerators in general. It is this latter use-case that has brought them into the attention of many scientific communities.

Technological advancements can no longer be a major driving force behind the advancement of

processing systems, as once anecdotally predicted by Moore's law [90]. The efficiency of these systems, which measures the amount of energy they require per processing power they provide, becomes even more important. As it turns out, this has also been predicted. Koomey's law [68], which predicts an doubling of efficiency every 1.57 years, can be confirmed from ENIAC up until current times. It is the efficiency of reconfigurable circuits, with their low energy requirements and flexibility, that has brought them an increasing amount of attention in recent times.

However, for reconfigurable circuits to have a chance of reaching mainstream status, or at least escape that of a niche, they must meet some simple, but essential criteria. First of all, they need good and reliable tools. Second, they need to be easy to use and program. It all starts at the level of computations. Reconfigurable circuits need reliable and efficient arithmetic computations.

Therefore, this thesis sets out to bring some contributions to the field of computer arithmetic for reconfigurable circuits, with the hope of improving the usability and accessibility of this class of devices, while providing a boost in their performance.

Objectives

The work presented throughout this thesis concentrates on Field-Programmable Gate Arrays (FPGAs), a class of reconfigurable devices, which consist of a large number of bit-level elementary computing elements, that can be configured and connected to implement arbitrary functions. Unlike CPUs, which use a sequential/parallel program as a programming model, FPGAs instead use circuits as a programming model, as they are programmable at the hardware level, and enables efficient, application-specific, parallel computations.

Parallel computational tasks can be laid out on the device, which means FPGAs are a good match for streaming architectures, where data is processed as it flows from one end to the other of the circuit. This dataflow efficiency can be complemented by the arithmetic efficiency on FPGAs. CPUs have fixed datapaths, usually of 32 or 64 bit, and have a poor arithmetic efficiency for most numerical applications, leading to wasted data bandwidth and increased power consumption. There is no such constraint in FPGAs. Therefore, in order to achieve arithmetic efficiency, FPGA arithmetic shouldn't be limited to basic arithmetic operations offered by microprocessors, nor should it imitate them. This can give birth to operators that are radically different.

This thesis studies the implementation of coarse arithmetic operators on FPGAs, with a focus on three main aspects: a generator for arithmetic cores, fine-grain operators that can constitute the basic building blocks and, finally, the coarse operators themselves.

For arithmetic operators to be able to achieve their maximum performance potential, application-specific and context-specific pipelining is required. This, however, is a tedious, long and error-prone process, usually of an iterative nature. Even if a designer's knowledge of the underlying algorithms

and of the target device are of great help, pipelining an arithmetic datapath remains a complex process, better left to automation.

Many techniques for developing arithmetic operators rely of the reuse of existing elementary ones. It is therefore crucial that these basic blocks offer the best possible performance and flexibility. In general, operator fusion offers the opportunity to perform optimizations on the developed operators, as one unique, global optimization. A framework for the development and automatic optimization of operators based on additions and multiplications is considered, as a possible backbone for the generator.

FPGA-specific approaches for computing arithmetic operators are required, targeting modern devices. Each of the methods should be tuned for its context, and be as versatile and flexible as possible. In order for them to be efficient, they require a fine understanding of the underlying algorithms, as well as a carefull error analysis. Overall these methods try to answer a fundamental question: what is the true cost of computing these arithmetic operators, in the context of contemporary reconfigurable circuits?

Digital filters are a very powerful tool for signal processing, and this family of coarse operators share many aspects in common with arithmetic functions. Digital filters too can be seen as mathematical objects, and can be specified at a high level of abstraction, as a transfer function with constraints on the signal/noise ratio. However, it is not obvious how to pass from this high-level specification to an implementation, while still satisfying the initial constraints. An efficient method which captures all these subtleties is needed.

This thesis aims to provide a generator of high-quality, ready-made operators for coarse computing cores, which brings FPGA-based computing a step closer to mainstream adoption. The cores are part of the open-ended and open-source generator FloPoCo, where functions are described as high-level objects such as mathematical expressions.

Outline of the Thesis

The rest of this thesis is organized as presented in the following. In Chapter 2, many of the basic notions required in this thesis are briefly introduced. Section 2.1 presents an overview of the target hardware platform, the FPGAs, with a focus on the features that are relevant in the following chapters. The introduction is continued with notions of computer arithmetic, in Section 2.2. Finally, Section 2.3 presents the FloPoCo project, a generator of arithmetic operators, which is also the setting for the majority of the work resulting out of this thesis.

Part I focuses on the generator of arithmetic cores, and, more specifically, on its back-end. The focus is on how to improve an operator's frequency through pipelining, as well as how to improve the typical design flow for a new arithmetic operator. Chapter 3 introduces the basics for creating

an operator's datapath with the help of an example, a Multiply-Accumulate (MAC) unit. It also gives an overview of the current state of the art solutions for automatically inferring pipelines inside arithmetic operator generators. Chapter 4 introduces FloPoCo's new automatic pipeline generation framework. And, finally, Chapter 5 showcases another automatic feature of the arithmetic operator generator, the hardware resource estimation.

Part II shifts focus from the generator's back-end towards arithmetic operators. The first subject to be treated, in Chapter 6, is the bitheap framework, used as a basic building block throughout the following chapters. The following two chapters, 7 and 8, provide examples of the opportunities offered by the bitheap framework, as well as the new challenges that it creates.

Part III is dedicated to a class of coarse-grain operators, the elementary functions. It showcases the possible ways of taking advantage of the features introduced in the two previous parts for creating state of the art implementations for new arithmetic operators. Chapters 11 and 9 deal with univariate functions, while Chapter 10 with multivariate functions. Chapter 11 presents some algebraic functions: the reciprocal, the square root and the reciprocal square root. Chapter 9 studies some elementary functions: the sine and cosine trigonometric functions. These are followed by another trigonometric function in Chapter 10, the two-input arctangent (commonly referred to as $\mathrm{atan}\,2$), which is a use-case for the study of multivariate functions.

Part IV tackles coarse operators for signal processing. It investigates a methodology for creating digital filters, from specification to hardware implementation, with the minimal designer intervention. A building block for the development of digital filters, the sum of products by constants, is introduced in Chapter 12. Most of the filter architectures presented in the following chapters can be based on it. Chapter 13 discusses finite impulse response (FIR) filters, with a focus on automating and facilitating the design process. Finally, Chapter 14 studies infinite impulse response (IIR) filters.

Some concluding remarks, as well as an outlook on the future opportunities offered by the work described in this thesis are given in Chapter 15.

2

Arithmetic on FPGAs

FPGAs: Reconfigurable Programmable Devices

Field Programmable Gate Arrays (FPGAs) are the most popular type of reconfigurable circuits. Introduced by Xilinx in 1985, the XC2064 [1] was the first device of this type [2], and can trace its origins back to Programmable Logic Devices (PLA) or Programmable Logic Arrays (PLD). However, unlike their precursors, they were designed to have a much finer granularity. Traditionally, FPGAs have been implemented using CMOS technology, allowing them to follow the advances of this technology. In its most basic form, the FPGA has a sea-of-gates architecture, consisting of a large number of programmable logic elements, interconnected using a configurable network. The logic elements are small programmable memories, whose contents, or implemented function, can be reconfigured. The switching matrix can be reconfigured as well, which means that a FPGA can implement any function that is sufficiently small so as to fit inside the device. As a reference, Figure 2.1 presents the architecture and an illustration of the die of the Xilinx XC2064. It consisted of a mere 8×8 array of logic elements, complemented by programmable routing and input/output blocks. The device had a maximum operating frequency of around 100 MHz, and were implemented using a fabrication process on $2.0\mu m$.

For comparison purposes, Figure 2.2 illustrates a device from Xilinx's newest and most powerful family of devices, Virtex 7. Contemporary devices have moved on from the homogeneous logic array architecture towards a heterogeneous one, containing embedded multiplier blocks, memory blocks, transceivers for high-speed input/output operations, and even full CPU cores. In Figure 2.2, these resources are grouped into columns, to form the so called ASMBL architecture, which can have

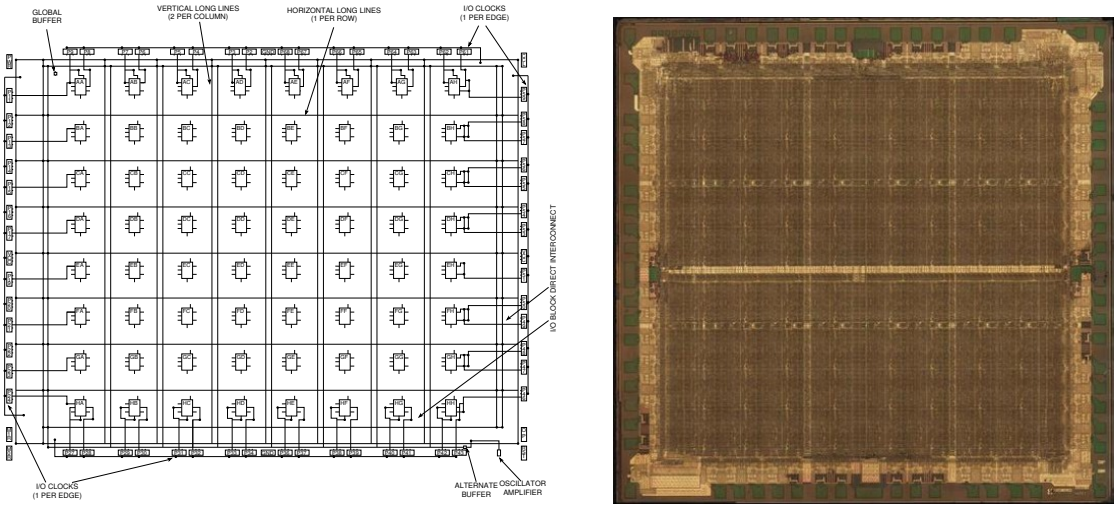


Figure 2.1: Architecture and die of the Xilinx XC2064 chip. Image extracted from [1].

different variations depending on the application domain. Contemporary devices contain between 300,000 and 1,200,000 logic elements, and between 1,000 and 2,000 multiplier and memory blocks, respectively. The devices have a maximum operating frequency of around 1800 MHz for the logic elements, 700 MHz for the multiplier blocks and 600 MHz for the memory blocks, and are implemented on a 28*nm* fabrication process.

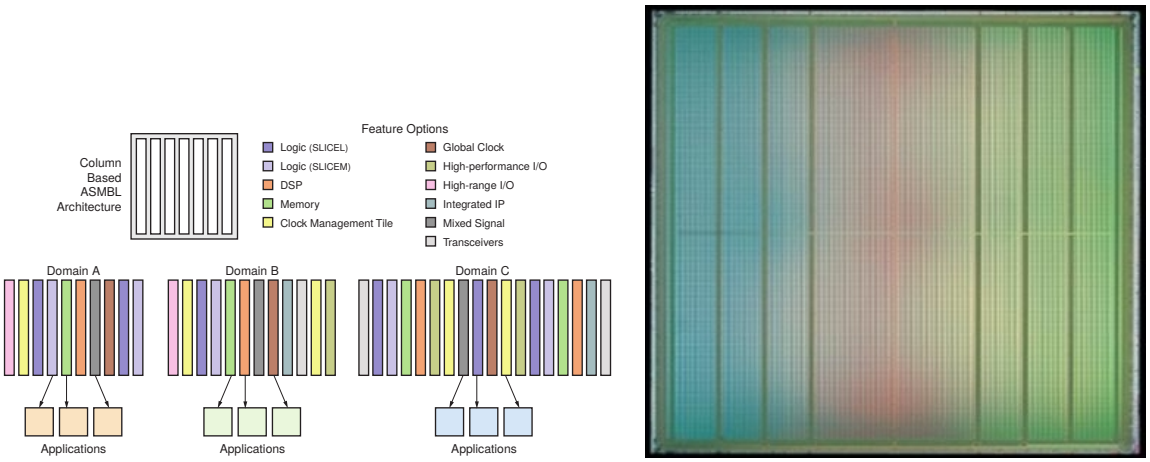


Figure 2.2: Architecture and die of a Xilinx Virtex 7 chip. Image extracted from [6].

However, FPGAs pay a price in performance, as compared to their fixed-function equivalents, due to their reconfigurability. This can have a negative impact factor of about 5-25 times on the area,

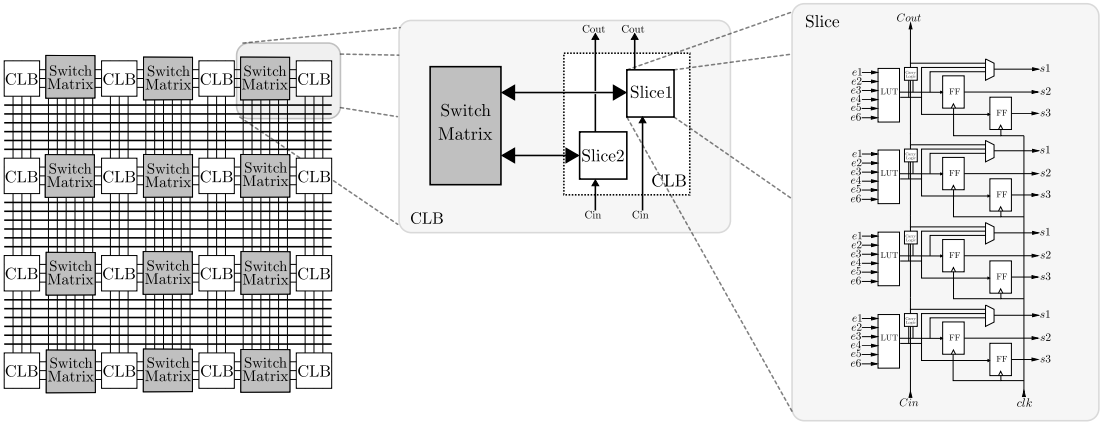


Figure 2.3: Organization of logic resources on Xilinx devices.

delay and performance [52], as compared to a CPU. This being said, they still remain extremely efficient and well suited for processing large streams of data. They also make up for their shortcomings by having a much lower cost for a new design, as well as a much lower development time (as compared to an ASIC, for example).

As contemporary FPGA devices are evolving, the number of features they propose are becoming increasingly complex, and are also increasing in number. In the following, some of the most used architectural features of FPGAs are reviewed, with a focus on those that are relevant throughout the thesis. Also, the discussions are limited to devices proposed by the two main FPGA device manufacturers, Xilinx and Intel, which dominate the market with a combined share of approximately 90%.

Logic elements

At the heart of the FPGA is the logic element, which in most cases is a programmable memory, that is used as a look-up table (LUT) from which a function of the input(s) can be read at the output(s) (as shown, for example, for the Xilinx devices in Figure 2.4). However, with the increasing number of logic elements available on the devices, an arrangement as the one in Figure 2.1 quickly becomes impractical and a bottleneck of the performance. Therefore, contemporary FPGAs have a possibly nested hierarchical structure, which groups together a number of logic elements, as well as with some optional resources, such as local routing, multiplexers, adders or storage elements.

Xilinx devices group logic elements into Configurable Logic Blocks (CLBs) [6]. A simplified view of its structure is presented in Figure 2.3. In turn, every CLB is made out of two slices, which cannot communicate directly between each other, but can communicate through a carry chain to

the slices that are directly below and above them. Finally, a slice is composed out of 4 LUTs, two additional levels of multiplexers, dedicated carry logic to speed-up additions and 8 flip-flops/registers (FF./reg.), two for each LUT.

There are two types of slices, SLICEL and SLICEM, where SLICEM has some extra functionality that allows it to be used as a distributed RAM (256 bits), or as a shift register (32-bit shift). A zoom on a simplified view of a slice is shown in Figure 2.4.

The logic elements themselves are 6-input look-up tables. They can be used in a variety of modes, being cable of implementing any function of up to 6 independent inputs, or two 5 independent input functions, as each LUT has two outputs. Using the two layers of multiplexers, the LUTs in a slice can be combined to implement 7- or 8-input functions, without paying the cost of the increased routing delay outside of the slice. The multiplexers inside a slice can also be used for the efficient implementation of wide multiplexers, up to a size of 16:1, in one slice.

When a SLICEM is used as distributed memory, it can implement either RAM or ROM. As a RAM, it can be used as $32/64 \times 1$ single-port memory, using a single LUT, or $32/64 \times 1$ dual-port memory using two LUTs, or in quad-port configuration, by doubling again the number of LUTS used, 128×1 single/dual port using two/four LUTs, or 256×1 single port. As a ROM, a SLICEM ca be used as a $64/128/256 \times 1$ memory, using one, two and respectively four LUTs.

The logic elements can also be used for implementing additions/subtractions, taking also advantage of the dedicated fast lookahead carry logic. This runs across an entire slice, and is disposed vertically in the device. Longer carry chains can be formed, as the carry propagation continues to the next slice, avoiding the long delays of the general routing network. This carry logic can also be manipulated, useful for implementing specific functions, in conjunction with the LUTs.

Finally, in each slice there are two flip-flops, that can store the output of each of the LUTs. When left unused by the corresponding LUT, a FF can store some data that comes from a different source, as long as the routing resources' configuration allows it.

INTEL devices group the logic resources in a slightly different manner, illustrated in Figure 2.5 and 2.6. At the top of the hierarchy is the Logic Array Block (LAB) [9], which is itself composed out of Adaptive Logic Modules (ALMs). There are two types of LABs, the second being the MLAB, that can be used as memory and that represent approximately a quarter of the LABs. A LAB contains 10 ALMs, contain a local interconnect allowing ALMs to communicate between themselves, connect to the adjacent LABs using a direct link and also connect to the local and global interconnect networks. At the end of the hierarchy are the ALMs, which contain an Adaptive LUT (made out of four 4-input LUTs), a 2-bit adder (with the corresponding carry chain) and 4 registers.

An ALM, illustrated in Figure 2.6, has three functioning modes: normal, extended and arithmetic. In normal mode, it can implement a 6 independent input function, or two 4 independent input

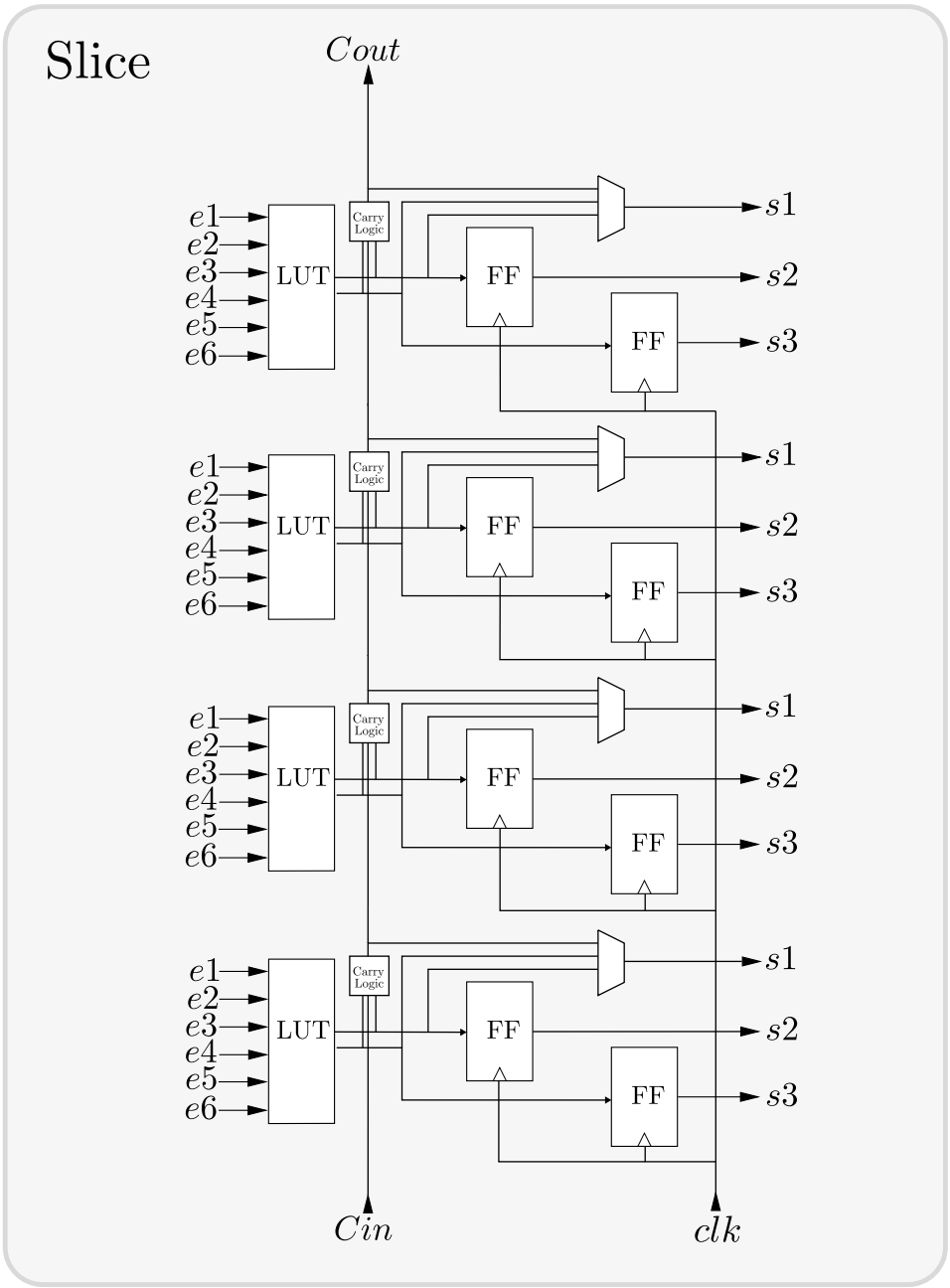


Figure 2.4: Overview of a SLICE on Xilinx devices.

functions, or two 5-input functions with two of the inputs shared, or a 5-input and a 3-input function, or a 5-input and a 4-input function with a shared input. Other combinations of two functions

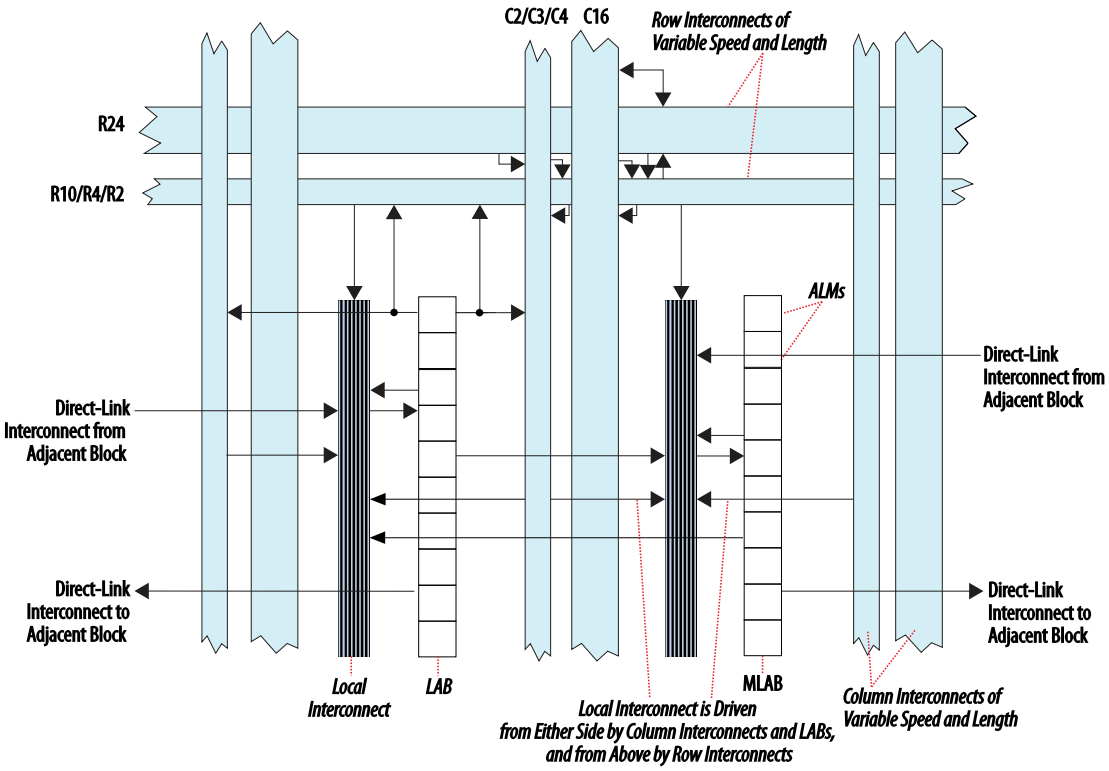


Figure 2.5: Organization of logic resources on Intel devices. Image extracted from [9].

of fewer inputs are also possible. In extended mode, certain functions of up to 8 inputs can be implemented. In arithmetic mode, the adders in the ALM can each implement the sum of two 4-input functions. The carry chain spans the entire length of a LAB, which means that adders of a length of at most 20 bits can be implemented inside a LAB.

A useful feature worth mentioning is the support of ternary addition (the variant of addition involving three addends) on Intel devices. This can be very easily implemented, and does not require special low-level manipulations from the designer, as it is directly supported in the logic fabric. On Xilinx devices, this is not directly supported and requires some low-level manipulations in order to be achieved. Ternary adders can be implemented at almost the same cost as the regular ones, but have a much higher latency.

The ALMs inside an MLAB can be used as RAM, each being capable of supporting a 32×2 bit configuration. Therefore, a whole MLAB can be configured as a 32×20 bit RAM, consisting of a total of 640 bits of memory.

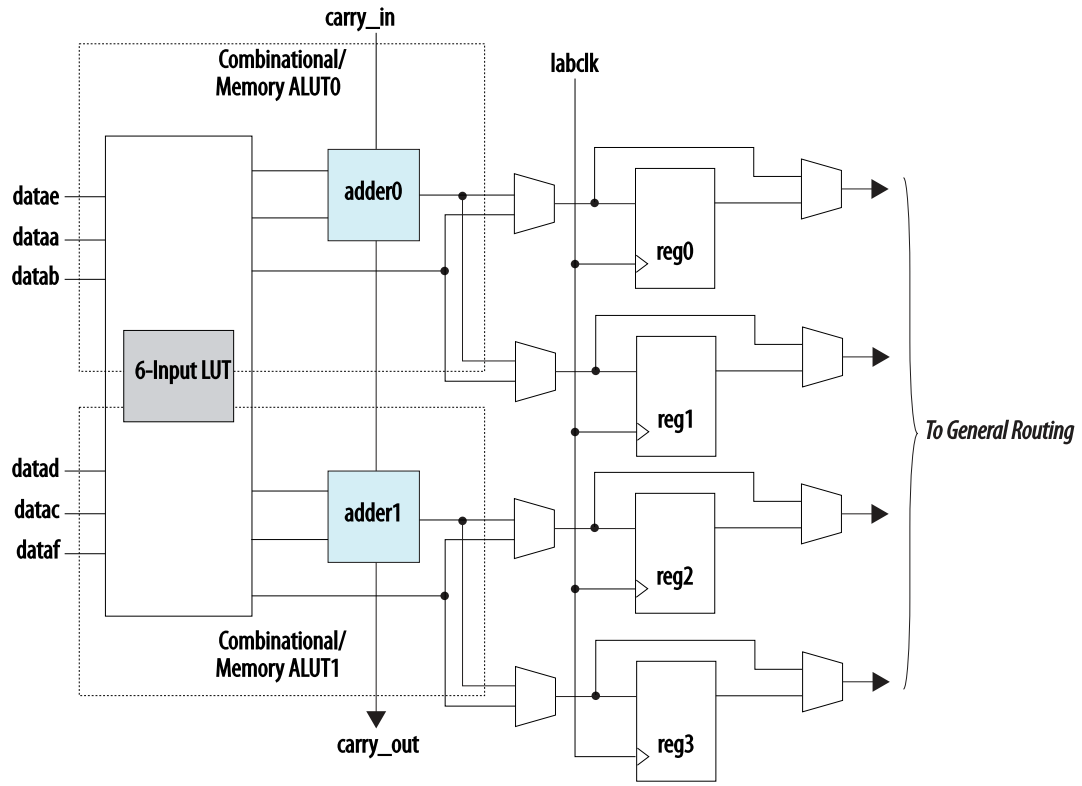


Figure 2.6: Overview of an ALM on Intel devices. Image extracted from [9].

Multiplier blocks

FPGAs also contain dedicated hard blocks for accelerating multiplications. Initially, these were just 18×18 bit multiplications, but have since evolved into more complex structures, commonly referred to as DSPs, due to their usefulness in signal processing.

Xilinx devices denote their DSP blocks as DSP48E1 [5], which is presented in Figure 2.7. They allow for a two's complement 25×18 bit signed/unsigned multiplication, followed by a 48 bit accumulation. The final addition is actually part of a 48 bit logic unit, and can thus also be used for performing a logic operation on two inputs (AND, OR, NOT, NAND, NOR, XOR, and XNOR). There is also a 25 bit pre-adder, that can serve the sum of two of the inputs as an input to the multiplier.

DSP slices can be cascaded together, to implement larger multiplications. In this mode, one of the operands is shifted by a constant amount of 17 prior to the accumulation.

The DSP slices also include pipeline registers. They are placed at the inputs, after the pre-adder

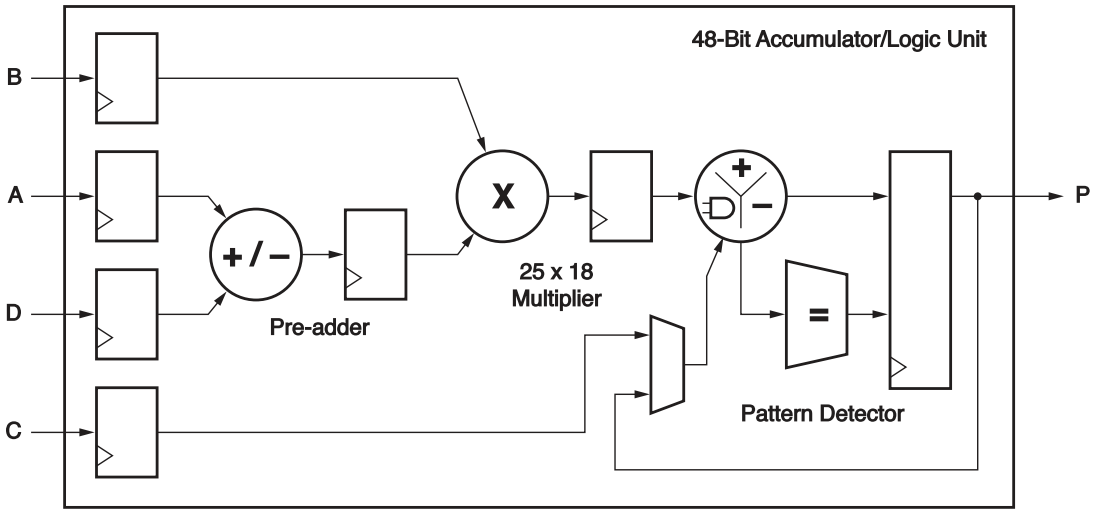


Figure 2.7: Overview of a DSP slice on Xilinx devices. Image extracted from [5].

and after the multiplication. A fortunate side-effect is that this can save registers in a design, using the DSP's bock internal registers instead of those in the logic fabric.

Intel devices denote their DSP blocks as Variable Precision DSP Blocks [10], a schematic overview of which is presented in Figure 2.8. Intel DSP blocks are coarser than the Xilinx ones. They can implement two 18×19 bit signed multiplications, or one 27×27 bit multiplications. Previous generations of devices had even more versatile blocks, being capable of implementing three 9×9 bit multiplications, two 18×18 bit signed multiplications, two 16×16 bit multiplications, or one 27×27 bit multiplication, or a 36×36 bit multiplication using two blocks.

It is worth mentioning that the series 10 of devices introduces direct support for floating-point operations in their DSP blocks. A DSP can be used for implementing one single-precision addition or multiplication, and several blocks can be used in conjunction to implement double-precision operations.

The DSP blocks on Intel devices also feature pre- and post-adders, and multiple levels of registers that ensure an elevated running frequency.

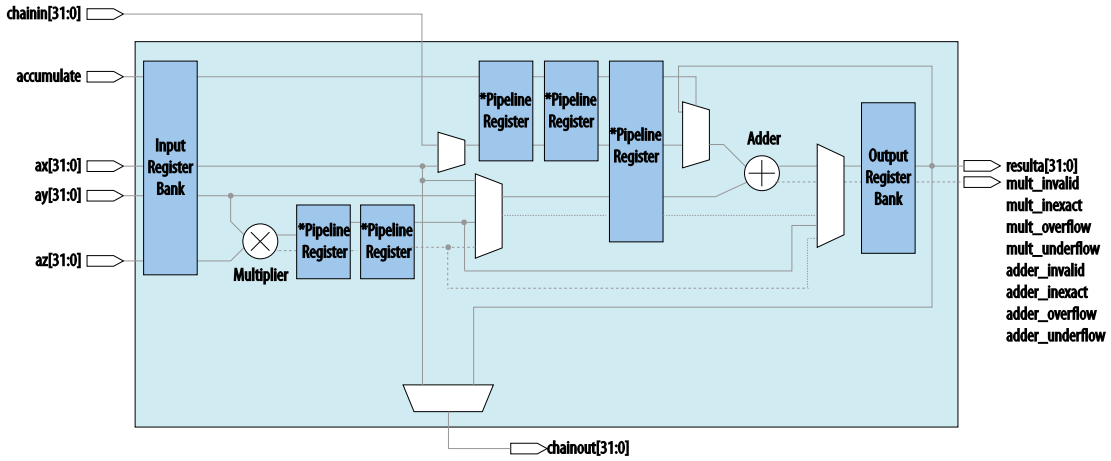


Figure 2.8: Overview of a Variable Precision DSP Block on Intel devices. Image extracted from [10].

Memory blocks

There are occasions when designers have storage needs that are past the possibilities offered by the logic elements, at least from a practical point of view. For those cases, FPGAs offer large memory blocks, often referred to as block RAM (BRAM). They have the advantage of being of a larger capacity, as compared to the memories in the logic elements, being faster and having a more predictable behavior.

These memories can be used in single, or dual port modes, and can be configured as RAM/ROM/-FIFO. They also have the necessary resources in a bock so as to allow the creation of larger memories using adjacent or neighboring blocks.

Xilinx offers blocks of size 36Kb, which can be configured as two independent 18Kb blocks [7]. They can be addressed in one of the following modes: $32K \times 1$, $16K \times 2$, $8K \times 4$, $4K \times 9$, $2K \times 18$, $1K \times 36$ or 512×72. The 18kb block can be configured as: $16K \times 1$, $4K \times 4$, $2K \times 9$, $1K \times 18$ or 512×36. Older devices only have 18Kb blocks.

Intel devices have memory blocks of size 20Kb [8]. They can be addressed as $2K \times 10$, $1K \times 20$, or 512×40. Older devices, or lower cost families have 10Kb memory blocks, with configurations similar to the ones of their 20Kb equivalents.

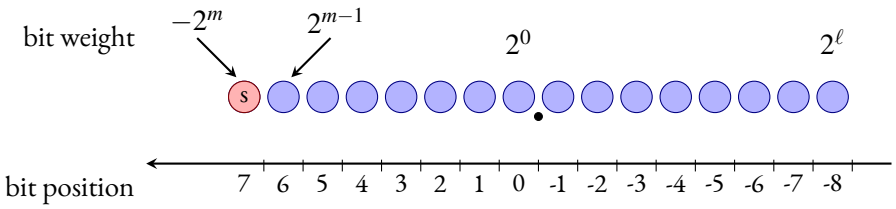


Figure 2.9: The bits of a fixed-point format, here $(m, \ell) = (7, -8)$.

Arithmetic for FPGAs

In the following, some notions related to computer arithmetic, that are used throughout the next chapters, are presented.

Fixed-point Formats

There are many standards for representing fixed-point data. The one chosen for this work is inspired by the VHDL `sfixed` standard.

A signed fixed-point format is described by sfix(m, ℓ) and an unsigned fixed-point format by ufix(m, ℓ). As illustrated in Figure 2.9, a fixed-point format is specified by two integers (m, ℓ) that respectively denote the weight (or the position) of the most significant bit most significant bit (MSB) and least significant bit least significant bit (LSB) of the data. Either m or ℓ can be negative, if the format includes fractional bits. The value of the bit at position i is always 2^i, except for the sign bit at position m, whose weight is -2^m, corresponding to two's complement notation. The LSB position ℓ denotes the *precision* of the format. The MSB position denotes its *range*. With both the MSB and the LSB included, the size of a fixed-point number in (m, ℓ) is $m - \ell + 1$. For instance, for a signed fixed-point format representing numbers in $(-1, 1)$ on 16 bits, there is one sign bit to the left of the point and 15 bits to the right, so $(m, \ell) = (0, -15)$.

Sign-extensions

The sign extension to a larger format of a signed number `sxxxx` (integer or fixed-point), where `s` is the sign bit, can (alternatively) be performed using a classical technique [42] as:

$$\begin{array}{rl} & \texttt{00...0}\overline{\texttt{s}}\texttt{xxxxxxx} \\ + & \texttt{11...110000000} \\ = & \texttt{ss...ssxxxxxxx} \end{array} \tag{2.1}$$

where $\overline{\mathtt{s}}$ is the complement of $\mathtt{s}$. Equation 2.2 can be verified for the two cases, $\mathtt{s} = 0$ and $\mathtt{s} = 1$. This transformation has the advantage of replacing the extension of a variable bit with an extension with constant bits. The usefulness of this technique will become apparent in later chapters, namely 6, 12 etc..

Rounding Errors, and Accuracy

In general, tilded letters are used in the following to denote approximate or rounded terms, for example $\widetilde{y}$ will represent an approximation of y.

The following conventions are used:

- ε for an absolute error,
- $\overline{\varepsilon}$ for the error bound, *i.e.* the maximum of the absolute value of ε;
- $\circ_\ell(x)$ the rounding of a real x to the nearest fixed-point number of precision ℓ.

The rounding $\circ_\ell(x)$, in the worst case, entails an error $|\circ_\ell(x) - x| < 2^{-\ell-1}$. For instance, rounding a real to the nearest integer ($\ell = 0$) may entail an error up to $0.5 = 2^{-1}$. This is a limitation of the format itself. Therefore, when implementing a datapath with a precision-ℓ output, the best thing that can be achieved is a *perfectly rounded* (or *correctly rounded*) computation with an error bound $\overline{\varepsilon} = 2^{-\ell-1}$.

A slightly relaxed constraint can be formulated as: $\overline{\varepsilon} < 2^{-\ell}$. This is called last-bit accuracy, because the error must be smaller than the value of the last (LSB) bit of the result. It is also referred to as *faithful rounding* in the literature. If the exact number happens to be a representable precision-ℓ number, then a last-bit accurate architecture will return exactly this value. This is still a tight specification, considering that the output format implies that $\overline{\varepsilon} \geq 2^{-\ell-1}$.

In terms of hardware, truncation is easy. Rounding to a certain lower precision can be achieved using the identity $\circ(x) = \lfloor x + \frac{1}{2} \rfloor$. Rounding to precision $2^{-\ell}$ is obtained by first adding $2^{-\ell-1}$ then discarding bits with weights lower than $2^{-\ell}$. In the worst case, this entails an error $\varepsilon_{\text{round}}$ of at most $2^{-\ell-1}$. Rounding to the nearest thus costs an addition. However, it is often possible to merge the bit at weight $2^{-\ell-1}$ in an existing addition.

The FloPoCo Project

The FloPoCo project [37] is an automatic generator of arithmetic cores for application-specific computing. One of the main premises of the FloPoCo project is that **the arithmetic operators** for FPGAs should be:

- **specific to the application** both qualitatively (for instance if an application can benefit from an operator for cube root, then this operator should be designed) and quantitatively (*e.g.* input/output formats should match application needs)

- **specific to the target platform** *i.e.* optimized for it

They should not be limited to those designed for mainstream CPU-based computing [33]. This should allow the designed operators to extract the most from what the hardware platform has to offer. A consequence of this premise is that **the arithmetic operators should compute just right and be last-bit accurate** .

Computing Just Right – Last-bit Accuracy

The rationale for last-bit accuracy architectures in the context of hardware is the following. On the one hand, last-bit accuracy is the best that the output format allows at an acceptable hardware cost. On the other hand, any computation less than last-bit accurate is outputting meaningless bits in the least significant positions, and therefore wasting hardware. When designing an architecture, this should clearly be avoided: each output bit has a cost, not only in routing, but also in power and resources downstream, in whatever will process this bit. This price should be paid only for the bits that hold useful information. Operators should be last-bit accurate. From another point of view, specifying the output format is enough to specify the accuracy of an operator.

Overview of the Project

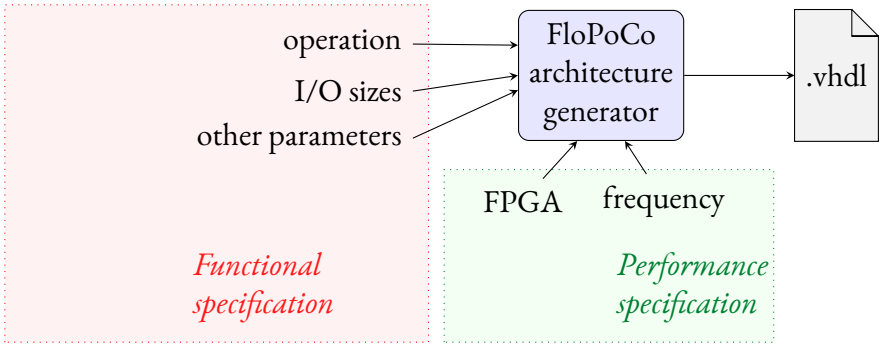


Figure 2.10: Interface to FloPoCo operators

A common drive behind the FloPoCo project was the ability to customize every arithmetic operator based on several parameters, such as:

- datapath sizes

- operator-specific parameters (*e.g.* a constant for a multiplication by a constant)
- implemented algorithm
- target hardware platform
- target frequency

To address this immense number of the possible combinations, even for a single operator, a choice was made in FloPoCo to compute the architecture out of the parameters. It also makes a large codebase easier to maintain. Therefore, using FloPoCo designers create parameterized operator generators, as shown in Fig. 2.10. The FloPoCo framework is written in C++ , and new operator generators can be specified in a mix of C++ and VHDL .

Wrapping VHDL code into C++ code eases the design process and offers a set of conveniences:

- eliminating some of the boilerplate inherent to VHDL
- generating easily-readable, parameter-free code
- keeping track of signals
- automating most of the process of circuit pipelining
- automating the testing process
- giving estimations of required resources and assisting with floorplanning

The following sections provide more details.

FloPoCo is also an extensive operator library which is already developed in the generator, and can be complemented by the designer's own operators. A comprehensive list of the available operators can be found on the project's homepage [*]. The class hierarchy is also briefly presented in Figure 2.11.

From a designer's perspective, there are two main points of interest in FloPoCo: *operators* and *targets*. Most of the code of the FloPoCo framework is dedicated to operators, while the devices on which these operators can run on are abstracted as targets [37].

Operators (such as `IntAdder`, `IntMultiplier`, `FixFIR` etc.) model the flow of information in hardware datapaths, while targets (such as `Virtex6`, `Kintex7`, `StratixV` etc.) model the underlying hardware. The two work in conjunction. Thus, FloPoCo offers an opportunity in terms of the design process: **the use of target-specific parameters during datapath creation** . These parameters cover both the available hardware and the timing inside the datapath. Using this information, it is possible to create operators that are optimized for a given target and a given context.

[*] `http://flopoco.gforge.inria.fr`

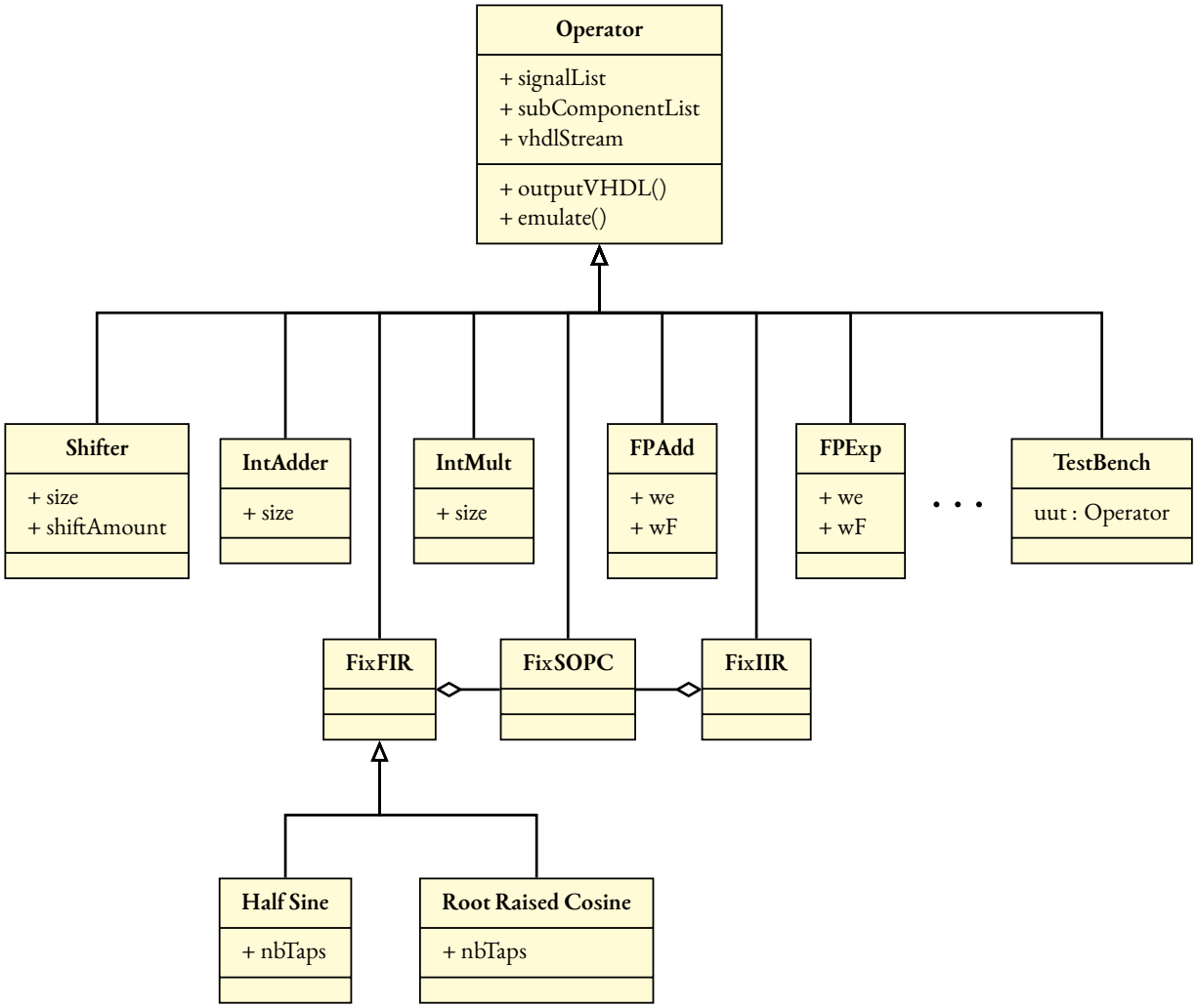


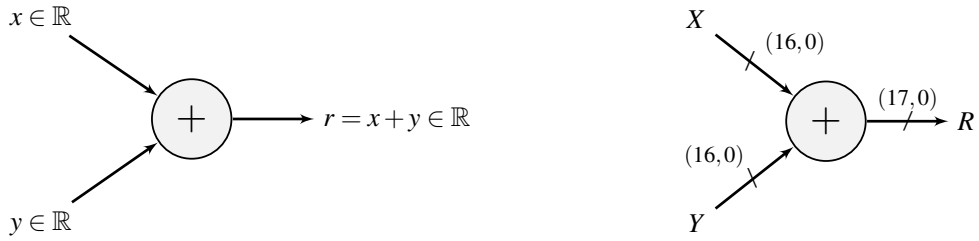
Figure 2.11: Partial Class diagram for the FloPoCo framework. Accent is put on the operators of interest to the work presented in this thesis.

The rest of Section 2.3 deals with these two building blocks of the FloPoCo framework. Section 2.3.3 presents in more details the process of arithmetic datapath design, illustrated by constructing a toy operator. Section 2.3.4 presents concepts related to targets and how they influence a operator's design.

Operators

FloPoCo arithmetic operators are hardware implementations of mathematical functions. They range from very basic ones like $+$ and $\times$, to trigonometric functions (sin, cos, atan), elementary functions (log, exp), to more complex ones like filters and polynomial approximations. From an architecture

point of view, they can be constructed as stateless (memoryless), combinatorial circuits[†]. From an arithmetic point of view, on the other hand, the operators are (in most cases) numerical approximations of the actual mathematical functions. Figure 2.12 shows an example of a mathematical function and its corresponding FloPoCo operator. The conceptual differences between the two (such as the domain for the inputs/outputs) are visible in the figure. Figure 2.13 shows another example, where the output of the operator is an approximation of the true mathematical result, on the target output precision.

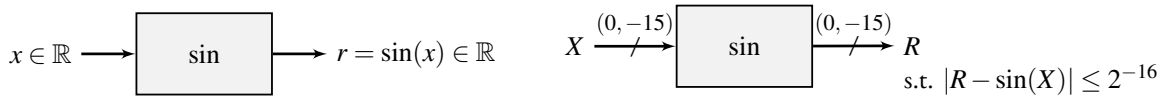


(a) A mathematical function

(b) An operator with fixed point inputs and outputs

Figure 2.12: Example of a mathematical function and its corresponding FloPoCo operator

Figure 2.10 shows that the **functional specification** and the **performance specification** of a FloPoCo operator are separated. This is a design paradigm that has existed in FloPoCo since its early versions [37].



(a) A mathematical function

(b) An operator

Figure 2.13: Example of a FloPoCo operator approximating a mathematical function

An operator's initial design phase is usually only concerned with the implementation of the functional specification. The implementation of the performance specification is left to further design iterations. These modifications should not impact the functionality of the operator, which might have already been tested and certified by this point. The main optimization is the pipelining of the datapath. This topic is introduced with the help of an example in Chapter 3 and discussed in more detail in Chapter 4, after a presentation of the state of the art in Section 3.3.

[†]this constraint is relaxed in the case of the FIR and IIR filters

Targets

The `Target` class tries to model the target hardware platform, in terms of timing and architectural resources. Examples of how to integrate targets in the design of an operator's datapath are provided in the listings of Chapter 3. The class diagram of the `Target` class is shown in Figure 2.14. Some of the details related to the hierarchy of the `Operator` class have been left out, for the sake of the clarity.

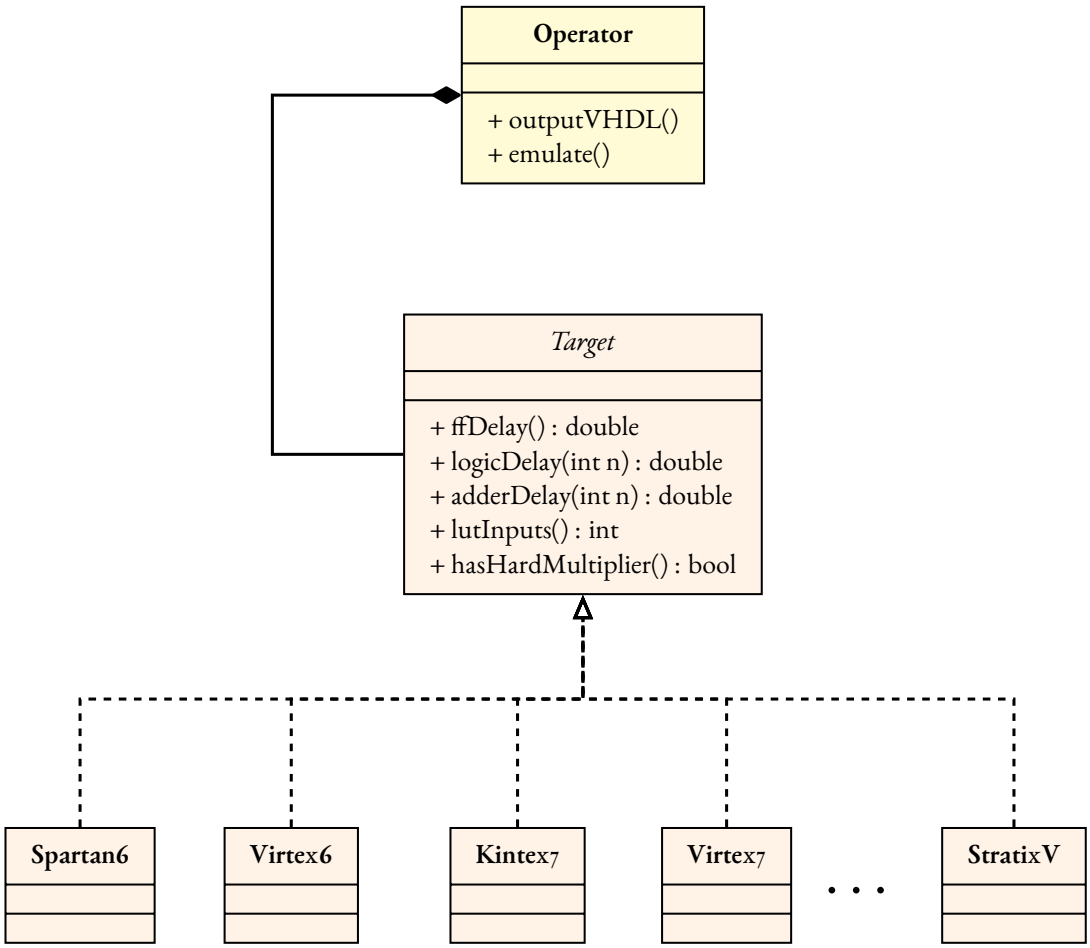


Figure 2.14: Class diagram for the FloPoCo framework with focus on the Target hierarchy

The `Target` class provides information on the delays of the various elements of the datapath of a FloPoCo operator. For example, a call to `DSPMultiplierDelay()` method the the `Target` class provides the delay of a hard multiplier block. The granularity of the information provided by the `Target` can be very fine, such as the delay of a look-up table (LUT), or of a wire, or of a multiplier block as in the previous example. The granularity can also be coarser. For example, the call to the `adderDelay(w)` method returns the delay of an adder of length `w`.

On the architecture side, the `Target` class abstracts details about the target device, that the designer can use inside its operators. For example, the `lutInputs()` method returns the number of inputs to the look-up tables (LUTs) on the target platform. The presence or the absence of certain capabilities in the hardware is also modeled in the `Target` class' methods and attributes.

The `Target` class can also abstract timing information on the various parameters of the target platform, for use in the design of a datapath. Such methods exist for most of the elements that can be found in a target device. Thus, the `Target` class allows the designer to create generic datapaths, which produce operators tailored specifically to the target device and the target constraints.

As pointed out in [37], constructing models for target devices is a perpetual effort. On the one hand, there are always new devices to model and add to the supported library. On the other, the timing information used for emulating the target devices is obtained from the evolving vendor-provided tools. This means that:

- the `Target` models are inherently approximate
- the `Target` models are constantly evolving

The structure and design of the `Target` class is in accordance to the paradigm used by the FPGA design suites. The class spans a large range of granularity for the level of control it provides to the designer. Experienced users can model their datapaths very precisely. This is in sync with the tools, which provide the necessary feedback so as to extract the information required to model the `Target` classes. However, there currently is a shift in the paradigm used by most of the vendor design suites. It is becoming increasingly difficult to predict the behavior and possible optimizations done by the synthesis/placement tools. In accordance, the evolution of the `Target` class, and the new design paradigm which it entails, is further discussed in Chapter 4.

Part I

Automatic Pipeline Generation for Arithmetic Operators

Part I introduces the typical **design flow for a FloPoCo operator** , as well as how the operator's frequency can be improved through pipelining. Chapter 3 presents the basics of creating an operator's datapath with the use of an example (a `Multiply-Accumulate unit`). It also gives an overview of the current state of the art solutions for inferring pipelines inside arithmetic operator generators. Chapter 4 deals with FloPoCo's new automatic pipeline generation framework. Finally, Chapter 5 discusses another new automatic feature of FloPoCo, the hardware resource consumption estimation.

The biggest difference between time and space
is that you can't reuse time.

Merrick Furst

3

Operators and Pipelines

Lexicographic Time

This Section introduces a few notions related to timing and the system used by the pipelining framework for ordering signals inside a datapath.

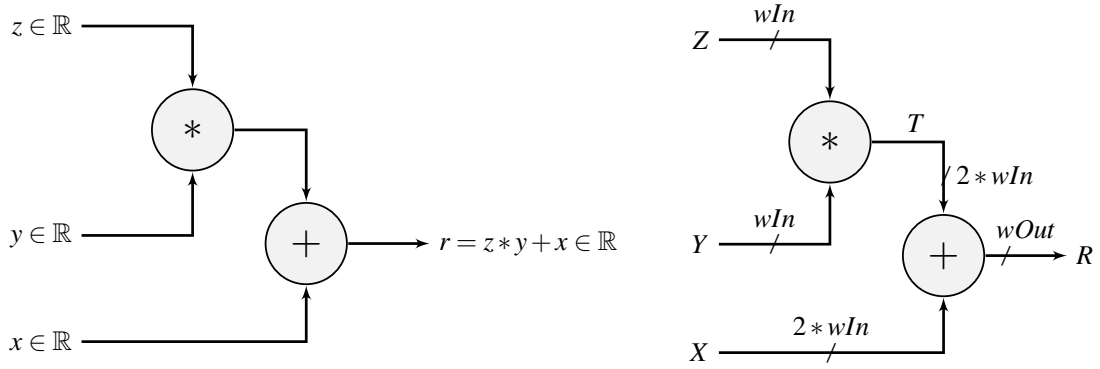
Inside a combinatorial circuit, the timing of signals can be determined by accumulating the delays on simple paths to inputs. The timing of a signal is given by the longest (in terms of accumulated delay) such path.

Inside a pipelined circuit, however, the timing of a signal is usually given in a lexicographic system by a pair (c, τ). The cycle c is an integer that counts the number of registers on the longest path from the input with the earliest timing to the respective signal. The critical path τ is a real number that represents the delay since the last register.

Signals can be ordered in a lexicographic order using the following relationship. A signal s_1, with timing (c_1, τ_1), is before a signal s_2, with timing (c_2, τ_2), if

$$(c_1, \tau_1) < (c_2, \tau_2) \quad \Longleftrightarrow \quad \begin{cases} c_1 < c_2 \\ \text{or} \\ c_1 = c_2 \quad \text{and} \quad \tau_1 < \tau_2 \end{cases} \tag{3.1}$$

Another important notion required for the scheduling algorithm is that of lexicographic time addition. For a target frequency f, the corresponding cycle latency is $1/f$. The addition between a



(a) The mathematical function

(b) The operator

Figure 3.1: Example of a multiply-accumulate unit

pair (c, τ) and a time delay δ is defined as:

$$(c\ ,\ \tau) \quad + \quad \delta \qquad = \qquad \left(c + \left\lfloor \frac{\tau+\delta}{\delta_{obj}} \right\rfloor,\quad \frac{\tau+\delta}{\delta_{obj}} - \left\lfloor \frac{\tau+\delta}{\delta_{obj}} \right\rfloor\right) \tag{3.2}$$

In Equation 3.2, δ_{obj} is defined as $\delta_{obj} = 1/f - \delta_{ff}$, where δ_{ff} is the time delay of a register and f is the user-specified target frequency, with the corresponding cycle latency of $1/f$.

The addition between a lexicographic pair (c_1, τ_1) and another one (c_2, τ_2) can be performed as:

$$(c_1\ ,\ \tau_1) \quad + \quad (c_2\ ,\ \tau_2) \qquad = \qquad (c_1 + c_2,\ \tau_1) \quad + \quad \tau_2 \tag{3.3}$$

which can further be reduced by using Equation 3.2. This is a sequential composition of operators.

Pipelining in FloPoCo 4.0

In the following, an example circuit is going to be built so as to give an idea of the design flow for an arithmetic operator in FloPoCo, as well as to illustrate the intricacies of the process. The chosen design is an example taken from the FloPoCo developer's manual [*], and describes a multiply-accumulate unit (MAC). A block schematic of the operator is presented in Figure 3.1.

Most of the code used for generating the datapath of an arithmetic operator is found inside the operator's constructor, as illustrated by the example of Listing 3.1. An exception to this remark is represented by code reuse and factoring through methods, macros, auxiliary classes etc..

Every datapath in FloPoCo is an operator . Thus, the core class of the generator is the `Operator`

[*] `http://flopoco.gforge.inria.fr/flopoco_developer_manual.pdf`

Listing 3.1: FloPoCo code generating a MAC unit

```
1  #include "Operator.hpp"
2
3  class MAC : public Operator
4  {
5  public:
6
7  MAC(Target *target, int wIn, int wOut): Operator(target)
8  {
9  setName("MAC");
10
11 addInput ("X", 2*wIn);
12 addInput ("Y", wIn);
13 addInput ("Z", wIn);
14 addOutput("R", wOut);
15
16 vhdl << declare("T", 2*wIn) << " <= Y * Z;" << endl;
17 vhdl << declare("R_tmp", 2*wIn) << " <= X + T;" << endl;
18 vhdl << "R <= " << range("R_tmp", wOut, 0) << endl;
19 }
20
21 ~MAC() {}
22 }
```

class, from which all the other operators inherit. This is shown in Figure 2.11, and in the case of the `MAC` operator, it is shown on line 3 of Listing 3.1.

The middle part of Listing 3.1, lines 9 to 18, shows the actual code needed for generating the datapath of the `MAC` unit. These instructions are stream operations to a FloPoCo VHDL stream, that will contain all of the code for the operator's datapath. The stream is an overloaded C++ stream, which accommodates extra functionality related to the performance specification (mainly to pipelining). The resulting VHDL code for the `MAC` unit is presented in Listing 3.2. It can be obtained by calling the `outputVHDL()` method of the `Operator` class.

The example illustrates how **certain design parameters have been abstracted** inside the constructor and how the **design is parameterized** . Take, for example, the arbitrary sizes for the inputs (`wIn`) and for the output (`wOut`). Of course these parameters could have been specified as VHDL generics, but FloPoCo's approach is at the same time much more powerful and flexible, and makes the design process easier and more fluid. Also, the parameters can be manipulated with more ease all along the design process.

The example of Listing 3.1 also shows how **signal declarations** are handled, e.g. line 16. Signal declarations are explicit in FloPoCo, and are achieved using the `declare()` function. They can be done on the first utilization of a signal. This is a small syntactic advantage over VHDL that requires

Listing 3.2: VHDL code generated for a MAC unit

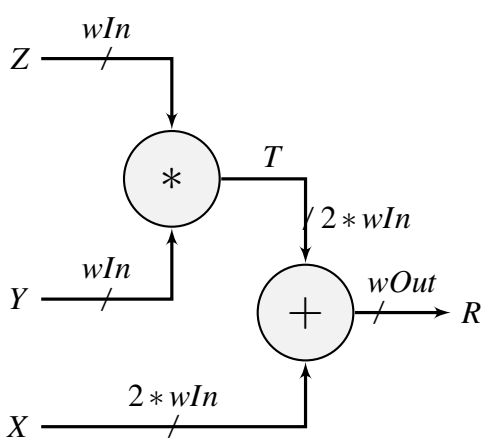
```
1  library ieee;
2
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_arith.all;
5  use ieee.std_logic_unsigned.all;
6
7  library work;
8
9  entity MAC is
10     port ( X   : in  std_logic_vector(63 downto 0);
11            Y,Z : in  std_logic_vector(31 downto 0);
12            R   : out std_logic_vector(63 downto 0));
13 end entity;
14
15 architecture arch of MAC is
16     signal T: std_logic_vector(63 downto 0);
17     signal R_tmp: std_logic_vector(63 downto 0);
18 begin
19     T <= Y * Z;
20     R_tmp <= X + T;
21     R <= R_tmp(63 downto 0);
22 end architecture;
```

signals to be declared before the beginning of the architecture. A declaration requires a *signal name*, and, optionally the *signal bit width* (e.g. signal `T` on line 16). Note that the inputs and the outputs of an operator are added using the `addInput()` and `addOutput()` methods, respectively (lines 11 to 14).

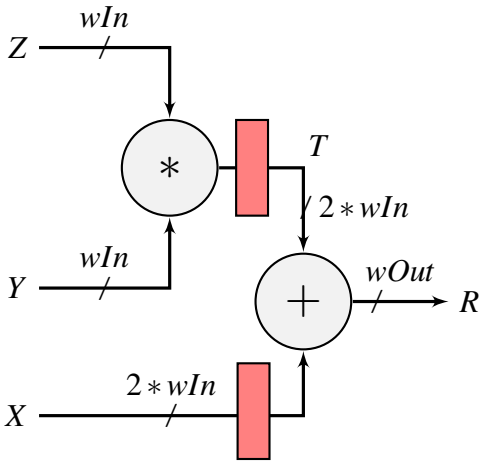
The `declare()` method serves double purpose. First of all, it constructs the architecture specification for a new signal. Second, it keeps track of signals by building a signal dictionary (used for automatic pipelining and detailed in Chapter 4).

This first version of the `MAC` operator only implements the functional specification: it creates the combinatorial datapath of the operator, as shown in Figure 3.2a. If the designer wishes to improve the performance of its operators, the frequency can be improved through pipelining, in the detriment of an increased delay at the output. A correct pipeline does not change the function implemented by an operator's datapath, only the time at which the results are produced. The pipelining process only affects the performance specification, and not the functional specification.

In FloPoCo terminology, a pipeline of depth n contains $n+1$ pipeline stages. They are separated by n synchronization barriers. For example, take the pipelined version of the `MAC` operators, as presented in Figure 3.2b. What the pipeline framework does is to associate a certain cycle to a signal's declaration. It then inserts the corresponding number of pipeline registers between a signal's declara-



(a) The combinatorial operator



(b) The pipelined operator

Figure 3.2: The multiply-accumulate unit in its combinatorial and pipelined versions

Listing 3.3: pipelined FloPoCo code

```
15 ...
16 vhdl << declare("T", 2*wIn) << " <= Y * Z;" << endl;
17 nextCycle();
18 vhdl << declare("R_tmp", 2*wIn) << " <= X + T;" << endl;
19 vhdl << "R <= " << range("R_tmp", wOut, 0) << endl;
20 ...
```

tion and a signal's use.

FloPoCo's pipelining methodology [37] orders nodes according to their *lexicographic ordering*. Every signal in the datapath is attributed a pair (*cycle*, *critical path*). A *cycle* is defined as the maximum delay between any two consecutive registers (or the delay between an input/output and a register, or between input and output for combinatorial circuits). A *critical path* is defined as the maximum delay since the last register, on the current path.

Listing 3.3 gives an example of how pipelining can be added to the datapath of an operator, as an evolution of the code in Listing 3.1. The most straight-forward synchronization method is the insertion of a *synchronization barrier*. The difference, as compared to the original code, is the call to the `nextCycle()` method, on line 16 of Listing 3.3.

Every signal declared inside the datapath of an `Operator` has an activation cycle, the cycle at which it is declared. The activation cycle is set by the `declare()` method for the internal signals. The same is done by the `addInput()` and `addOutput()` methods for the inputs and outputs of an operator, respectively. A call to the `nextCycle()` method advances the cycle at which the signals are declared, thereby inserting a new synchronization barrier. The cycle count starts, by default, at zero.

Listing 3.4: pipelined VHDL code generated for a MAC unit

```
1  library ieee;
2
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_arith.all;
5  use ieee.std_logic_unsigned.all;
6
7  library work;
8
9  entity MAC is
10     port ( X   : in  std_logic_vector(63 downto 0);
11            Y,Z : in  std_logic_vector(31 downto 0);
12            R   : out std_logic_vector(63 downto 0);
13            clk : in std_logic);
14 end entity;
15
16 architecture arch of MAC is
17     signal X, X_d1: in  std_logic_vector(63 downto 0);
18     signal T, T_d1: std_logic_vector(63 downto 0);
19     signal R_tmp: std_logic_vector(63 downto 0);
20 begin
21     process(clk)
22     begin
23         if clk'event and clk = '1' then
24             X_d1 <= X;
25             T_d1 <= T;
26         end if;
27     end process;
28
29     T <= Y * Z;
30     R_tmp <= X_d1 + T_d1;
31     R <= R_tmp(63 downto 0);
32 end architecture;
```

Listing 3.5: more advanced pipelined FloPoCo code

```
15      ...
16      manageCriticalPath(2e-9);
17      vhdl << declare("T", 2*wIn) << " <= Y * Z;" << endl;
18      manageCriticalPath(1e-9);
19      vhdl << declare("R_tmp", 2*wIn) << " <= X + T;" << endl;
20      vhdl << "R <= " << range("R_tmp", wOut, 0) << endl;
21      ...
```

Using the pipelining framework changes the resulting VHDL code from the one in Listing 3.2 to the one in Listing 3.4. It should be outlined that in the example VHDL code of Listing 3.4, the signal names ending in "`_dXX`" represent the fact that the respective signal is delayed by `XX` cycles. The listing also shows that the infrastructure required for pipelining is automatically generated by the pipelining framework. This is done based on a signal's lifespan, which is the largest difference between the activation cycle and the cycle at which the signal is used. The lifespan of a signal can increase as the signal is used inside the datapath of an operator.

One of the differentiating features of the FloPoCo generator from other available circuit design tools (like for example the vendor-provided ones) is the ability to generate designs at a user specified frequency. However, obtaining a frequency-directed pipeline would be quite hard work using only the techniques presented in Listing 3.3. It could possibly be achieved by introducing conditional statements around the pipelining directives of Listing 3.3. However, this would require quite an effort on the designer's side. Another shortcoming of the example of Listing 3.3 is that the level of granularity is quite low. The timing inside the circuit is only managed on the level of cycles/pipeline stages.

A better solution would be that of Listing 3.5, which shows an example of how to manage the cycle and the critical path inside the `MAC` unit. The calls to `manageCriticalPath()` are done before the corresponding datapath element (signal) is added. The listing also shows that the arguments to `manageCriticalPath()` are the delays of the elements being added to the datapath. The designer also has at its disposal methods allowing for a finer control over the timing inside the datapath it is creating. However, for the sake of the clarity of the `MAC` operator example, they have been left out. More details can be found in FloPoCo's developer manual[†].

Listing 3.5 also shows a shortcoming of the current approach to pipelining. The example of the `MAC` operator is pipelined for a target assuming that the delay of a multiplication is $2ns$, while the delay of an addition is equal to $1ns$. While this might certainly be true for a certain target device-target frequency combination, it lacks in generality. Therefore, a better solution would be to resort to the `Target` class for providing timing and architectural details with which to parameterize the

[†] `http://flopoco.gforge.inria.fr/flopoco_developer_manual.pdf`

Listing 3.6: more advanced pipelined FloPoCo code

```
15      ...
16      manageCriticalPath(target->DSPMultiplierDelay());
17      vhdl << declare("T", 2*wIn) << " <= Y * Z;" << endl;
18      manageCriticalPath(target->adderDelay(2*wIn));
19      vhdl << declare("R_tmp", 2*wIn) << " <= X + T;" << endl;
20      vhdl << "R <= " << range("R_tmp", wOut, 0) << endl;
21      ...
```

design. The resulting VHDL architecture is even closer to the designer's requirements. Listing 3.6 shows the minor changes needed, as compared to Listing 3.5.

In all the previous examples of Chapter 3 the `MAC` operator is assembled from the ground up. FloPoCo, however, offers an extensive library of operators that the designer can take advantage of[‡]. Listing 3.7 shows a version of the `MAC` operator designed using subcomponents. The example also uses some of the framework's other methods for manipulating the timing inside the datapath (such as `setCriticalPath()`, `syncCycleFromSignal()`, `getMaxInputDelays()`).

The subcomponents (e.g. the adder and the multiplier components in the example of Listing 3.7) are themselves operators. They can be used as any other C++ class, through instantiation. Further more, once the designer adds the `MAC` operator to FloPoCo's library, it will be able to use it as a subcomponent. The calls on lines 18 and 27 show how the subcomponents are connected to the design, in what timing is concerned.

There are several advantages to using subcomponents. The architectures they create can be better tailored to the designer's requirements than just using the $*$ or $+$ VHDL operators (where the designer is at the mercy of the predefined library functions). For example, a multiplier might be split into blocks matching the target device's hard multiplier blocks, allowing for a more resource effective solution and for better performance. An adder might be split into chunks, so as to meet the target frequency. The subcomponents receive as input the same `Target` object as the parent operator, thus they will be built for same frequency-driven performance specification as their parent operator.

As it can be seen from the examples of Listing 3.6 and Listing 3.7, there are several problems with this approach to pipelining an operator's datapath. First of all, the implementation of the functional specification is completely separated from the implementation of the performance specification. This is error prone, and produces operators that are difficult to maintain. Second of all, it is also quite verbose, requiring a significant amount of code dedicated to the implementation of the performance specification. This can distract the designer from a correct and optimal implementation of the functional specification. The solution for datapath pipelining proposed in [37] managed to well formalize the notion of a *cycle* inside a datapath, but badly formalized that of *critical path*. This entailed

[‡] `http://flopoco.gforge.inria.fr/operators.html`

Listing 3.7: pipelined FloPoCo code using subcomponents

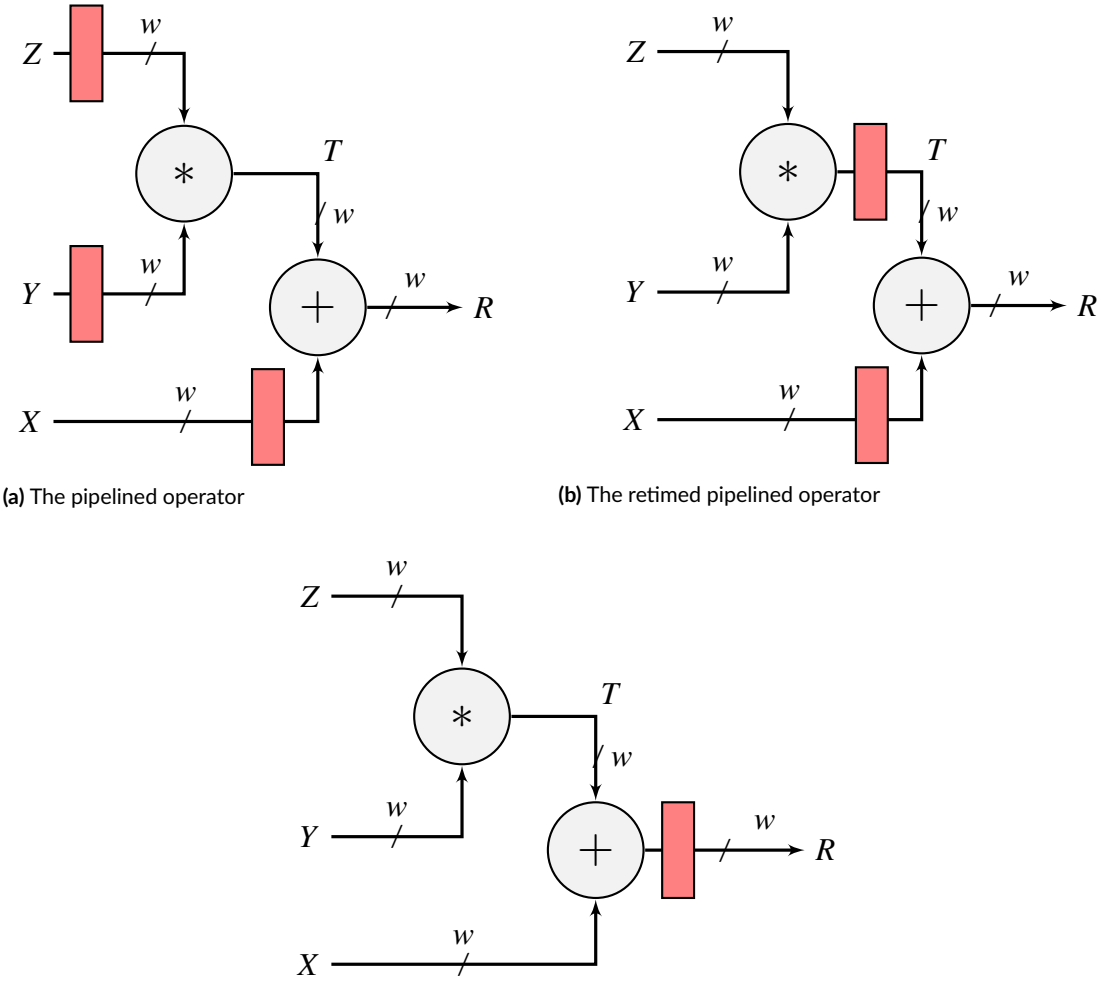
```
...
setCriticalPath(getMaxInputDelays(inputDelays));

IntMultiplier* my_mult = new IntMultiplier(target, wIn, wIn, ↩
    inDelayMap("X",getCriticalPath()));
oplist.push_back(my_mult);
inPortMap    (my_mult, "X", "Y"); // formal, actual
inPortMap    (my_mult, "Y", "Z");
outPortMap   (my_mult, "R","T");
vhdl << instance(my_mult, "my_mult"); // 2nd param is the VHDL ↩
    instance name

// advance to the cycle of the result
syncCycleFromSignal("T");
setCriticalPath(my_mult->getOutputDelay("R"));

IntAdder* my_adder = new IntAdder(target, 2*wIn, inDelayMap("X",↩
    getCriticalPath()));
oplist.push_back(my_adder);
inPortMap    (my_adder, "X", "X");
inPortMap    (my_adder, "Y", "T");
inPortMapCst(my_adder, "Cin", "0"); // carry in set to a constant ↩
    signal
outPortMap   (my_adder, "R","R_tmp");
vhdl << instance(my_adder, "my_add");

// advance to the cycle of the result
syncCycleFromSignal("R_tmp");
setCriticalPath(my_adder->getOutputDelay("R"));
vhdl << "R <= " << range("R_tmp", wOut, 0) << endl;
outDelayMap["R"] = getCriticalPath();
...
```

a sub-optimal pipeline, which was error-prone and inherently verbose. The next evolution of the pipelining framework tries to resolve some of these problems, while also facilitating the designer's effort. It is presented in Chapter 4.

State-of-the-Art on Automatic Pipelining

Before presenting ways of improving, or even achieving, the performance specification of an operator in Chapter 4, an overview of the state of the art in the field is provided. The literature on the subject is quite vast, so the current section focuses on the approaches currently used in the mainstream arithmetic operator generators.

One of the most important and relevant works in the field is that of Leiserson and Saxe [72]. Their work introduces the concept of **retiming** . Informally, retiming consists of moving pipeline registers from the output of gates to their inputs, or the other way around. This is done in such a way so as to only modify the timing of the circuit, and leave the functional specification unchanged. There are two main reasons for using retiming.



(a) The pipelined operator

(b) The retimed pipelined operator

(c) Another possible retimed version of the operator

Figure 3.3: The multiply-accumulate unit retimed

First of all, it can reduce the number of pipeline registers used. This is usually achieved by combining the registers on multiple inputs/outputs of a gate into registers on the outputs/inputs of the respective gate. There is, of course, the premise that there are less outputs than inputs (when replacing registers on the inputs), or vice versa. Take the example of the `MAC`, illustrated in Figure 3.3. The

method	cycle	delay within cycle	mathematical function
Leiserson & Saxe retiming	unchanged	changed	unchanged
Intel/FloPoCo pipelining	changed	changed	unchanged

Table 3.1: Comparison of pipeline generation techniques

total number of registers is decreased, when comparing the original version of the pipeline, shown in Figure 3.3a, and the retimed version shown in Figure 3.3b.

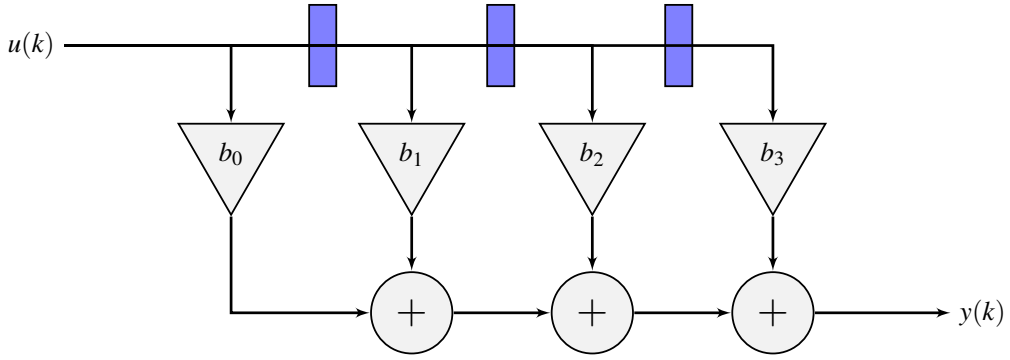
The second reason for retiming a circuit is that it can improve the frequency of a circuit. This is, in a way, the opposite process to the previously described one. Improving the frequency is achieved by increasing the number of registers in the circuit, all the while still meeting the functional specification. The difference between Figures 3.3c and 3.3b illustrates this point, the circuit of the latter having a higher frequency as compared to the former. The circuits are functionally equivalent, and both have a pipeline depth of 2.

In order to apply retiming on a datapath, the number of necessary registers, as well as their locations, must be known in advance. The authors of [72] also propose a method for inserting registers in a fully constructed datapath. Their technique is based on ordering the nodes of the datapath, according to their lexicographic order, and them assigning the nodes to the pipeline cycles.

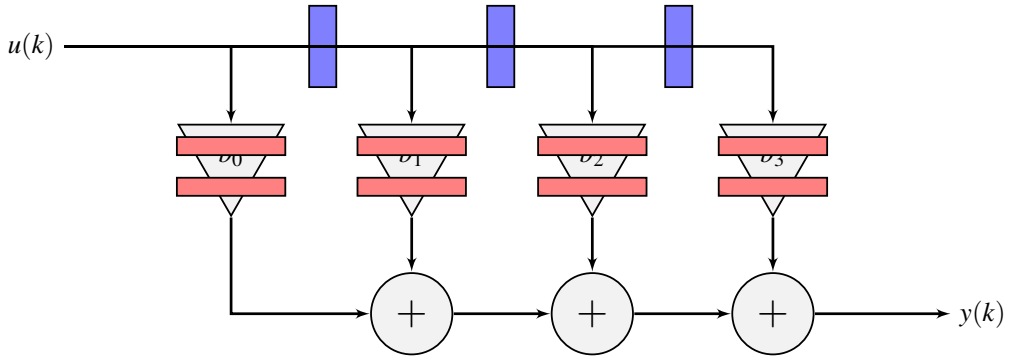
In spite of all of the advantages that the retiming technique offers, it remains quite complex and intricate to apply in practice. The designer must ensure that every modification that it makes to the circuit results in a circuit that is functionally to the original one, all along the retiming process. Another connected issue is the initialization of the registers, which is an equally difficult problem. There is also the scenario of multiple clock sources, and several other problematic situations. Due to all these aspects, it has taken a long time for retiming to make its way into mainstream commercial tools, such as Synopsys Synplify. Even so, its main use in current FPGA vendor-tools is to push registers into hard blocks inside FPGAs (DSP blocks, block-RAM etc), which is a very well-controlled scenario.

Leiserson and Saxe [72] deal with a circuit that already contains the pipeline registers. In order to pipeline a circuit using such a technique, registers must first be inserted at the inputs/outputs of the circuit, which are then pushed into the datapath by the retiming tools. This requires to know in advance the number of registers that are needed. Indeed, Leiserson and Saxe's retiming does not change the functionality of the circuit (when implemented correctly), including its cycle-level timing behavior. The problem studied here is slightly different: the cycle-level timing behavior can be changed. The number of registers to insert is also required. A comparison of the different techniques is shown in Table 3.1.

Intel's solution to the pipelining problem is briefly described in [28]. It draws inspiration from that of [72]. Their method consists of first building a directed acyclic graph (DAG) representing the circuit. Delays are then inserted in the DAG in a greedy fashion, similar to what was done in [37]. While the scheduling proposed in [28] is greedy, its conclusion seems to point to a shift towards a solution based on a constrained integer programming formulation of the problem.



(a) The original operator



(b) The pipelined operator

Figure 3.4: An example of a FIR filter

The authors identify three main challenges with this approach. The first are user-inserted delays/registers. This kind of registers are inserted for functional, rather than performance purposes. They are part of the functional specification. FIR (finite impulse response) filters are a good illustration of the use of these kind of registers. Figure 3.4a shows the architecture of a FIR filter (where the blue rectangles are the functional registers), while Figure 3.4b shows a pipelined version of the operator (where the red rectangles represent pipeline registers). The authors of [28] choose to ignore the the functional registers during the construction of the schedule. They are taken into account only when generating the VHDL code describing the datapath. This decision seems, however, to be a missed opportunity, deliberately ignoring extra information about the designed circuit.

The second challenge is represented by loops inside an operator's datapath. The authors' ap-

proach is to first break the loops. This transforms the graph into a DAG, which is subsequently scheduled. Once the schedule is created, the loops are reconnected. Even if not explicitly mentioned in [28], finding the points where to break the loops, so as not to introduce errors when reconnecting them, is a complex process.

The third, and final, major challenge identified by the authors of [28] are the sub-components. In their approach, during the pipelining process the subcomponents are treated as already pipelined, coarse grain blocks. This makes the overall schedule less flexible than having the ability to schedule subcomponents on-the-fly, for the specific context.

The approach described in [28] seems to be working at the level of abstraction which of the register transfer level (RTL). Xilinx's take on pipelining described in [46], however, works with placed and routed circuits. The circuits are also abstracted as DAGs. The authors' approach consists of first determining the critical path inside the circuit. This is the path with the longest accumulated delay, among all the possible paths of the circuit. Once determined, they insert registers on this path so that it respects the performance specification. All the parallel paths to the critical path are then determined and pipelined, so as to ensure the functional correctness of the circuit. The previous two steps are repeated until all the paths in the datapath respect the constraints imposed on the frequency.

On the plus side, this method offers an improved precision, seeing as it works with an already placed circuit. The chosen locations for the registers are almost final (save for the possible modifications introduced during the routing phase). In comparison, registers placed on the RTL level might be moved around during the place and route design phases. On the downside, though, the method requires at each step the determination of the critical path and all paths parallel to it. This can be quite computationally demanding.

More recently, Xilinx introduced a new tool (`report_pipeline_analysis`) for automatic circuit pipelining [49]. The tool tries to determine the pipelining potential of a circuit and then improve the circuit, when possible. They start by determining the maximum achievable frequency for a given circuit. This comes from either the delay of the largest loop, or from the maximum delay of any of the circuit's components. The next stage of the method is to iteratively determine so-called pipeline stages, which are made of subsets of nodes of the DAG representing the circuit. The subsets of nodes contain the so called critical nodes. The extra pipeline registers are placed at these specific nodes. During one iteration, the nodes connected to the critical ones are further excluded from the construction of the pipeline stage.

Matlab and Simulink [123] have an approach which is derived from the retiming technique of [72]. They work with a circuit that has already gone through the synthesis and placement phases. The authors' main contribution lies in the simplification of the retiming process, under specific conditions. One of the main drawbacks of retiming is the need to test and ensure the behavioral equivalence of the retimed circuit original version of the datapath (before the start of the retiming

procedure). Their improvement the authors of [123] provide over the original algorithm consists of removing the need for this test. This can be achieved by imposing a set of limitations on the initial design. The result is that they can considerably speed up the retiming algorithm, while benefiting from most of what the retiming technique has to offer. On the downside, they still retain the other shortcomings inherent to retiming, which have already been introduced.

Within this context, the solution for pipelining the datapaths of FloPoCo's operators tries to satisfy two main constraints. The first is the **portability** of the VHDL code describing the arithmetic operators. Therefore, creating the pipeline after the place and route design stages is not an option. The second is the possibility to **provide timing-related feedback** to the designer inside the operator constructor.

4

Automatic Pipelining in FloPoCo 5.0

As presented in Chapter 3 and illustrated in Listing 3.5, there are several inconveniences to FloPoCo's pipelining framework. Among these are the need for the user to advance and rewind the critical path by itself. Also, the design aspects related to timing are separated from those related to the architecture. The next evolution of the pipelining framework tries to improve on these aspects. All the while, it tries to maintain an important design feature in the FloPoCo generator: the ability to use timing information during datapath creation.

The examples of Chapter 3 have shown that the pipelining process is of a tedious and error-prone nature. Combine this with the need for a specific and effective pipeline for every combination of the supported targets and the desired frequencies in the framework. It is, therefore, desirable to automate this process as much as possible.

This chapter presents a method for generating a frequency directed pipeline for the datapaths of FloPoCo operators. The method requires minimal user intervention. As in the previous version of the pipelining framework, it requires the designer to specify the latencies of the datapath elements. These can be specified on a level of abstraction (finer or coarser) according to the user's knowledge of the circuit and target FPGA device. This is a reasonable prerequisite, as it comes in a stage of the operator design process when the user should be aware of these informations. This also allows the proposed pipelining method to work on a circuit specified at the RTL level, without the need for synthesis/placement to be performed. The overall operator design process is therefore very fast.

An overview of the proposed pipelining methodology is presented in Figure 4.1. Contrary to the solution of Chapter 3 and to most of the ones available through commercial tools, the proposed solution is *on-line*. The term *on-line* refers here to the fact that the pipeline is computed during the

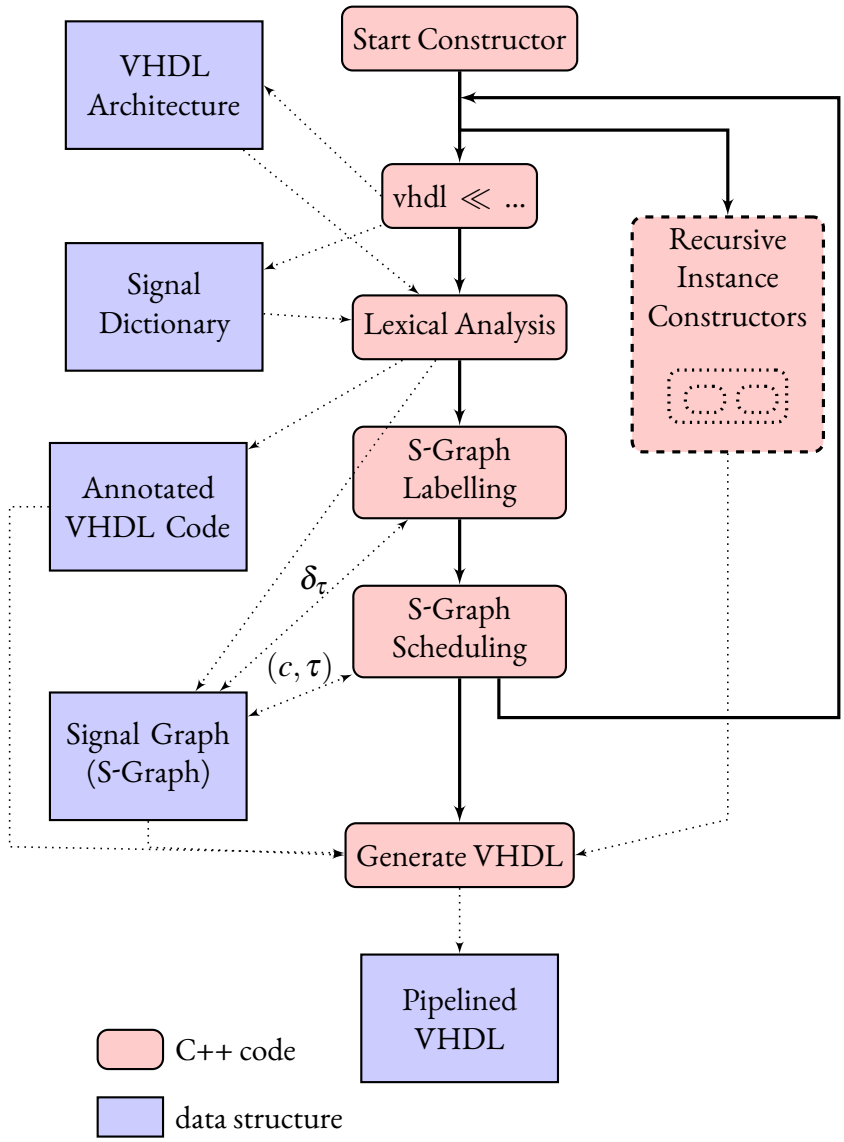


Figure 4.1: Constructor flow overview

Listing 4.1: pipelined FloPoCo code in the new pipelining framework

```
15      ...
16      vhdl << declare(target->DSPMultiplierDelay(), "T", 2*wIn) << " ↩
            <= Y * Z;" << endl;
17      vhdl << declare(target->adderDelay(2*wIn), "R_tmp", 2*wIn) << "↩
             <= X + T;" << endl;
18      vhdl << "R <= " << range("R_tmp", wOut, 0) << endl;
19      ...
```

construction of the architecture, and not afterwards. This is a major design decision: it limits the maximum efficiency of the scheduling strategies, but it makes the precious information of timing information available for use during the design process. This parameter is missing from the other mainstream circuit generators and it offers the user new opportunities design-wise.

As in the previous version of the pipelining framework, an operator's creation is started in the constructor, shown in the top part of Figure 4.1. These are the same kind of stream operations to a FloPoCo VHDL stream, as in the listings of Chapter 3. Listing 4.1 shows an part of an example design using the new pipeline framework.

It is the based on the `MAC` unit presented in Listing 3.5, but adapted to the new pipeline framework. The resulting VHDL code would be the same as the one generated by the code of Listing 3.5. This is, of course, assuming that the target device and the target frequency are the same.

Listing 4.1 illustrates the designer's intervention in the implementation of the performance specification in the new pipeline framework. The declarations of `T` and `R_tmp` contain the delay that the signals add to the datapath (`target->DSPMultiplierDelay()` and `target->adderDelay(2*wIn)`, respectively). This information is intended for the pipelining procedures. It is also the only information required from the designer in order to implement the performance specification. In the example of Listing 4.1, the delay added by `T` and `R_tmp` is specified in a generic way. The user can specify the delay directly, if he or she so chooses, but this may affect the operator's generality. Signal declarations can be done at any time, even after the first use of the signal. However, the common practice is to declare signals as they appear on the left-hand side of an assignment.

As illustrated in the top part of Fig. 4.1, most of the stream operations towards the FloPoCo VHDL buffer help build the combinatorial architecture of the operator, and at the same time a signal dictionary. As this is an on-line approach, the construction of both the architecture and the signal dictionary are done gradually, as the user inputs the description of the datapath.

The Signal Graph

Up until this point, Chapter 4 has only touched on the aspects related to the datapath design part of the pipelining process. This corresponds to the upper part of Figure 4.1. The rest of the chapter discuss the automatic transformation of the combinatorial architecture into a pipelined architecture. This corresponds to the middle and lower part of Fig. 4.1.

However the example presented up until now, the `MAC` unit, has been fairly simple. Therefore, a new, more relevant example is also used, in the form of a single precision floating-point adder (operator named `FPAddSinglePath`).

The relationships between the signals of an arithmetic operator's datapath can be extracted from the VHDL specification of the architecture. This can be accomplished by using a lexical analyzer, shown in the middle part of Fig. 4.1. Only a fairly low amount of computations need to be added to the lexer for this task, avoiding the need for a full-blown compiler making it very fast.

The lexing phase has two main outcomes. It uses the signal dictionary and the extracted relationships to build a signal graph (S-Graph). Representing a circuit as a directed graph is quite natural, and a common abstraction. Thus, the problem of pipelining a circuit can be reduced to the scheduling of a directed graph. The latter is a transformation that is conceptually straight-forward and well studied in multiple disciplines, while pipelining is a more difficult problem.

The informations extracted by the lexer are the data dependencies between signals of the operator's datapath. Take, for example, the code of Listing 4.2. The fragment is extracted from the architecture of the `FPAddSinglePath` operator. Looking at the example, there is a data dependency between signal `effSub` and signals `signX` and `signY`. Signals `signX` and `signY` are on the right-hand side of the assignment instruction to `effSub`. Therefore, the S-Graph associated to the operator contains edges from the node representing `effSub` to the nodes corresponding to `signX` and `signY`, respectively. Figure 4.3 is a zoom in on the part of the S-Graph (shown in full in Figure 4.2) that focuses on the signals concerned by Listing 4.2.

FloPoCo allows users to delay signals used in an operator's constructor by an arbitrary number of cycles. This is much like the user-inserted delays presented in Section 3.3, while discussing the approach of [28]. The information is extracted by the lexer, as well, for use in the construction of the pipeline, and inserted in the S-Graph. It represents a data dependency between signals, and is shown in Figures 4.2 and 4.3 as the labels on the S-Graph's edges.

At the same time, the lexing phase annotates the architecture's original VHDL code (either input by the user of generated by FloPoCo). The reason is to facilitate and speed-up the next pass over the architecture's VHDL code. A second pass is required towards the end of the processing flow (the bottom part of Figure 4.1), when re-inserting information on the computed pipeline. This is detailed in Section 4.4.

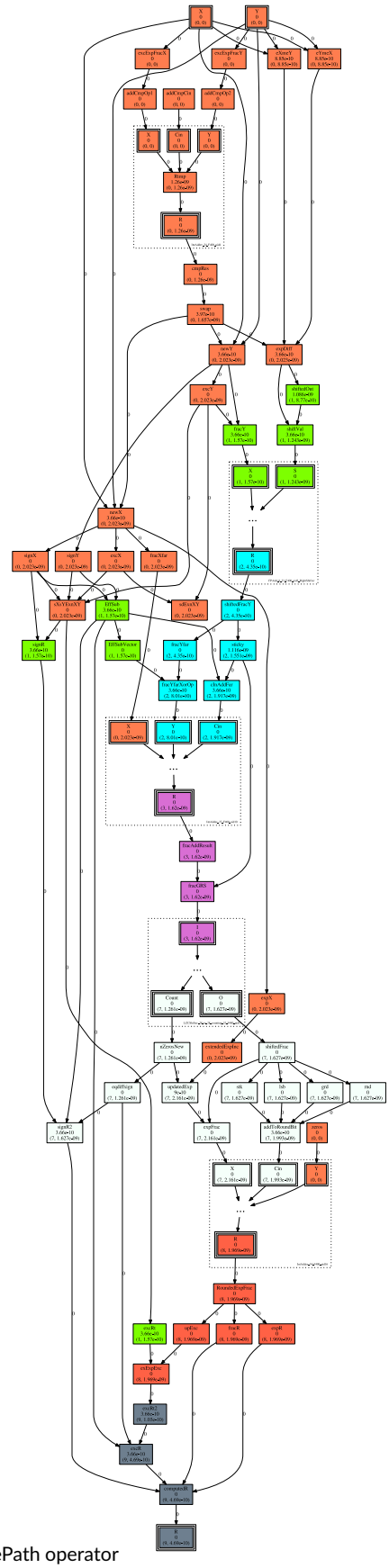


Figure 4.2: S-Graph of FPAddSinglePath operator

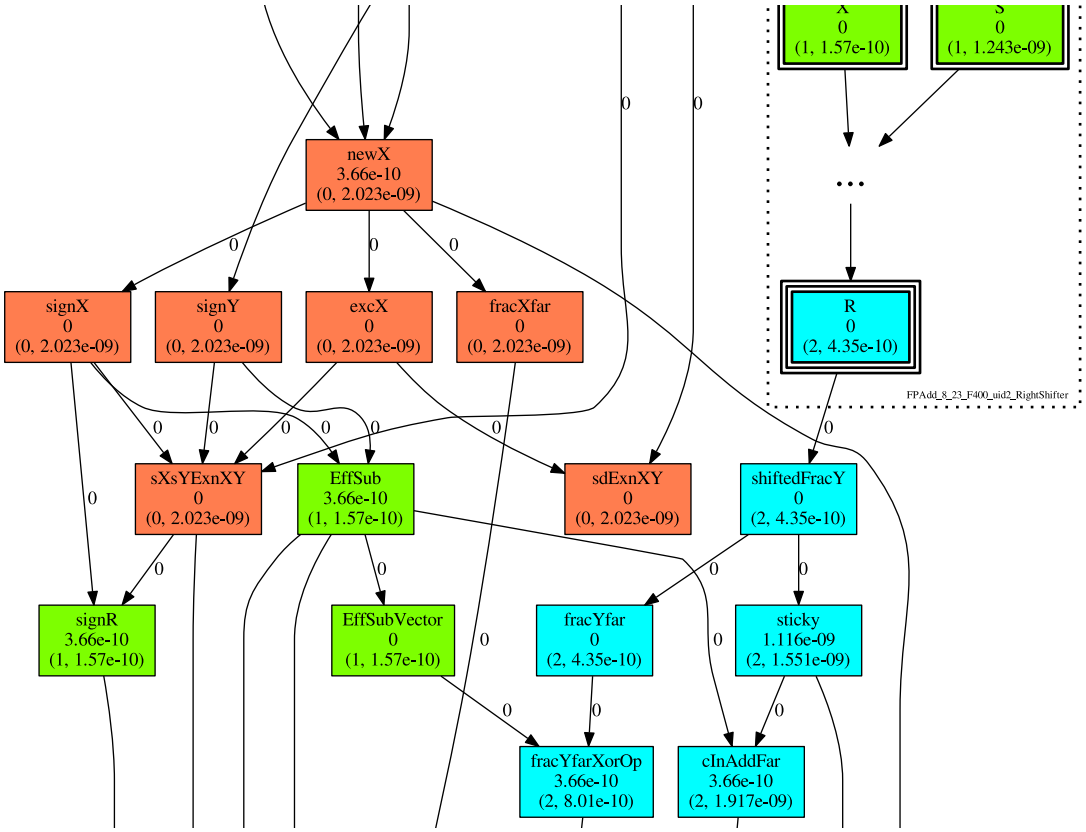


Figure 4.3: A zoom in on the S-Graph of FPAddSinglePath operator

Listing 4.2: pipelined FloPoCo code

```
15      ...
16      vhdl << declare("signX")  << "<= newX("<< wE + wF <<");";
17      vhdl << declare("signY")  << "<= newY("<< wE + wF <<");";
18      vhdl << declare(target->logicDelay(), "effSub")
19          << "<= signX xor signY;";
20      (...)
21      vhdl << declare(target->logicDelay(2), "excR", 2)
22          << " (...) when effSub ='1' (...)"
23      (...)
24      vhdl<< declare(target->adderDelay(wE+1),"eXmeY", wE)
25          << " <= (X" << range(wE+wF-1,wF)<<") "
26              << " - (Y" << range(wE+wF-1,wF)<<");";
27      ...
```

At this point, the S-Graph represents an ideal circuit, without any information about delays. Thus, the next step is to label the nodes with each signal's contribution to the critical path. The edges of the S-Graph are labeled with the cycle delays inserted by the user. In Figure 4.1, δ_τ represents a node's delay.

Every FloPoCo operator has an associated S-Graph. As shown in the top left part Figure 4.1, an operator's constructor can launch a chain of constructor calls for its sub-components. It is therefore natural to integrate the S-Graphs of sub-components in the S-Graph of the parent operator, as sub-graphs. The result is that a global operator (possibly containing sub-components) can be represented as a fully flattened S-Graph.

Fig. 4.2 shows an example S-Graph for the `FPAddSinglePath` operator. The figure is a compact representation of the graph. The content of the sub-graphs has been compressed (and is not shown), for the sake of a better legibility of the image as a whole.

The Scheduling Problem

Scheduling a circuit is a well studied problem which has numerous variations, depending on its specific scenario. It can be reduced to other similar problems from domains such as graph theory or multi-machine scheduling, among others.

Section 4.2 gives a formulation of the scheduling problem being solved in Section 4.3, within the context introduced in this chapter and 3.3.

An **on-line** [*] **scheduling problem** is considered, where signals become known over time. The objective is to schedule a set of interdependent signals (term used with the connotations of Section 4.1). The total number of signals is not known in advance. A signal's characteristics (e.g width, type, critical path contribution) are not known either. They are revealed only when the signal becomes available. However, a signal's dependences are not completely specified at the moment when the signal becomes available They can change with the arrival of other signals.

The objective for such a problem is the minimization of the operator's output signal(s) timing. This is discussed in Sec. 4.3.

On-line Scheduling of the S-Graph

The approach chosen for scheduling an operator's datapath, with the problem formulation of Section 4.2, is a greedy, as-soon-as-possible solution. Signals are scheduled as they become available, and that their precedence constraints are satisfied. The signals are scheduled to their earliest possible tim-

[*] the term on-line refers to the fact that the schedule is built during the operator's construction, as opposed to building it once the operator's architecture is completely generated

ing. This choice is constrained by the on-line nature of the scheduling algorithm. An overview of the scheduling process is presented in Algorithm 1.

As this is an on-line approach, Algorithm 1 shows that the scheduling process is invoked as soon as there are VHDL instructions that have not been processed (line 0). This approach ensures that every new instruction used for building the datapath has a scheduled S-Graph at its disposal.

The core of the scheduling algorithm is part of the loop at the top of Algorithm 1, lines 2 through 12. The loop is triggered as long as the VHDL instruction buffer still contains instructions that have not been processed.

The call to the EXTRACT_SIGNALS_TO_SCHEDULE() function on line 3 selects the signals that have been affected by the latest FloPoCo VHDL stream operations. These are the nodes in the S-Graph from where the scheduling must be started. The signals that are their direct and transitive successors must also be scheduled. The scheduling might be started multiple times from the same node. This is done so as to maintain compatibility with VHDL 's way of specifying concurrent instructions. VHDL concurrent instructions are considered to be executed in an infinite loop, all at the same time (emulated usually through a delta increment system). This justifies the need for restarting a node's scheduling, as its initial specification might not be complete (e.g. the node's predecessors are specified after the node).

The EXTRACT_SIGNALS_TO_SCHEDULE is detailed in lines 15 to 21. The call to the lexer (invoked on line 16) has already been presented in Section 4.1. The resulting signal dependences are used on line 17 to build the S-Graph, together with the existing signal dictionary. The method returns the nodes of the S-Graph where the scheduling should be started, as presented in the previous paragraphs, determined by POPULATE_SIGNALS_TO_SCHEDULE.

The calls to SCHEDULE_SIGNAL() on line 10 trigger recursively the scheduling process. It starts with the signals selected on line 19 and propagates to their dependences, as per the S-Graph, and depicted in lines 28-30. Lines 24-26 describe how a signal's timing is chosen. This is the smallest (lexicographically) pair (c, τ) which satisfies the following constraint:

$$(c, \tau) > (c_{\text{pred}}, \tau_{\text{pred}}) + \delta_\tau, \quad \forall \text{pred} \in \text{predecessors(signal)}$$

where δ_τ is the critical path contribution of the signal. This is also where new pipeline levels are added. The addition in the previous equation is a lexicographic time addition, as described in Section 3.1. The number of inserted pipeline cycles is equal to the cycle difference between a signal and its predecessor. When rescheduling a signal, a similar constraint must be met with respect to the successors:

$$(c, \tau) < (c_{\text{succ}}, \tau_{\text{succ}}) - \delta_{\tau\ \text{succ}}, \quad \forall \text{succ} \in \text{successors(signal)}$$

where $\delta_{\tau\ \text{succ}}$ is the critical path contribution of the successor signal, in addition to the original constraint.

Algorithm 1 S-Graph Scheduling

Require: partial circuit to schedule
Ensure: timing information available for current signals in S-Graph

```
1:  procedure SCHEDULE(instruction)
2:      while instruction_buffer ≠ ∅ do
3:          signals_to_schedule ← EXTRACT_SIGNALS_TO_SCHEDULE(instruction)
4:          if instruction ≡ NEW_COMPONENT_INSTANCE then
5:              new_instance ← CREATE_NEW_INSTANCE( )
6:              CONNECT_COMPONENT(new_instance)
7:              SCHEDULE(new_instance)
8:          end if
9:          for all signal ∈ signals_to_schedule do
10:             SCHEDULE_SIGNAL(signal)
11:         end for
12:     end while
13: end procedure
14:
15: procedure EXTRACT_SIGNALS_TO_SCHEDULE(instruction)
16:     new_signal_dependences ← LEXING(instruction)
17:     UPDATE_SGRAPH(new_signal_dependences)
18:     new_signals_to_schedule ←
19:         POPULATE_SIGNALS_TO_SCHEDULE(new_signal_dependences)
20:     return new_signals_to_schedule
21: end procedure
22:
23: procedure SCHEDULE_SIGNAL(signal)
24:     predecessor ← GET_LATEST_PREDECESSOR(signal)
25:     timing ← COMPUTE_SIGNAL_TIMING(predecessor, δτ(signal))
26:     SET_SIGNAL_TIMING(signal, timing)
27:     successors ← GET_SUCCESSORS(signal)
28:     for all successors do
29:         SCHEDULE_SIGNAL(successor)
30:     end for
31: end procedure
```

Functional registers, as introduced in Section 3.3 and Section 4.1, are taken into account during the S-Graph construction, S-Graph scheduling and a signal timing. They annotate the edges of the S-Graph. There can be several edges connecting the same pair of nodes of the S-Graph, as long as the edges are annotated with different delays. The functional equivalence between the combinatorial and the pipelined circuits is ensured when using functional registers.

The scheduling of subcomponent instances is presented between lines 5 and 7. As a component's datapath architecture might depend on timing information, the scheduling of a subcomponent is triggered only when all of the signals connected to its inputs have been scheduled. Implicitly, the generation of the datapath construction itself is triggered only when all of the signals connected to the subcomponent's inputs have been scheduled. This ensures that the timing informations are available on all paths inside the generated subcomponent. The call to CONNECT_COMPONENT on line 6 connects the input signals to the subcomponent to the signals in the parent operator. Therefore, the subcomponent is included as a sub-graph in the original operator's graph.

An operator can have subcomponents that belong (ideologically speaking) to one of two classes: shared or unique subcomponents. A shared subcomponent is reused multiple times with the same performance and functional specifications. A unique subcomponent, on the other hand, is used only once in the datapath, as its performance specification can lead it to having different architectures on each instantiation of the operator. For large designs, it is convenient to use shared subcomponents, wherever possible, as it results in a smaller size for the final VHDL code. A relevant example for unique subcomponents are the adders. It is desirable to have them split as best as possible so as to fill each of the pipeline changes. This means that each of their instances is specific to the situation. A compressor or a small look-up table, on the other hand, are good candidates for shared subcomponents, if their architecture remains the same for all their instantiations throughout the parent operator, whatever the timing.

There are certain limitations that need to be imposed on the schedule of a shared subcomponent. The main one is that the cycles of the signals inside the various instances of the subcomponent must match between the instances. This means that every signal must be at the same delay, in term of cycles, from its predecessors in all the instances. It is possible, though, that the signals have different critical paths, in different instances. If creating such schedules for all the instances is not possible, then the inputs to the instances are registered and the instances are rescheduled.

There are scenarios in which a subcomponent might need to be rescheduled, after its initial schedule. Take, for example, the `Wrapper` operator, shown in Figure 4.4. It needs to reschedule the operator it wraps, which is generated before the `Wrapper` operator itself. In such cases, rescheduling the operator is much simpler than creating the initial schedule. A great advantage is that the S-Graph is already created, and the relationships between the nodes are already extracted. The operator can, therefore, be scheduled much faster than the first time, even in a single traversal of the S-Graph. See-

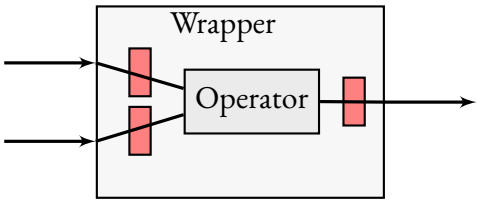


Figure 4.4: The Wrapper operator

ing as for the rest of the operators the scheduling strategy is greedy as-soon-as-possible, the same strategy is applied in this situation, as well.

VHDL Code Generation

After the pipeline generation, detailed in Sections 4.1 to 4.3, the FloPoCo design flow presented in Figure 4.1 on page 42 also handles the VHDL code generation.

The VHDL code is input by the user during the creation of an operator's datapath, as detailed in Chapter 3. With a few exceptions, such as the code produced by the `declare()` and `delay()` methods that allow the declaration and the use of a delayed version of a signal, respectively, this code ends up directly in the FloPoCo VHDL stream.

The lexical analysis, at the top of Figure 4.1 (the block labeled 'Lexical Analysis'), does not only extract the dependences between the signals in the datapath. It also annotates the original VHDL code, so as to mark the dependencies, identifying the left-hand side and right-hand side signals in the assignments. This process is illustrated with the help of Listings 4.3 (which shows the code describing the datapath of the `LeftShift` operator) and 4.4 (which presents the annotated version of the code in Listing 4.3). This eases the second pass over the VHDL code (performed in the block at the bottom of Figure 4.1, labeled 'Generate VHDL'). The pipelining information, obtained from the scheduled S-Graph of Section 4.3, is re-inserted in the VHDL code during the second pass over the VHDL code. Listing 4.5 shows the pipelined version of the code in Listing 4.3. The first annotation of the VHDL code, performed during the lexical analysis, allows for the use of a lexer for back-annotating the VHDL code.

Both the first and the second lexing stages are of a complexity that is linear in the size of the VHDL code. Thus both phases are fast, for the majority of operators, and only incur a small computational overhead.

The generation of the pipelined VHDL code concludes the pipelining process and the generation of the FloPoCo operators.

Some of the transformations described in Chapter 4 are also performed for combinatorial circuits. As the code does not need to be annotated, the two parsing phases can be disconnected for combina-

Listing 4.3: VHDL code generated for a LeftShift operator

```
(...)
level0 <= X;
level1 <= level0 & "0" when S(0)='1'
    else "0" & level0;
level2 <= level1 & "00" when S(1)='1'
    else "00" & level1;
level3 <= level2 & "0000" when S(2)='1'
    else "0000" & level2;
level4 <= level3 & "00000000" when S(3)='1'
    else "00000000" & level3;
R <= level4;
(...)
```

Listing 4.4: annotated VHDL code for a LeftShift operator

```
(...)
??level0?? <= $$X$$;
??level1?? <= $$level0$$ & "0" when $$S(0)$$='1'
    else "0" & $$level0$$;
??level2?? <= $$level1$$ & "00" when $$S(1)$$='1'
    else "00" & $$level1$$;
??level3?? <= $$level2$$ & "0000" when $$S(2)$$='1'
    else "0000" & $$level2$$;
??level4?? <= $$level3$$ & "00000000" when $$S(3)$$='1'
    else "00000000" & $$level3$$;
??R?? <= $$level4$$;
(...)
```

Listing 4.5: pipelined VHDL code for a LeftShift operator

```
(...)
level0 <= X;
level1 <= level0_d1 & "0" when S(0)='1'
    else "0" & level0_d1;
level2 <= level1_d1 & "00" when S_d1(1)='1'
    else "00" & level1_d1;
level3 <= level2_d1 & "0000" when S_d2(2)='1'
    else "0000" & level2_d1;
level4 <= level3_d1 & "00000000" when S_d3(3)='1'
    else "00000000" & level3_d1;
R <= level4;
(...)
```

torial versions of the operators. This simplifies the flow of Figure 4.1. Timing analysis for combinatorial circuits provides valuable information. It is particularly useful to have for performance analysis, as well as for testing and debugging purposes. This a progress with respect to the previous pipelining framework.

New Targets

The redesign of the pipelining framework brought about a change in the design paradigm used inside the FloPoCo generator. This change extends to the `Target` class introduced in Section 2.3.4, as hinted towards the end of the section. The pipelining framework has seen a shift towards a simpler and more effective approach. The same treatment has been applied to the `Target` class and its ecosystem. It is also an effort towards the possible development and inclusion of future ASIC targets.

The new `Target` class relies on several main methods, discussed in the following. The methods are mainly concerned with the timing aspects of the target devices.

- `ffDelay()` represents the delay (time required for a signal to propagate from the input to the output of a register) of a flip-flop, or register. For the time being, this delay is a constant value, specific to each target device, that has been obtained through the vendor-provided design suites.
- `logicDelay(n)` tries to model the delay introduced by an arbitrary logic function of n arguments. The implementation of the actual function is dependent on the architecture of the target device itself. For example, for Xilinx devices, the function takes into account the possible use of the multiplexers inside slices/CLBs (combinatorial logic blocks), which group several LUTs (look-up tables) together. Using the multiplexers avoids going out of the group, which, in return, reduces the routing delay. This method replaces the `lutDelay()` method of previous versions, which:
  - assumed a LUT-based architecture of the FPGA device
  - did not handle well coarse CLBs with multiple LUTs
- `adderDelay(n)` returns the delay of an addition of size n. As in the case of `logicDelay(n)`, the implementation is dependent on the target platform. For example, the carry chains have a length of 4 or 6 on Xilinx devices, depending on the generation and on the family, while they have a lenght of 5 or 10 on Altera devices.
- `multiplierBlockDelay()` and `memoryBlockDelay()` try to model the delay on the embedded multiplier and memory blocks, respectively. These functions are, however, complex as the embedded blocks usually contain internal registers, and can be use in multiple modes. An alternative to using these blocks directly is to abstract them and use the `Table` or `IntMult`

Operator	Target	estimated delay	measured delay
IntAdder 32		1.23 ns	1.54 ns
Shifter 63	Virtex6	2.2 ns	2.0 ns
FPAdd 8 23	(ISE)	19.2 ns	11.4 ns
IntAdder 32		1.4 ns	1.4 ns
Shifter 63	Kintex7	6.28 ns	6.8 ns
FPAdd 8 23	(Vivado)	19.8 ns	17.2 ns
IntAdder 32		1.26 ns	1.39 ns
Shifter 63	StratixV	2.68 ns	2.88 ns
FPAdd 8 23	(Quartus)	17.6 ns	9.8 ns

Table 4.1: Accuracy of the new `Target` models

operators, respectively. There is, though, a trend in the recent devices for increasing the number of dedicated multiplier and memory blocks. If the user requires to use them directly inside a design, these two functions might be the solution.

- `wideOrDelay(n)`, `eqComparatorDelay(n)`, `eqConstComparatorDelay(n)` attempt to capture the use of fast-carry logic to implement wide OR and wide AND operations.

For the methods detailed in the previous paragraphs, an estimation of the associated routing delay is also provided. This is contrary to the previous versions of the `Target` class, where the user had to estimate this delay itself.

An evaluation of the accuracy of the estimations provided by the `Target` class, in its current state, is provided in Table 4.1 and in Table 4.2. Results shown here are post place-and-route, including only the circuit elements (delay to/from pins not included, for example).

For a combinatorial version of an operator, the synthesis tools manage to find opportunities to merge the logic in the Look-up tables. For complex operators, this results in a decrease of the quality of the estimations. Table 4.1 illustrates this point. The comparison is made between three operators, with different levels of granularity and complexity, and using three different targets, with their associated vendor-provided synthesis tools. The first line, for each target, shows that for basic logic functions the estimations are accurate. The same still holds for the second line, where the constructor of the barrel shifter manages to predict how the synthesis tools group logic levels into the same look-up table. However, for a floating-point adder the estimations provided with the use of the `Target` models get farther away from the data obtained after synthesis.

When pipelining an operator, there are fewer levels of logic per pipeline stage, thus fewer opportunities for the synthesis tools to optimize and merge the logic. Table 4.2 illustrates this point. It shows the target device, the tools used for synthesizing the design, followed by the achieved performance and the resources occupied by the design on the target device. The performance is measured in terms

of number of cycles at a given maximum frequency, while the resource consumption as registers and look-up tables. Table 4.2 also shows illustrations of the corresponding S-Graphs for each operator, in the corresponding line, following the performance and resource measurements.

Target Device	Tool suite	Performance	Resource Usage
StartixV	Quartus	7 cycles @ 448 MHz	486R + 239L
Virtex6	ISE	9 cycles @ 424 MHz	500R + 495L
Kintex7	Vivado	9 cycles @ 427 MHz	517R + 359L

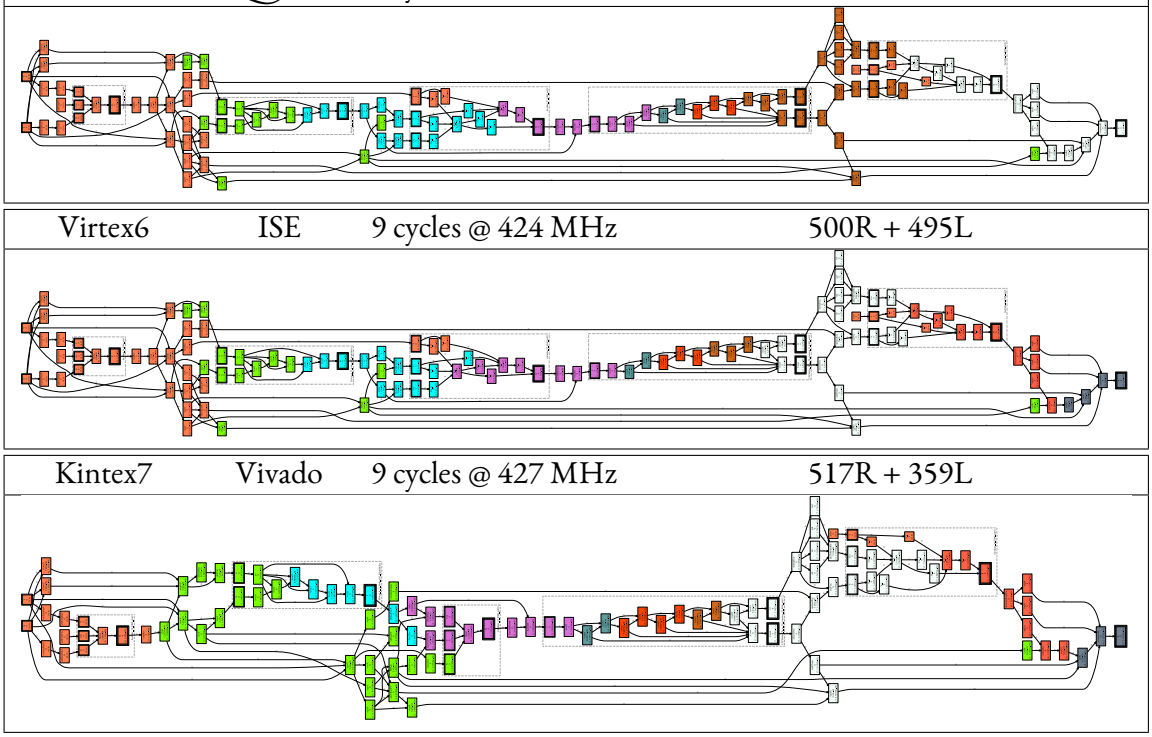


Table 4.2: Pipelining the FPAdd operator for a target frequency of 400MHz. for three different targets using three different vendor tools

Results

There are several benefits to the new pipeline framework. First of all, it results in a significant reduction of the code needed for the implementation of complex designs. In the case of the `FPAddSinglePath` used throughout Chapter 4 as a running example, the constructor is reduced from 557 lines of code to 472. Most of the eliminated code was used for directives connected to the specification of the pipeline, like synchronization management, which are no longer required. From an ergonomics point of view, not having to deal with local synchronizations of the critical paths is a great relief in the design effort on the user's side.

Specification		Performance		Resources
wE=8 wF=23	400 MHz	old:	9 cycles @ 423 MHz	604R + 233L
		new:	7 cycles @ 448 MHz	486R + 239L
	300 MHz	old:	7 cycles @ 305 MHz	505R + 208L
		new:	5 cycles @ 350 MHz	375R + 225L
	200 MHz	old:	3 cycles @ 232 MHz	281R + 248L
		new:	3 cycles @ 231 MHz	270R + 224L
	100 MHz	old:	2 cycles @ 132 MHz	234R + 264L
		new:	1 cycle @ 126 MHz	149R + 233L
	comb	both:	0 cycle @ 102 MHz	102R + 242L
wE=11 wF=52	400 MHz	old:	14 cycles @ 414 MHz	1690R + 628L
		new:	10 cycles @ 295 MHz	1250R + 509L
	300 MHz	old:	7 cycles @ 270 MHz	976R + 498L
		new:	7 cycles @ 235 MHz	929R + 501L
	200 MHz	old:	5 cycles @ 220 MHz	653R + 532L
		new:	4 cycles @ 246 MHz	579R + 512L
	100 MHz	old:	2 cycles @ 130 MHz	450R + 509L
		new:	2 cycles @ 123 MHz	352R + 492L
	comb	both:	0 cycles @ 82 MHz	198R + 514L

Table 4.3: Comparison of the FPAdd operator (single and double precision) between the old and new pipelining framework of FloPoCo. Results for Altera StratixV, using Quartus 16.0. Registers include the ones on the input and output ports.

The running times for operator constructors remain as fast as in previous versions of the FloPoCo generator. There are example of operators for which the running times for large, pipelined versions have been reduced. Working on the level of signals, as opposed to net, or bit level, has the advantage of simplifying and speeding up the whole pipelining process. For the majority of the operators, however, the running times were, and still remain, negligible, compared to the following stages in the design flow of a circuit, such as synthesis, placement and routing.

As shown by Table 4.3, in most cases the new pipelining framework manages to improve the performance of the operators, as compared to the previous implementation: for the same frequency specification, less registers are required, on most of the lines in Table 4.3. The operators have a reduced latency, in terms of pipeline cycles, while the frequency is increased and the resource consumption is decreased. For the double-precision operators, a decrease in the running frequency can be observed. This is mainly due to difficulties in capturing the abstractions of the circuit, as well as issues with the modeling of target devices. The issues should be ironed out in upcoming evolutions.

The data of Table 4.3 was obtained using the following command line:

```
flopoco target=StratixV frequency=400 FPAdd we=8 wF=23 Wrapper
```

The `Wrapper` operator is used for registering the inputs and the outputs of the `FPAdd` operator, which is previously specified in the command line. It is required so that the synthesis tools correctly analyze all of the datapaths inside the circuit. The target device used is a StratixV (5SGXEA3K1F35C1) FPGA, and the synthesis tool is Quartus 16.0. The synthesis process is started using the `tools/quartus_runsyn.py` utility of FloPoCo. Using the previously given specifications, the results should be reproducible. It should be noted that current design suites require the specification of a clock constraints file for timing to be performed. Therefore, FloPoCo generates it, along with each operator: the `.xdc` file for Xilinx Vivado, respectively `.sdc` for Altera Quartus. Without taking the pipeline into consideration, the corresponding versions of the pipelined operators are identical, between the two versions of FloPoCo.

Discussion and Future Improvements of the Pipeline Framework

This section discusses some of the choices made in Section 4.3, as well as possible future evolutions of the pipeline framework.

The requirements on when the scheduling procedure is triggered can be relaxed (currently it is triggered after each new VHDL instruction added by the designer), thus allowing for improvements to Algorithm 1. It can be noticed that the designer needs the S-Graph to be scheduled only when using instructions that use the timing information. Therefore, the scheduling procedures can be invoked only when timing informations are required, as opposed to invoking them after each

VHDL instruction. If no timing informations are needed during the construction of the datapath, the scheduler can be invoked only once, when the datapath has been built. This improvement represents a future work on the implementation of the scheduling procedures. It will allow improvements in the runtime of the algorithm, as well as open the way to new possible optimizations.

There is a point to be made about the signals selected as starting points for the scheduling procedures. Algorithm 1 only selects the signals which are on the left hand side of an assignment instruction. That is, signals which are assigned to. The signals on the right hand side, however, are not selected. This choice is valid as long as each signal is assigned to only once. Such a constraint on the datapath design style is quite reasonable, as most problematic situations are pathological cases. They can be avoided without much effort during design time. The trade-off is an important decrease in the number of paths traversed in the S-Graph.

The `FPDiv` operator serves as an example for illustrating this point. As can be seen in Figure 4.5, there are reoccurring dependences from signal `prescaledfY` to many of the other nodes in the datapath. The number of traversals of the S-Graph becomes exponential in the number of nodes times the number of their connections. Even for a relatively small operator such as `FPDiv_8_23` (with a width of the mantissa `wE`=8 and of the exponent `wF`=23) this renders the operator generation process a couple of orders of magnitude longer. Taking the right-hand side signals into consideration as points for restarting the scheduling increases the running time from 0.048 seconds to 16 minutes and 14.942 seconds. The running times were measured using the `Linux` utility `time`.

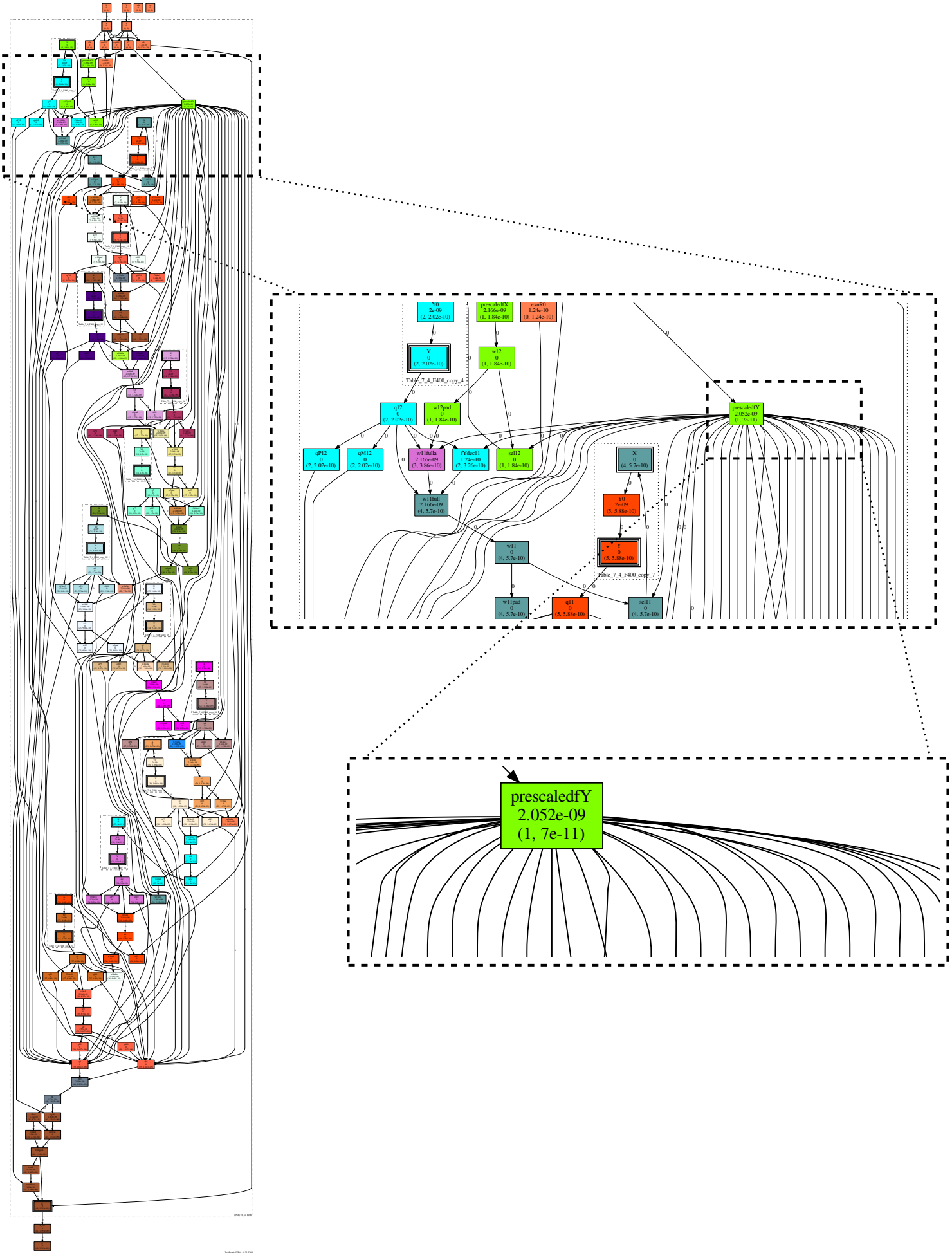


Figure 4.5: S-Graph for a simple-precision floating-point divider

5
Resource Estimation

An operator's generation is complete once the call to the constructor has finished, as shown in Figure 4.1. However, while this is true concerning an operator's functional and performance specification, extra functionality can still be added. Among these, the most notable example is the automatic test framework. It allows the generation of either a user-defined number of random tests, or predefined tests, usually for corner cases. It can even create exhaustive tests for operators, where applicable.

Another extra feature of the FloPoCo generator is the resource estimation framework. However, it has been lacking in popularity and visibility, mostly due to the fact that it has never made its way out of beta versions of FloPoCo.

In a typical design flow for FPGA targets, as illustrated in Figure 5.1, resource estimation reports are provided after the routing stage, when analyzing both timing and resource consumption. Intermediary estimations on resource consumption can also be had after the synthesis stage, but these are often less precise and subject to change in the following stages. As can be seen from the flow of Figure 5.1, such an iterative design style can become cumbersome, when the feedback on the decisions taken during design specification comes late in the design cycle [110]. Therefore, resource estimation is not just useful for predicting the hardware requirements, but also to indicate possible flaws in the design and shorten the development time [94].

Related Work

There are multiple types of methods for resource estimation [107]. First, there is the class of methods which relies on understanding the design suite. Understanding the design suite allows for predicting

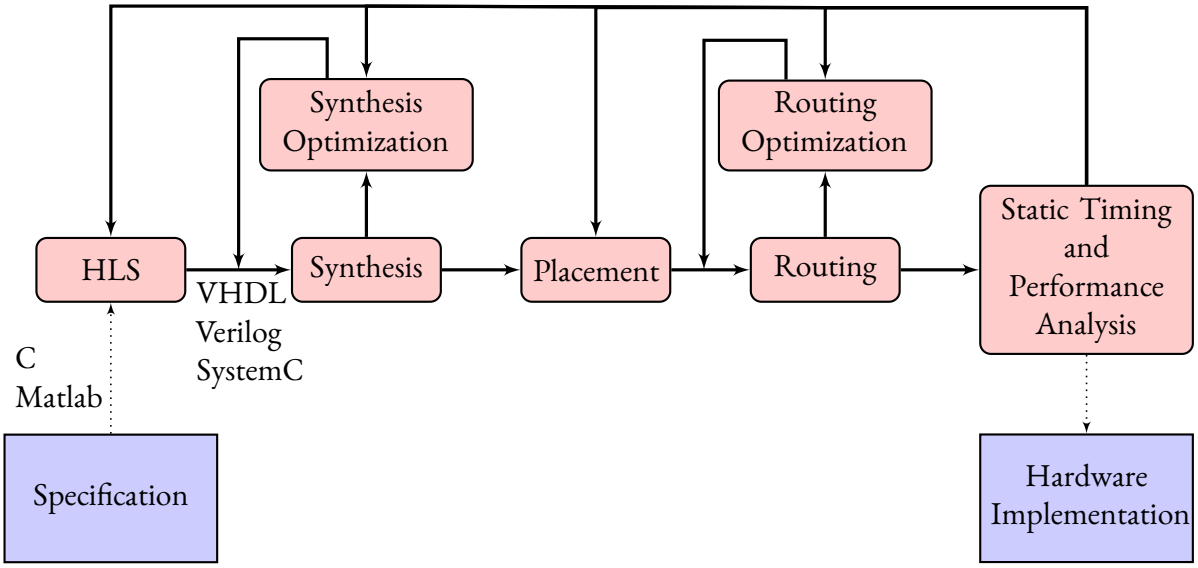


Figure 5.1: FPGA design flow from specification to hardware implementation

the resource consumption, as well as predicting the possible changes that tools can make on a design [22, 62].

There are also methods that take a bottom-up, constructive approach to resource estimation. They infer the resource consumption by assembling predefined cores. They use knowledge about the target FPGA in order to be as precise as possible. An example of such an approach can be found in [86].

The disadvantage with these two approaches is that they do not generally adapt well to change. A change in the underlying FPGA fabric, or a change in the basic building blocks used for designs, entails the need for a complete redesign of the resource estimation framework.

Another class of methods are iterative fitting approaches [62, 67]. Initial data for the resource estimations are obtained from test circuits. Estimation functions are then built using this data, together with knowledge about the target devices. The estimation functions are evolved using gradually more elaborated tests and by varying the parameters of the test designs. It is then iterated on this process, with new test designs, until the estimations are satisfactory.

There are resource estimation methods that split the process into several layers of abstraction [62, 110]. The first layer is the front-end, at the design specification level, with which the designer interacts. The next layer filters the design specification, from which it derives an abstract model. The resource estimation can be performed using the circuit model, knowledge about the target device and knowledge about the synthesis/placement/routing procedures. These types of methods perform well in terms of generality and extensibility, as compared to the other described methods.

The methods surveyed only cover a small part of a very vast domain, developed over several decades,

dating back to VLSI design. The main reason for reducing the surveyed domain is that the majority of the available methods rely on obtaining information about the design by parsing or partially synthesizing the code describing the circuit. If any gains are to be expected, as compared to the design flow of Figure 5.1, the resource estimation should not replicate the functionality of the synthesis stage. This would be time and resource consuming, as well as redundant [110].

User-guided Resource Estimation

The resource estimation framework in FloPoCo relies on the designer's knowledge of the circuit being created. As in the case of the pipelining framework, this requirement is not unreasonable. An arithmetic operator designer is usually aware of the elements that make up the operator it creates. Thanks to the designer's input, the need for synthesis-like processes is eliminated. However, the resulting estimation will be as good as the designer's input.

The first version of the resource estimation framework is designed in such a way so as to be accessible to both experienced and novice designers. A front-end layer, allows the user to model its operator in an abstract way and works in a similar way to the FloPoCo VHDL code stream. Adding new information about the resource consumption is similar to adding new VHDL instructions. The designer can add detailed information, at the level of look-up tables, registers, multiplier blocks etc., and go as low in the granularity of the design and the target device as needed. On the other hand, the designer can also use a coarser level of granularity and abstract the used components. For example, the resource consumption can be specified as logic functions of a given number of inputs, large multiplexers, adders, multipliers, tables etc.. The front-end is where the designer intervention stops. The rest of the resource estimation process is automatic and is part of the back-end.

There is another analogy to be made between the resource estimation framework and the pipelining framework: both are hierarchical. Estimations are made from the bottom up. As such, resource estimations for an operator containing only sub-components can be made using almost only the data obtained from its sub-components.

Another aspect worth mentioning is that the resource consumption due to pipelining is taken into account by the framework. The estimations are based on the pipelining framework. Therefore, there is no need for the designer's intervention for estimating the number of registers used.

As in the case of Section 4.5, measuring the performance of the first version of the resource estimation framework gives mixed results. Some results are shown in Tables 5.1 and 5.2. The test operators have been chosen so as to show the weaknesses and the strong points of the approach.

Table 5.1 shows the estimations for two implementations of the `CORDIC` sin / cos operator as described in Chapter 9. This operator was chosen as its resource consumption can be accurately modeled. As Table 5.1 shows, the accuracy of the predictions for both the logic and the registers remains

Precision (in bits)	Operator	LUT	Registers	DSPs	Estimated LUT (% deviation)	Estimated Register (% deviation)
12	CORDIC	611	545	0	701 (+12.83%)	597 (+8.54%)
	Reduced CORDIC	472	346	2	454 (-3.96%)	416 (+16.82%)
16	CORDIC	1026	873	0	1104 (+7.06%)	951 (+8.20%)
	Reduced CORDIC	706	545	2	690 (-2.31%)	629 (+13.35%)
18	CORDIC	1242	1070	0	1348 (+7.86%)	1158 (+7.59%)
	Reduced CORDIC	880	694	2	831 (-5.89%)	752 (+7.71%)
24	CORDIC	2007	1781	0	2199 (+8.73%)	1899 (+6.21%)
	Reduced CORDIC	1369	1095	2	1319 (-3.79%)	1187 (+7.75%)
32	CORDIC	3401	3179	0	3760 (+9.54%)	3265 (+2.86%)
	Reduced CORDIC	1927	1652	4	2148 (+10.28%)	1921 (+14.00%)
48	CORDIC	7027	6775	0	8934 (+21.34%)	6919 (+2.08%)
	Reduced CORDIC	4014	3580	8	5097 (+21.24%)	4060 (+11.82%)

Table 5.1: Resource estimation for two versions the `CORDIC` operator for a Xilinx Virtex 5 target using the old resource estimation framework

fairly constant as the size of the operator increases. As was the case for pipelining and delay estimations, the gap between the estimations and the synthesis reported resource requirements increases with the size of the operator. This is due to the optimizations performed by the synthesizer. The usage reports for the multiplier blocks is not analyzed in Table 5.1 as the estimations were always correct.

This discrepancy becomes even more evident for more complex operators. The second test case chosen is a floating-point `FFT` operator. Optimizations such as fusing signals or elimination of redundant signals increases the large difference between the reported resource requirements and the estimations. This effect can be observed in Table 5.2.

User-guided Resource Estimation for the New Design Paradigm

As in the case of the old pipelining framework, described in Chapter 3, there are some inherent problems with the approach. First of all, the resource estimation is not connected to the specification of an operator's datapath. Consequently, the resource estimations for an operator do not reflect its evolution. Second of all, separating the resource estimation from an operator's specification introduces a redundancy.

The solution for these problems lies in both the new pipelining framework, introduced in Chap-

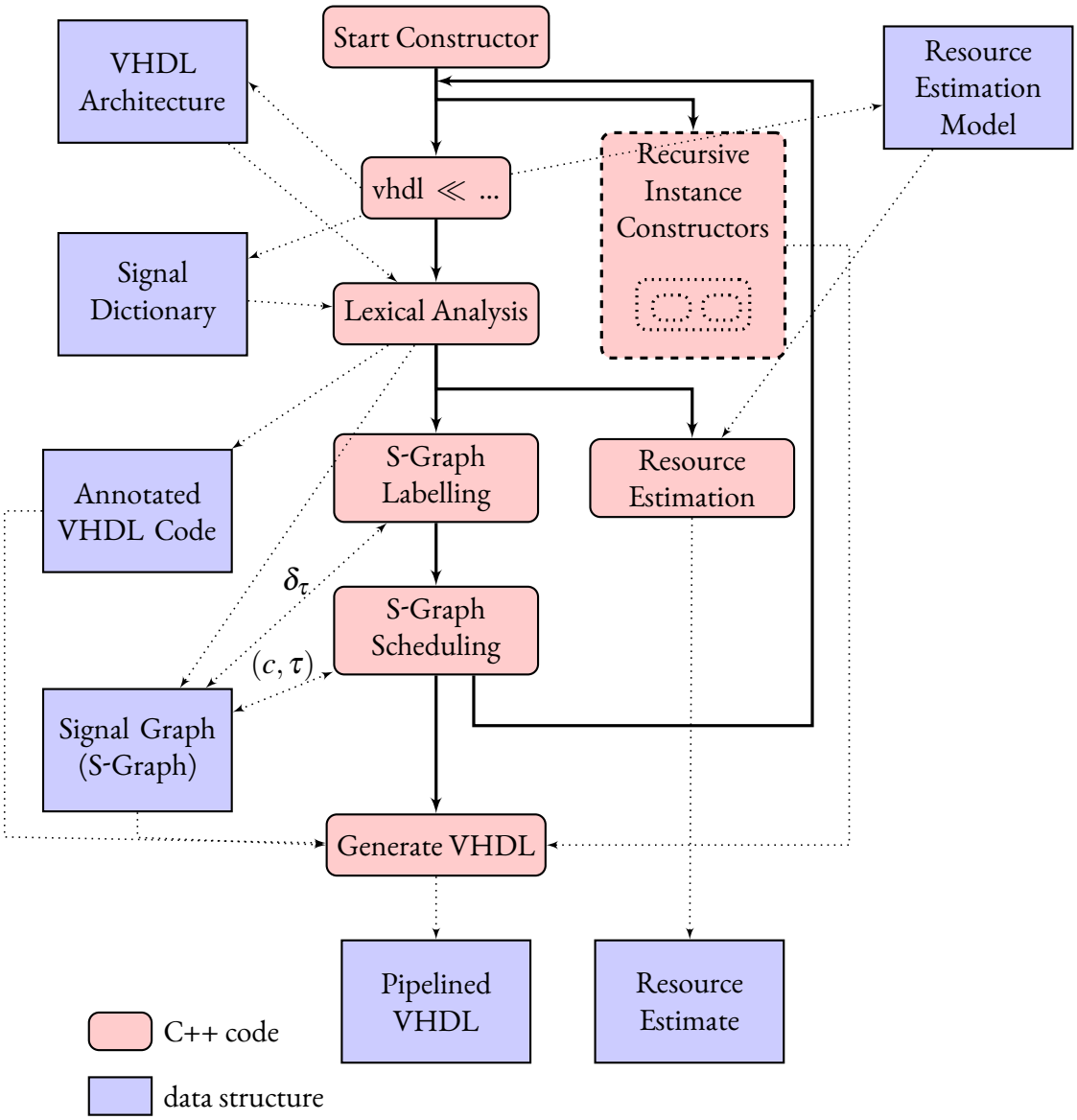


Figure 5.2: Constructor flow overview including resource estimation

Number of inputs	Precision $w_E + w_F$	LUT	Registers	Estimated LUT (% deviation)	Estimated Register (% deviation)
32	5+10	20388	11020	26794 (+23.09%)	21102 (+47.77%)
	8+23	49036	29224	52467 (+6.53%)	26214 (-11.48%)
	11+52	153490	124136	138785 (-10.59%)	95956 (-22.07%)
64	5+10	33263	15665	36821 (+9.66%)	29079 (+46.12%)
	8+23	134170	82096	142483 (+5.83%)	70150 (-17.02%)
	11+52	419975	342878	372401 (-12.77%)	310326 (-10.48%)

Table 5.2: Resource estimation for the FFT operator for a Xilinx Virtex 5 target using the old resource estimation framework

ter 4, and the new `Targets`, introduced in Section 4.5. The new front-end for resource estimations is therefore integrated in the pipelining framework. This solution comes quite naturally. In order for a designer to provide an accurate estimation of the delays in its operators, it needs to have a good understanding of the elements making up the datapath of an operator. Integrating the resource estimation framework's front-end in the pipelining framework's front-end eases the designer's effort.

With the new approach, the abstraction level is slightly increased. This makes for a good match with the current design suites, for the reasons exposed in Section 4.5.

As for the back-end of the resource estimation framework, the same approach still holds, requiring only minor adjustments so as to accommodate the new primitives used in the front-end. The precision of the back-end can be further improved, however. The VHDL code parsings described in Section 4.4 are currently only focused on extracting the data dependencies. The lexing could easily be extended (as a future work) so as to also extract the operations used inside the VHDL instructions. This would improve the model used for the circuit by the resource estimation framework, which would, in turn, help improve the estimations themselves. There are many subtleties, however, which cannot be detected by simply parsing the VHDL code. This is further amplified by VHDL 's ambiguous nature as a programming language. Detecting these subtleties remains in the charge of the designer.

Figure 5.2 gives an illustration of the way in which the new resource estimation framework integrates in the design flow. The figure focuses on the pipelining and the resource estimation processes.

Part II

Bitheaps and Their Applications

The man who moves a mountain begins by carrying away small stones.

Confucius, Confucius: The Analects

6 Bitheaps

Sections 2.3 and onwards introduced the `Operator` class as one of the main interest points from a developer's point of view. As shown in Figure 2.11, most of the FloPoCo framework is centered around the `Operator` class. There are, however, exceptions to this rule such as the `Bitheap`* class, as shown in Figure 6.1.

Bitheaps come from the literature on multipliers, from where the concept of bit arrays originates. Bitheaps are a generalization of bit arrays. The term denotes an unevaluated sum of weighted bits. At the same time, a bitheap is also a basic building block for creating arithmetic operators.

Bitheaps expose the bit-level parallelism. The notion originates from multiplier literature, so an example based on an integer multiplication is going to be used to illustrate and motivate the need for bitheaps. Consider the binary representation of a positive fixed-point binary variable X:

$$X = \sum_{w=LSB}^{MSB} 2^w \cdot x_w \tag{6.1}$$

where terms 2^w are the *weights* of the bits of X. MSB and LSB are the minimum and maximum weights, respectively. The variable X can be illustrated using a dot diagram as the one in Figure 6.2.

*The concept is classically referred to as *'bit array'* or *'bit heap'* in the literature. However, neither of these terms fully covers all the aspects of the concept, so the *'bitheap'* term is preferred all throughout Chapters 6 through 8

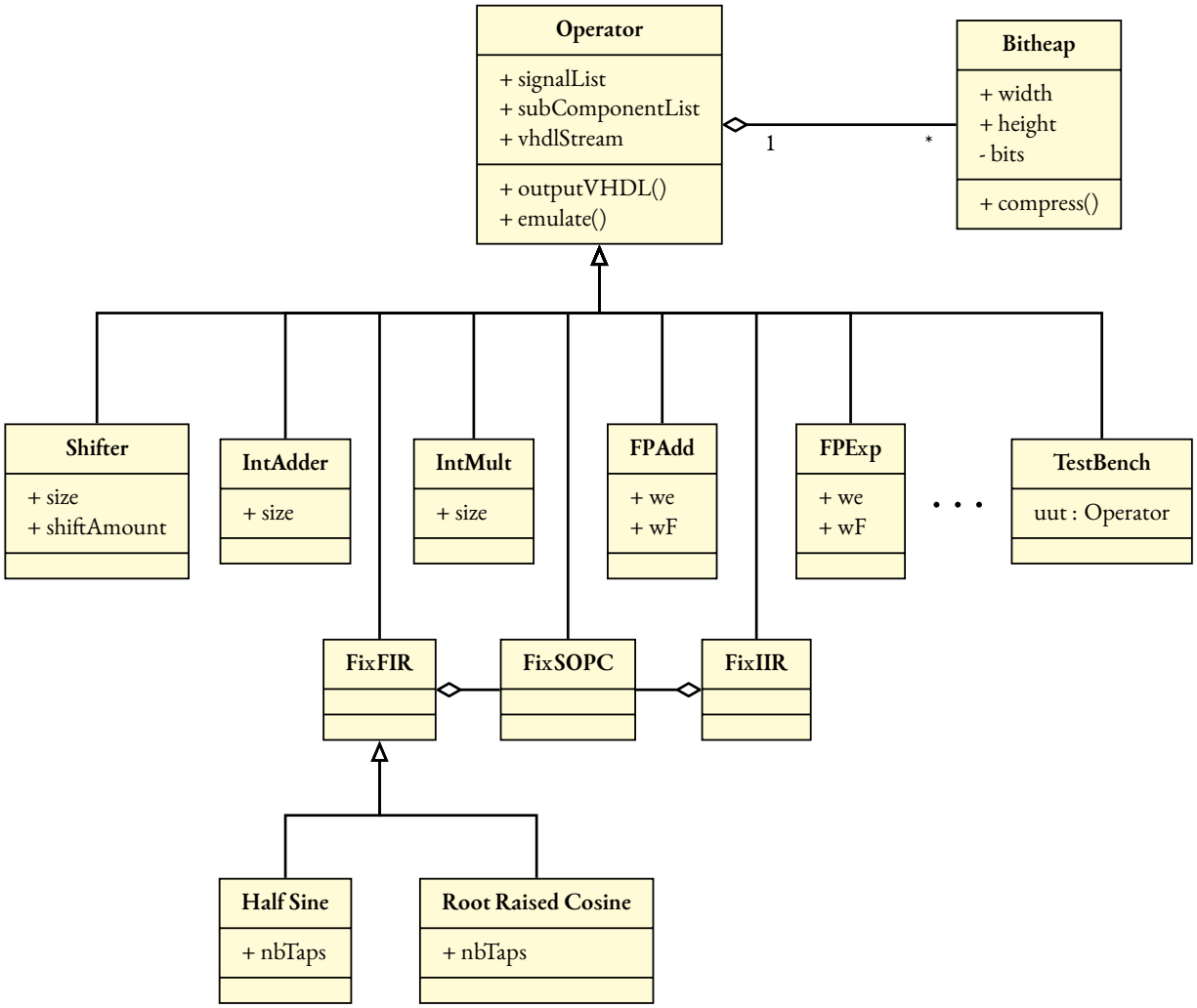


Figure 6.1: The Bitheap class in the class diagram for the FloPoCo framework

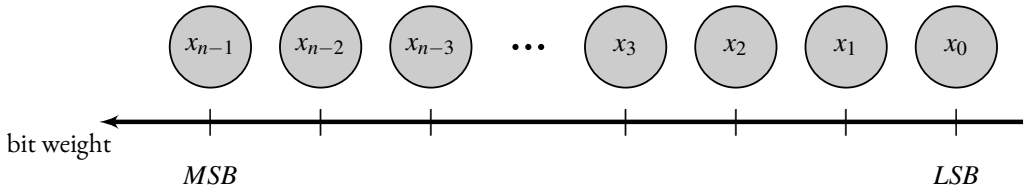


Figure 6.2: Dot diagram for a fixed-point variable X

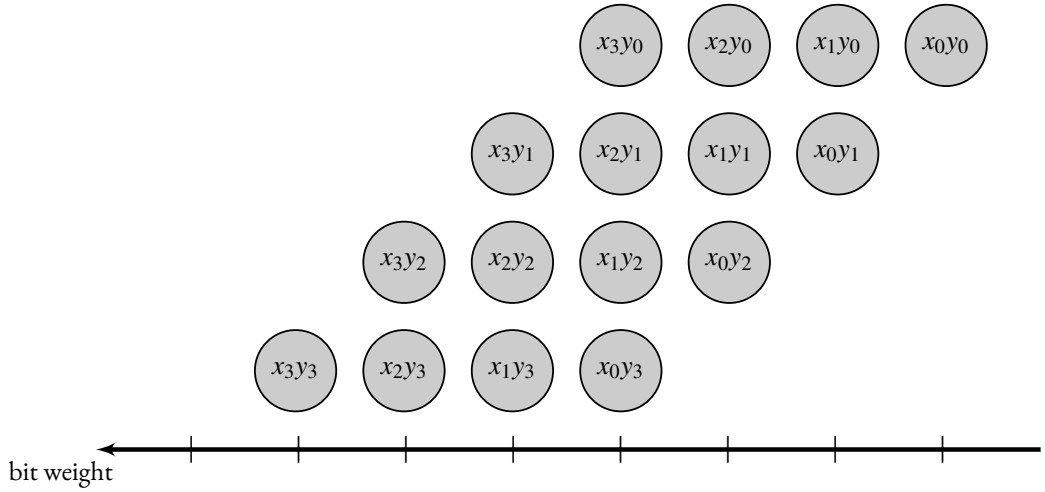


Figure 6.3: Dot diagram for the multiplication of two fixed-point variables X and Y

A second variable Y can be expressed in a similar way, and their product gives:

$$\begin{aligned} X \cdot Y &= \left(\sum_{w=w_{min}}^{w_{max}} 2^w \cdot x_w \right) \cdot \left(\sum_{t=t_{min}}^{t_{max}} 2^t \cdot y_t \right) \\ &= \sum_{w,t} 2^{w+t} \cdot x_w \cdot y_t \end{aligned} \tag{6.2}$$

which is also a sum of weighted bits. The $x_w \cdot y_t$ terms can be computed using a logical AND. This is classically illustrated as shown in the dot diagram of Figure 6.3. An alternative representation is given in Figure 6.4, where the bits in the columns are collapsed.

The advantage of the representation of Equation 6.2 and Figure 6.4 is that it exposes the low-level parallelism that is inherent to the operation. Indeed, addition is associative, so the order of the bits with the same weight in the columns is not important. Exploiting this bit-level associativity has enabled many fast multiplier architectures [30, 42, 96, 114, 129]. The hardware requirements are linear in the number of bits in the bitheap (the **size** of the bitheap) and it runs in time logarithmic in the maximum size of the columns of the bitheap (the **height** of the bitheap). If it is possible to construct the summation architecture for a multiplier, then it is also possible for any operator that can be expressed as a bitheap.

Even though they originate from multiplier literature, **bitheaps are not limited to multipliers** . The addition of two bitheaps is a bitheap. The multiplication of two bitheaps is a bitheap, as well. Therefore, polynomials (either univariate or multivariate) can be expressed as bitheaps. Other examples of functions that can be implemented as bitheaps are complex addition and multiplication, the multiply-accumulate in DSP processors, sums of products (and therefore digital filters), approxima-

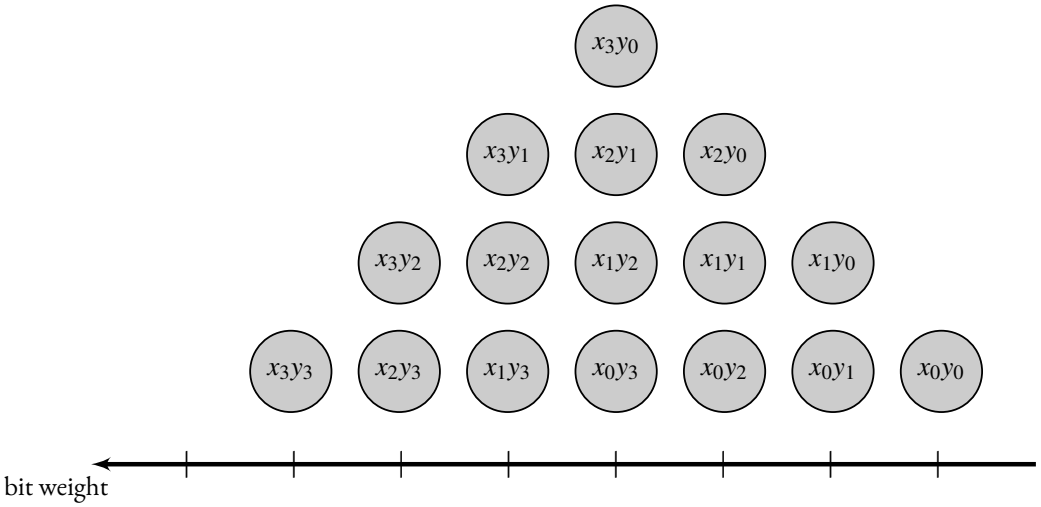


Figure 6.4: Dot diagram for the multiplication of two fixed-point variables X and Y. Bits in the same column have the same weight. The bits in the columns are collapsed.

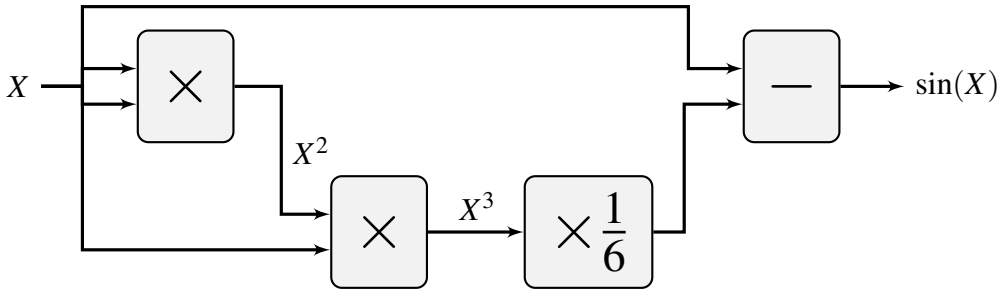


Figure 6.5: Classical architecture for evaluating $\sin(X) \approxeq X - \frac{X^3}{6}$

tions of functions (using polynomial approximations, tabulation) etc. [98, 117, 124].

Instead of assembling such coarse operators out of basic blocks such as adders, (constant) multipliers, tables etc. it is better to express them as bitheaps. This allows for a single, global optimization, instead of several local ones. It also produces more efficient operators, as compared to those assembled from sub-components.

Bitheaps allow for algebraic optimizations. In order to illustrate another opportunity made possible by the use of bitheaps, consider the sine operator. The sine of a fixed-point variable X can be computed using a Taylor polynomial of the third order:

$$\sin(X) \approxeq X - \frac{X^3}{6} \tag{6.3}$$

Discussions concerning the error analysis and the accuracy of the operator are deferred to Chapter 9.

A naive implementation for this operator is presented in Figure 6.5. It consists of two consecutive multiplications (or a squarer and a multiplier, in a more optimized implementation) followed by a constant multiplier by $\frac{1}{6}$ and a subtraction.

The sine operator can also be implemented using a bitheap. Exploiting the form of Equation 6.2, the X^3 term can be written as:

$$\begin{aligned} X^3 = & \sum_{i=i_{min}}^{i_{max}} 2^{3i} \cdot x_i^3 \\ & + \sum_{i_{min} \leq i,j \leq i_{max}} 3 \cdot 2^{i+2j} x_i x_j^2 \\ & + \sum_{i_{min} \leq i,j,k \leq i_{max}} 6 \cdot 2^{i+j+k} x_i x_j x_k \end{aligned} \tag{6.4}$$

The expression of Equation 6.4 can be simplified using basic properties of boolean algebra and some classical tricks:

- $x_i^3 = x_i^2 = x_i$ (with x_i a boolean variable)
- $2 \cdot 2^{i+j+k} \cdot x_i x_j x_k = 2^{i+j+k+1} \cdot x_i x_j x_k$
- $3 \cdot 2^{i+2j} x_i x_j$ can be re-written as $2^{i+2j+1} x_i x_j + 2^{i+2j} x_i x_j$

Therefore, the expression for the sine operator can be written as:

$$\begin{aligned} X - \frac{1}{6}X^3 = & \sum_{i=i_{min}}^{i_{max}} 2^i \cdot x_i \\ & - \frac{1}{3} \sum_{i=i_{min}}^{i_{max}} 2^{3i} \cdot x_i \\ & - \sum_{i_{min} \leq i,j \leq i_{max}} \cdot 2^{i+2j-1} x_i x_j \\ & + \sum_{i_{min} \leq i,j,k \leq i_{max}} \cdot 2^{i+j+k} x_i x_j x_k \end{aligned} \tag{6.5}$$

The division of X^3 by 6 is fortunately compensated by a factor 3 in some of the terms in the expansion. Even without taking into account the multiplication by $\frac{1}{3}$, there is already a significant decrease in the amount of hardware required for implementing the sine operator using Equation 6.5 (only about a third of the initial requirements).

A more naive approach for the multiplication by $\frac{1}{3}$ is to use the binary expression of the constant, $0.101010\ldots$, truncated to an intermediary precision, and to perform the multiplication. A

second option is the multiplication by the constant $\frac{1}{3}$, using a dedicated constant multiplier. Without delving into too many details, a candidate method for this task is be the tabulation-based KCM method. It adds the partial results of the multiplication by the constant, coming from the tables, to the bitheap. A slight improvement can be made, taking into consideration that $\sum 2^{3i} \cdot x_i$ is of the form $00x_i00x_i00x_i \ldots$. A third option is the division by 3, for which there are also efficient implementation techniques.

Techniques similar to the ones used for the implementation of X^3 can be generalized for the high-order terms of polynomials. Computing them based on bitheaps will generally have a lower complexity, a lower latency and incur a lower cost in terms of hardware requirements. As a side-effect, truncations and sign management are easier to manage (more details on these topics are provided in the following sections).

Bitheaps are versatile tools: a bitheap is a meta-operator. It is convenient to have an operator generator based on the bitheap concept for several reasons.

From a developer's point of view, having access to an optimized, flexible and versatile data structure such as the bitheap eases the development of arithmetic operators. In order to create the architecture of its operators, the designer only needs to throw bits (results of various computations) into the bitheap and then call the generic methods that will build an architecture computing the sum of these bits. The bits in a bitheap can come from various sources (logic, hard multiplier blocks, embedded memory blocks etc.) and even from multiple operators.

Another reason for which bitheaps are convenient is that it integrates well with the heterogeneous available resources on an FPGA device. As shown in the rest of Chapter 6, bits coming from look-up tables, multiplier blocks and memory blocks can be managed inside an arithmetic operator generator based on bitheaps.

What sets FloPoCo's approach to bitheaps apart, from other available ones in the literature, is the systematic management of the bitheaps and the construction of a framework which is dedicated to their management, in an effort to make them an ubiquitous tool.

Most of the literature assumes that all the bits are present at the moment when the compression is started [96][114]. However, this not necessarily true, especially for designs making use of multiplier and memory blocks, that have a considerably longer delay, as compared to that of the look-up tables. Therefore, new data structures and compression strategies are required for this new context.

There are several ways in which a bitheap can be viewed. From a **developer point of view, a bitheap is a dynamic data structure** in which bits are added and removed during the construction of an arithmetic operator. Bits are initially added, then removed during compression, then new bits are added, as the results of the compression and so on. From a **circuit point of view, a bitheap is a series of interconnected compressors** (each feeding bits for compression into the compressors on the next level). This is during the implementation of the circuit (using notions such as cycles, critical paths or

delays).

Section 6.1 presents some technical details and characteristics of the bitheap framework implemented in FloPoCo. Section 6.1.2 shows that signed numbers can be managed without much additional effort. Details about the compression of a bitheap are presented in Section 6.2. In Chapter 7, ways in which bitheaps can help improve the speed and efficiency of multiplications are investigated. Chapter 8 presents a related problem, that of multiplications by constants.

Technical Bits

In most of the relevant literature, bitheaps typically belong to a single arithmetic operator. But this is not necessarily true. As large operators often rely on sub-components coming from a library of available operators, it is therefore natural to share the same bitheap between all the components and have only one global compression (where possible). This is also the approach used in FloPoCo, where operators can either be built independently (relying on their own local bitheap), or integrated in a larger design (using a shared bitheap).

Bitheap Structure

At it's core, a bitheap is built around a data structure that stores the bits. The bits are organized in columns, based on their weight. Inside each column, the bits have the same weight and are sorted by their time of arrival in the bitheap, this time being the lexicographic time introduced in Section 3.1.

A bitheap naturally has support for fixed-point numbers.

A bitheap has several **basic attributes**, illustrated by Figure 6.6:

- the **MSB**; this is the largest weight that a bit added to the bitheap can have.
- the **LSB**; this is the smallest weight that a bit added to the bitheap can have.
- the **signedness**; this shows whether the bitheap deals with signed or unsigned numbers.
- the **height**; it represents the maximum height of any of the columns of the bitheap.
- the **size**; it represents the total number of bits in the bitheap.
- the **heap**; it contains the bits which make up the bitheap. This can be any data structure that allows the storage of the bits in columns according to their weight.
- the **compression strategy**; this is used for summing up the bits of the bitheap so as to reduce the height to one.

There are several remarks to be made concerning the attributes of a bitheap.

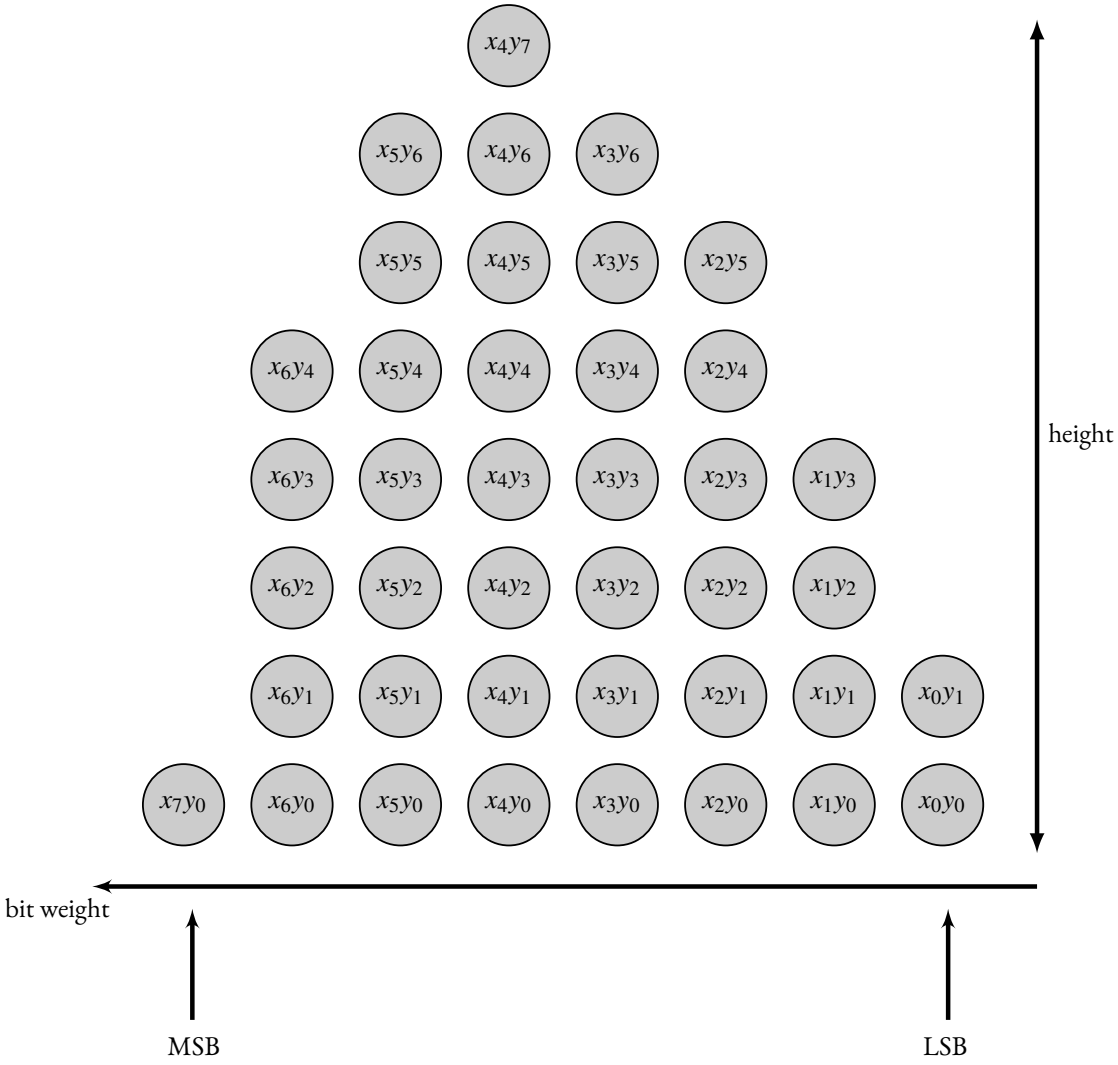
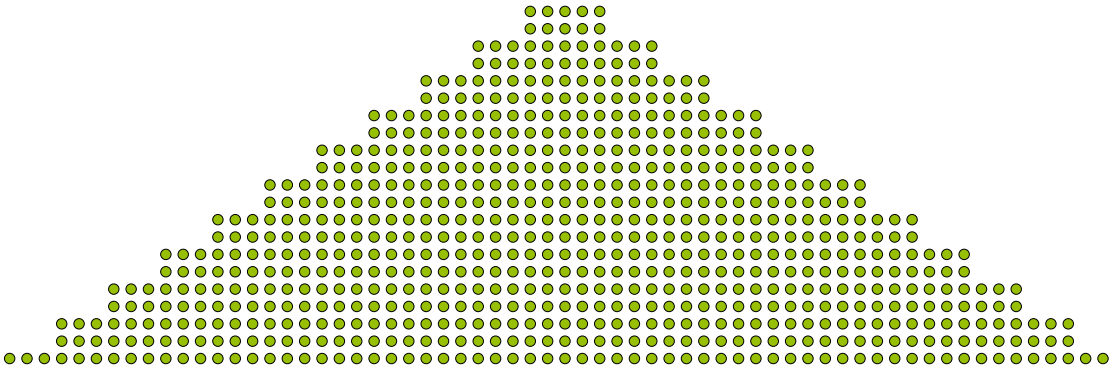
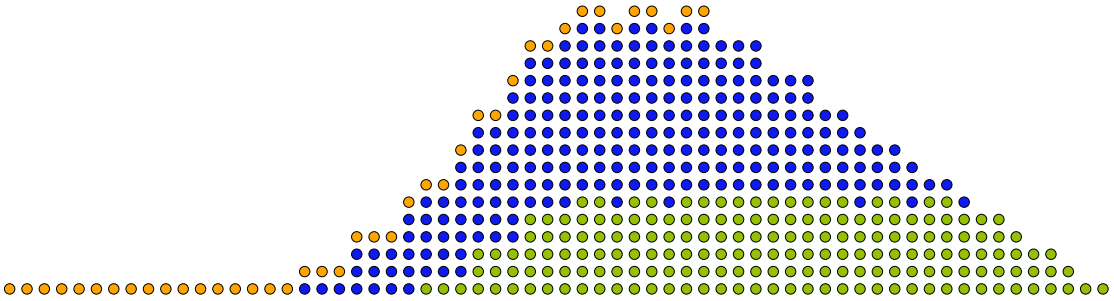


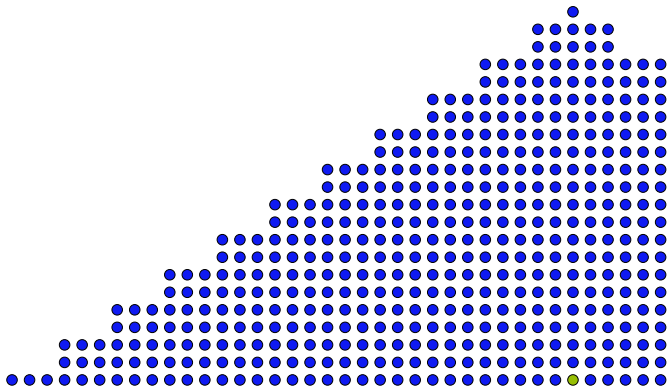
Figure 6.6: Dot diagram for a bitheap showing its basic attributes



(a) Bitheap of a 32-bit by 32-bit multiplier with the result on 64 bits (using logic elements only)

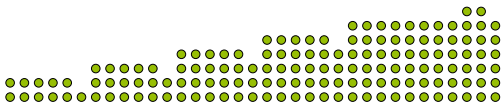


(b) Bitheap of a 32-bit by 32-bit multiplier with the result on 64 bits (using logic elements and DSP blocks)

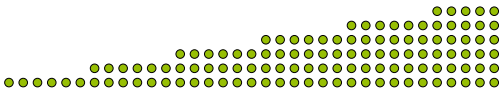


(c) Bitheap of a 32-bit by 32-bit multiplier with the result truncated to 32 bits

Figure 6.7: Example of bitheaps coming from FloPoCo's IntMultiplier, corresponding to an integer multiplier, with the two inputs on 32 bits. Figure 6.7a illustrates the multiplier implemented using only look-up tables, with the output having the full precision (64 bits). Figure 6.7b shows the multiplier implemented using not just look-up tables, but also the embedded multiplier blocks. The corresponding bitheap is of a considerably smaller size. Figure 6.7c illustrates the same operation, but the output is truncated to 32 bits. Only the partial products which form part of the final result are computed, and thus shown in the figure.



(a) Bitheap of a 32-bit constant multiplier by $\cos(\frac{3\cdot\pi}{4})$



(b) Bitheap of a 32-bit constant multiplier by $\log(2)$

Figure 6.8: Example of bitheaps coming from FloPoCo's FixRealKCM operator, corresponding to multiplications by the constants $\cos(\frac{3\cdot\pi}{4})$ and $\log(2)$, respectively. The inputs to the multipliers have widths of 32 bits. The technique used for the multiplications is the tabulation-based KCM method. This is much more efficient than the typical multiplication of Figure 6.7. More details on this topic are presented in Chapter 8.

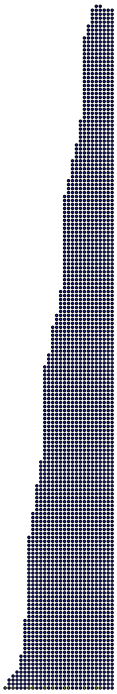
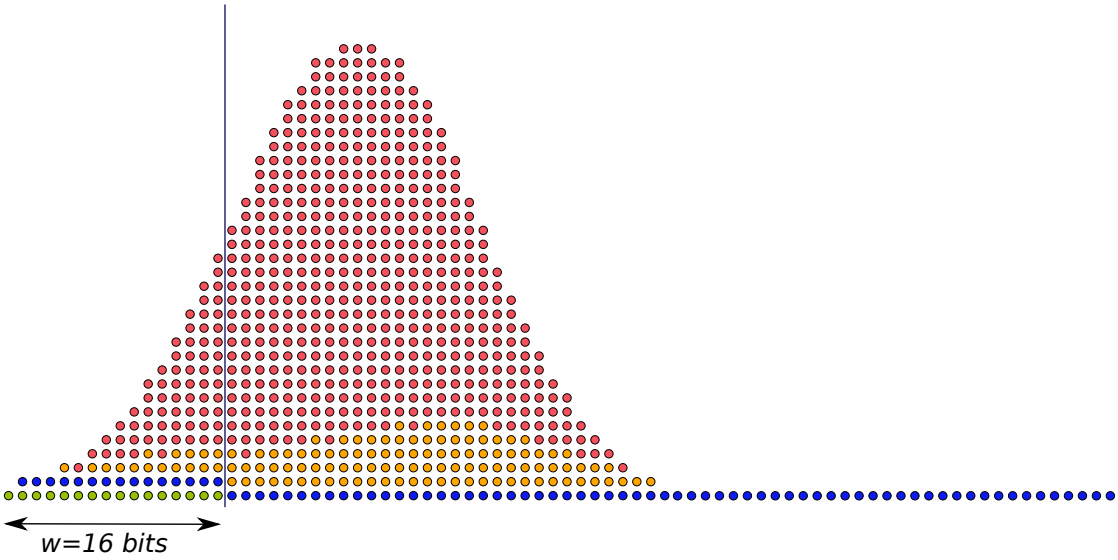
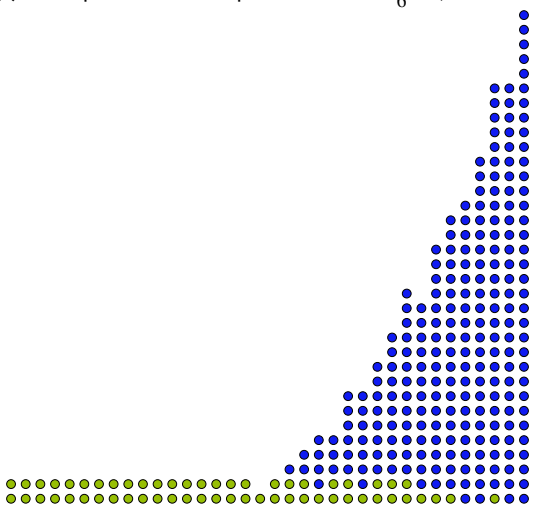


Figure 6.9: Example of bitheaps coming from FloPoCo's FixFIR operator, corresponding to a large FIR filter. The architecture of the filter is based on constant multiplications by the coefficients of the filter.



(a) Bitheap for 16-bit computation of $X - \frac{1}{6}X^3$, with $X \in [0, 1)$



(b) Bitheap for $X - \frac{1}{6}X^3$, as used in an actual sine computation, with $X \in [0, 2^{-6})$

Figure 6.10: Example of bitheaps corresponding to $X - \frac{1}{6}X^3$, presented in the beginning of Chapter 6. The upper part of the image presents the bitheap corresponding to an architecture that implements the previously presented optimizations. The lower part of Figure 6.10 shows how much of the architecture can be eliminated, in the context of a 16-bit sine/cosine operation computing just right. This optimization is detailed in Chapter 9.

- using only positive values for the MSB and LSB transforms the bitheap into an integer bitheap.
- the data structure used for the heap, is currently a vector of vectors, due to their dynamic nature and the need to access elements randomly.
- a bitheap can have multiple compression strategies. However, only one can be used for implementing the final architecture of the operator. The best compression strategy can be chosen according to the context.
- In general, having a separate VHDL code stream for the bitheap is useful when the code corresponding to the bitheap needs to be re-generated. One situation when this is useful is when several different compression strategies are applied on the same bitheap, with the scope of comparison. In such cases, only one compression strategy is chosen in the end, and the code corresponding to that compression is output.

Figures 6.7, 6.8, 6.9 and 6.10 show examples of bitheaps having different sizes and shapes. They correspond to various FloPoCo operators and have been automatically generated by FloPoCo, together with their corresponding operator. In the original images in the SVG format, hovering the mouse over a bit provides its timing information.

Aside from the **basic attributes** presented in the previous paragraphs of Section 6.1.1, a bitheap also has several **basic actions**:

- **add/subtract a bit to/from the bitheap**; this adds a new bit to the bitheap, at a given weight. The signal containing this bit must have already been created, prior to it being added to the bitheap.
- **remove a bit from the bitheap**; this removes an existing bit from the bitheap. The bit can either be a specific bit, or a bit in a specified order.
- **add/subtract a constant bit to/from the bitheap**; the bit is added at a given weight.
- **add/subtract a signal to/from the bitheap**; the signal is added at a given wight in the bitheap. This can either be specified by the designer, or it can be deduced from the data format of the signal (or both).
- **add/subtract a constant value to/from the bitheap**.
- **remove a signal from the bitheap**; this removes all the bits associated to a signal from the bitheap.
- **mark a bit/signal in the bitheap**; mark the selected bit (or bits associated to a signal) in order to change its status. More details are provided on the subject of bits and their properties in the following paragraphs of Section 6.1.1.
- **resize a bitheap**; a bitheap can be either extended or shrunk. This process can be achieved by eliminating some columns at a specified end of the bitheap (either MSB or LSB). This can also be specified as a data format.

- **merge bitheaps**; this operation represents the addition of the two values computed by the two bitheaps. Adding or multiplying two bitheaps together gives a bitheap, as well. This justifies the merge action for bitheaps. The compression of a bitheap is delegated to a compression strategy. A bitheap can have several compression strategies, with a default one being allocated for each created bitheap.
- **compress a bitheap**; this creates an architecture computing the sum of a bitheap.

The actions of adding or subtracting constant bits or signals to the bitheap brings up the matter of how these bits are managed. Constant bits can come from various sources, among which the most common are rounding and sign management. As they are known in advance, it would be wasteful to handle them in the same way as the bits which are the result of some computation. In order to handle the constant bits, FloPoCo's bitheap framework includes a multi-precision integer/fixed-point number which accumulates these bits. This is a technique coming from the literature on multipliers and multi-input adders [17][42]. However, typical implementations manage the constant bits by hand, which is a tedious and error-prone process. The implementation in FloPoCo handles them automatically.

Signed Numbers

Even if it is possible to create bitheaps that are based on working directly with positive or negative bits [60], the following advocates that the management of two's complement signed numbers comes at a low cost.

Sign extensions are managed using the constant vector. It is a multi-precision integer which accumulates the constant bits/numbers added to the bitheap.

Adding a signed signal to the bitheap requires its sign extension, when the MSB of the signal is less than that of the bitheap. The sign-extension can be done using the classical technique of Section 2.2.2, on page 14. This is all done at the cost of complementing one bit, as the constant is added to the constant vector. As are all the constants coming from sign extensions. Therefore, the cost of managing signed numbers is in the worst case a line of bits of length equal to $MSB\_bitheap - LSB\_bitheap + 1$, plus one complemented bit for each sign extension. This can turn out to be quite insignificant, in the grand scheme of things.

Subtraction of a signed number is handled in a similar manner. When subtracting a two's complement signed number:

- the identity $-X = \overline{X} + 1$ is used
- its complement is added to the bitheap, sign extended.

In addition, a constant 1 bit is added at the LSB weight of the signal.

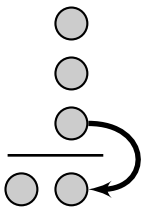


Figure 6.11: Dot diagram for a 3:2 compressor

Bits

In FloPoCo, a **Bit** can be seen as a wrapper around a **Signal**. Each bit has several attributes:

- the **signal** which it wraps.
- the RHS (right-hand side) assignment; this is used for implementing the underlying signal for the bit.
- a **UID** (unique identifier); this is used to identify the bits in the bitheap, as well as in the resulting VHDL code.
- a **name**; this is also used for identifying the bits in the bitheap, with easier access.

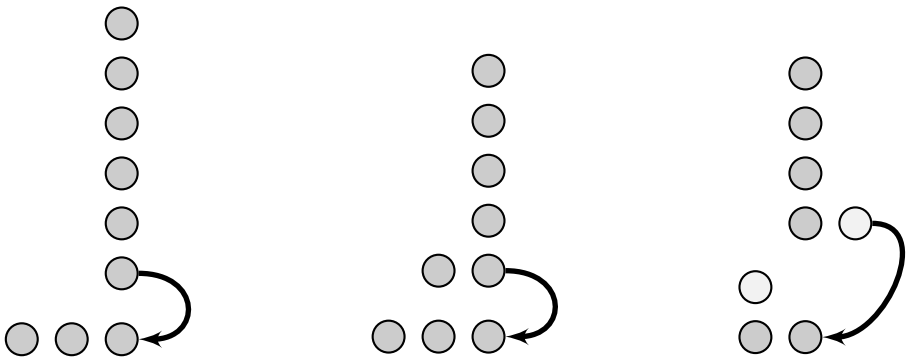
During compression, another attribute is set: the compressor, which compresses the bit (along with possibly other bits). This attribute is optional. Each bit in the bitheap is summed-up by a compressor, an exception to this rule being the bits which are on the last compression level, which make up the sum of the bitheap.

Compressors

The compressors are the basic building blocks for reducing the size of the bitheap during the compression process (which will be presented in detail in Section 6.2). The principle behind the compressors is that they take as inputs a certain number of bits and they return their sum, on a smaller number of bits.

The most classical compressor is the full adder, also known as a 3-to-2 (3:2) compressor, illustrated in Figure 6.11. It takes as inputs three bits of a given weight w, and returns their sum as two bits at weights w and $w + 1$. In general, a compressor takes as input at least two bits, which can have different weights (not necessarily contiguous), and returns a smaller number of bits, which are also distributed over different weights (not necessarily all distinct). Compressors should have a lower number of bits at the output than at the input, thus reducing the size of the bitheap. They should also have a lower height at the output than at the input (the term height is used here with the same connotation as for the bitheap, in Section 6.1.1).

Some example compressors are illustrated in Figure 6.12.



(a) A 6:3 compressor **(b)** A 1,5:3 compressor **(c)** A 4:2 compressor

Figure 6.12: Dot diagrams for some examples of compressors

- The 6-to-3 (6:3) compressor takes as input 6 bits of a certain weight w and returns their sum on 3 bits, of weights w, $w+1$ and $w+2$. It is presented in Figure 6.12a. It eliminates 3 bits (the difference between the 6 bits at the input and the 3 bits at the output) and reduces the height of the bitheap by 5 on the column of weight w.
- As another example, the 1-5-to-3 (1,5:3) compressor, also takes as input 6 bits 5 at weight w and 1 at weight $w+1$, as shown in Figure 6.12b. The sum of these 6 bits is at most 7, so it can be written on 3 bits: the compressor returns this sum at weights w, $w+1$ and $w+2$. It eliminates, as well, 3 bits, but only reduces the height of the bitheap by 4. On the other hand, it compresses bits from two columns of different weights, which avoids a possible carry propagation.
- The 4:2 compressor is an example of a compressor that has multiple bits of the same weight at the output, presented in Figure 6.12c. Its name is somewhat misleading. There are 4 bits at the input at weight w, but there is also an additional carry-in bit, which does not, however, influence the value of the 2 bits at the output. There are 2 bits at the output of the compressor, at weights w and $w+1$, but there is also an additional carry-out bit (also at weight $w+1$). Therefore, in the output, at the column of weight $w+1$, there is an unevaluated sum. The carry bits are usually intended for connecting the compressors in a chain.

The state of the art on compressors for FPGAs is mainly represented by [98], complemented by the work done for this chapter, which are all improved by [71]. The authors of [98] show how basic compressors can be built using the look-up tables on FPGAs, as well as the fast carry propagation logic. In doing so, however, the authors sacrifice the portability of the resulting VHDL code, as vendor-specific primitives are required for manipulating the underlying structure of the target device when building the compressors. The same is true for [71], which also has a target-specific approach.

The rest the section analyzes elementary compressors which can be implemented on generic FPGA targets that use 6-input look-up tables (LUT6). The FPGA model also assumes that the

LUT6 ca be split into two independent 5-input look-up tables (LUT5). The target FPGA model also assumes the presence of a fast-carry chain and of routing resources. In the following the notations of [42] and [98] are used for describing the compressors, referred to as **generalized counters** in the following[†]. For example, the full adder is denoted $GCNT(3:2)$. More generally, a generalized counter is denoted as $GCNT(k_{n-1},\ldots,k_1,k_0 : r_{m-1},\ldots,r_1,r_0)$, where n is the number of columns at the input (the difference between the smallest and the largest weight) and where m is the number of columns at the output.

In order to compare and discuss the performance of compressors, a few criteria need to be introduced:

- the **compression factor** ; denoted γ. It represents the quotient between the number of bits at the input and the number of bits at the output of the compressor. Using compressors with a hight γ reduces the height of the bitheap during the compression process.
- the **number of bits removed from the bitheap** ; denoted δ_{bits}. This is the difference between the number of inputs and the number of outputs. Using compressors with a high δ_{bits} reduces the size of the bitheap during the compression process.
- the **cost in 6-input look-up tables** ; it is denoted Σ_{LUT6}. It shows the total cost in terms of logic for implementing the compressor. Using compressors with a low Σ_{LUT6} results in a more efficient architecture.
- the **cost per removed bit** ; it is denoted η and can be computed as $\eta = \frac{\Sigma_{LUT6}}{\delta_{bits}}$. This metric can be used to show the area efficiency of the compressor.
- the **delay of a compressor** ; it is denoted τ and is expressed in terms of the delay of a 6-input look-up table (τ_L) and the delay of the fast-carry propagation of a bit (τ_{CP}). On modern FPGA devices $\tau_L \gg \tau_{CP}$, typically $\tau_L \approxeq 30 \cdot \tau_{CP}$.
- the virtual carry-propagation; it shows the number of columns which the compressor covers, thus avoiding the need for an adder for the carry propagation.

Table 6.1 illustrates theses metrics for some compressors. The top part of the table deals with basic compressors that can be created using portable VHDL code and which do not rely on target-specific primitives and manipulations for their implementation. The middle part of Table 6.1 deals with a selection of the best compressors presented in [98]. This is by no means an exhaustive analysis of the compressors of [98], and more can be found in the article such as details about their implementation. It should be noted that the authors of [98] only consider compressors with two columns at the input. The bottom part of the table analyses the cost and performance of adders as compressors.

[†]It should be noted that the counters described in [98] are referred to as **generalized parallel counters** . However, this notion is somewhat misleading, as the counters used the carry propagation in their implementation. The parallelism becomes evident only when multiple of them are used for compressing a bitheap, for example. Thus, the nomenclature **generalized counters** seems more appropriate.

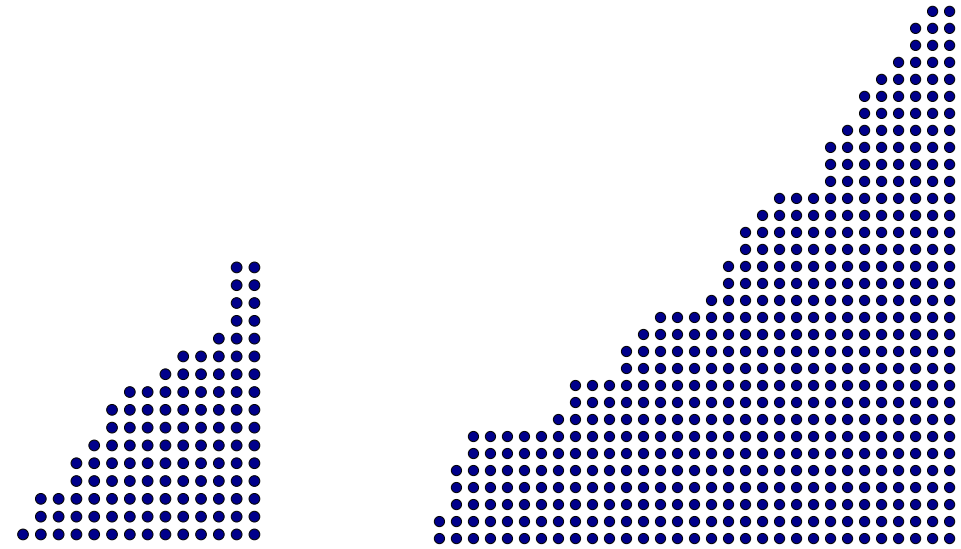
name	γ	δ_{bits}	Σ_{LUT6}	η	delay
Basic LUT-based compressors					
GCNT(3:2)	1.5	1	1	1	τ_L
GCNT(6:3)	2	3	3	1	τ_L
GCNT(1,5:3)	2	3	3	1	τ_L
GCNT(7:3)	2.33	4	4	1	$2\tau_L$
GCNT(1,4:3)	1.66	2	2	1	τ_L
GCNT(5:3)	1.66	2	2	1	τ_L
GCNT(4:3)	1.33	1	2	0.5	τ_L
GCNT(1,3:3)	1.33	1	2	0.5	τ_L
Arithmetic-based compressors of [98]					
GCNT(1,5:3)	2	3	2	1.5	$2\tau_L + 2\tau_{CP}$
GCNT(3,5:4)	2	4	3	1.33	$3\tau_L + 2\tau_{CP}$
GCNT(7:3)	2.33	4	3	1.33	$3\tau_L + 2\tau_{CP}$
Adders as compressors					
adder(n)	2	n	n	1	$\tau_L + n\tau_{CP}$
adder3(n)	$3\frac{n}{n+2}$	$2n-2$	$n+2$	$\frac{2n-2}{n+2}$	$\tau_L + n\tau_{CP}$
adder3(16)	2.66	30	18	1.66	$1.5\tau_L$

Table 6.1: Feature comparison of several elementary compressors

Table 6.1 can be used for choosing the desired set of compressors during the compression process. The choice is heavily influenced by the compression strategy used and by the trade-off the user is willing to make: trading a faster compression for a higher cost of the architecture. For example, the fastest architecture is obtained by ordering the compressors by γ. The lowest hardware cost is obtained by ordering the compressors by η.

Analyzing the bottom part of Table 6.1 shows that 3-input adders perform the best out of all the compressors in the table. They have the best γ and the best η. Their delay remains comparable to that of the other compressors in the table (between τ_L and $2 \cdot \tau_L$). However, this is true only for platforms that have support for 3-input addition. Altera, for example, allow additions of up to 10 bits and ternary adders can be easily specified using the + operator in VHDL . Xilinx, on the other hand, requires the use of specific primitives for implementing the ternary adders.

This is not the only limitation for using ternary adders. As remarked in [98], using generalized counters is more efficient for compressing a bitheap in situations when the addends need to be padded with zeros. In such cases resources are clearly wasted, if adders are used. Instead, Section 6.1.4 suggests the use of ternary adders of small sizes, which can be used efficiently for paving a bitheap during the compression process without the need for padding with zeros.



(a) Bitheap of width 16

(b) Bitheap of width 32

Figure 6.13: Example of two bitheaps, one summing 16 numbers of various lengths (at most 16 bits) and another one summing 32 numbers of various lengths (at most 32 bits)

Bitheap Compression

This section deals with one of the basic and most important actions on a bitheap, its compression. The **compression is the process of evaluating the sum of all the bits inside a bitheap** . The rest of the section deals with **timing-driven compression**, which makes the assumption that not all the bits in the bitheap are available for compression at the same time. This is unlike the classical approaches, where all bits to be compressed are available at the same time (*e.g.* in a multiplier): the architecture of the compression is built having timing as a constraint. Therefore, applying optimization algorithms, such as the retiming of Section 3.3, on a classical compression is not equivalent and does not suffice.

A bitheap compression is based on a chosen set generalized counters (be they look-up table- or adder-based). The GCNTs take as input a certain number of bits and output a smaller number of bits. Traditionally, the compression of a bitheap is done in successive stages, each taking input bits from the previous stage and feeding bits to the next stage, until the bitheap height is equal to 1. In each stage, the bitheap is covered with as many compressors as possible. All the compressors in one stage work in parallel. The inputs to a stage are either bits coming from the previous stage or bits which could not be compressed during the previous stages.

There are, thus, two main concerns regarding the compression of a bitheap: the choice of the set of compressors (dealt with in Section 6.1.4) and the compression strategy for assigning bits to compressors, addressed in the following.

The compression process is illustrated by compressing the bitheaps of Figure 6.13. The bitheap of Figure 6.13a corresponds to the sum of 16 numbers, of various sizes. The numbers are of widths of up to 16 bits. The bitheap of Figure 6.13a is a larger example, and corresponds to the sum of 32 numbers, again of various sizes. The numbers are of widths of up to 32 bits.

Compressing a bitheap is a difficult problem in itself: bitheap compression can be reduced to the tiling problem (covering a board of given dimensions with a given set of tiles), which is known to be NP-hard. As such, there are many aspects that influence the compression of a bitheap. They can be technological (as the choice of the compressors used and the way in which their characteristics influence the compression process) or technical (such as finding the optimal strategy). In addition, the designer might wish to optimize the compression for one of several possible objectives: minimal delay, minimum cost or a compromise in between. Therefore, most solutions in the literature take a greedy approach. They try to fit as many compressors in each stage and use as much as possible the most efficient compressors. This is the case for [98], as well as for the solution proposed in the following. In the rest of the section, the choice of compressors is first presented, followed by the definition of a compression stage, the presentation of the overall compression heuristic and the final addition stage.

Choice of compressors. The compression strategy starts by creating a list of compressors. They are ordered by cost per removed bit η, then compression factor γ, then number of bits removed δ_{bits} then delay τ. As it turns out, there aren't many trade-offs to be made in terms of the delay/area efficiency trade-off when ordering the list.

The next step is the actual compression of the bitheap, which is done in iterative stages. At each stage, the best applicable compressor is used for compression, as many times as possible. Applying a compressor has implications on two different levels. On a logical level (where the bitheap can be seen as a container for the bits), it consists of removing the bits that are consumed by the compressor and replacing them with the bits at the output of the compressor. On a hardware level, applying a compressor consists of first creating the hardware for the compressor, then connecting the set of bits to its inputs, and the bits at its output back to the bitheap.

A compression stage. The timing of the bits inside the bitheap is handled automatically by the pipelining framework, previously discussed in Chapter 4. Bits produced in one compression stage are not taken into consideration as inputs for the compressors in the same stage. This is so as to avoid as much as possible the creation of carry chains. In order to have the compression in several stages, the compression strategy chooses at the beginning of each stage the earliest compressible bit. This is the bit with the earliest arrival time and which is part of a column of bits that can be compressed by the current set of available compressors. All the bits that are considered for compression in the current

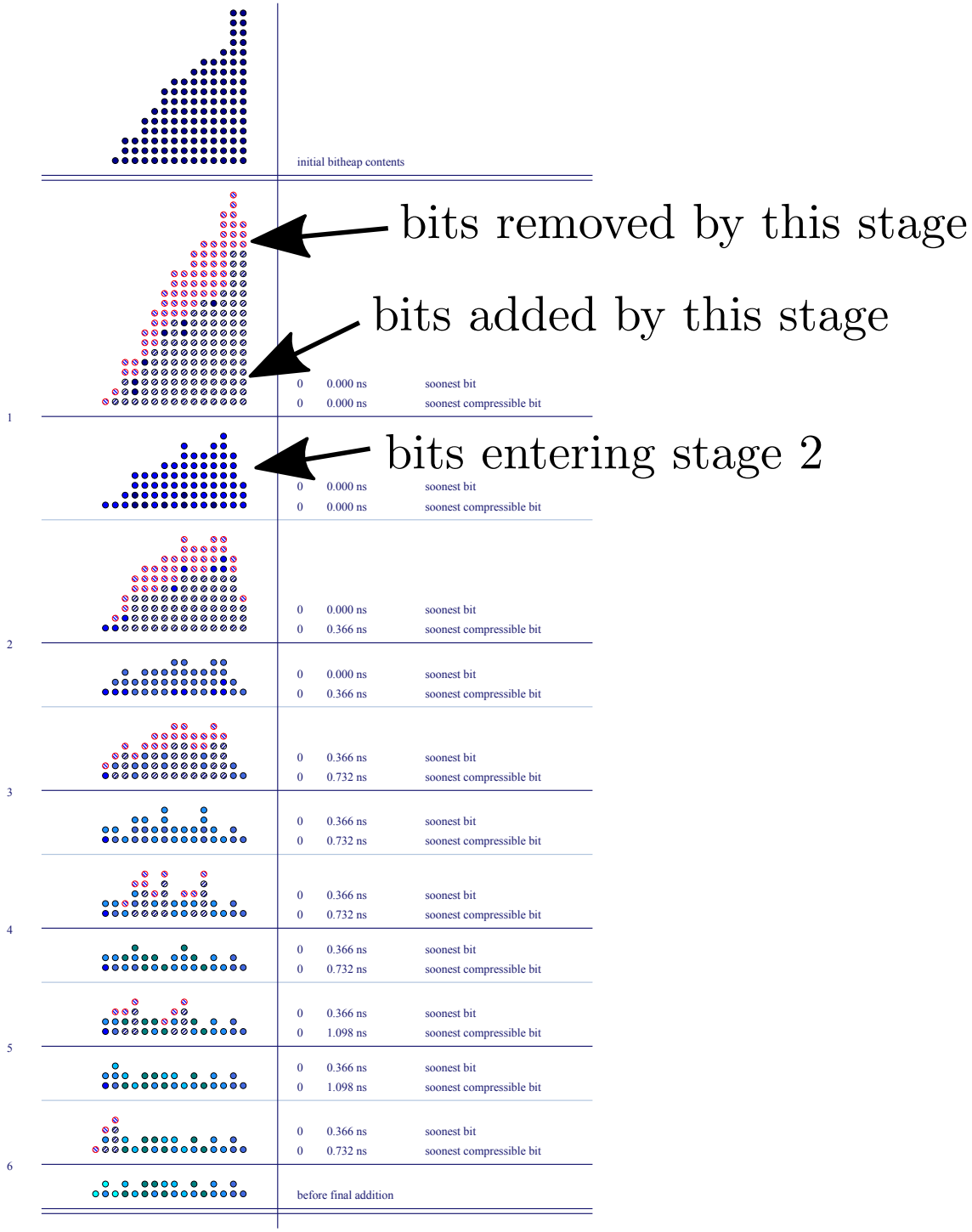


Figure 6.14: Compression stages for the bitheap of Figure 6.13a

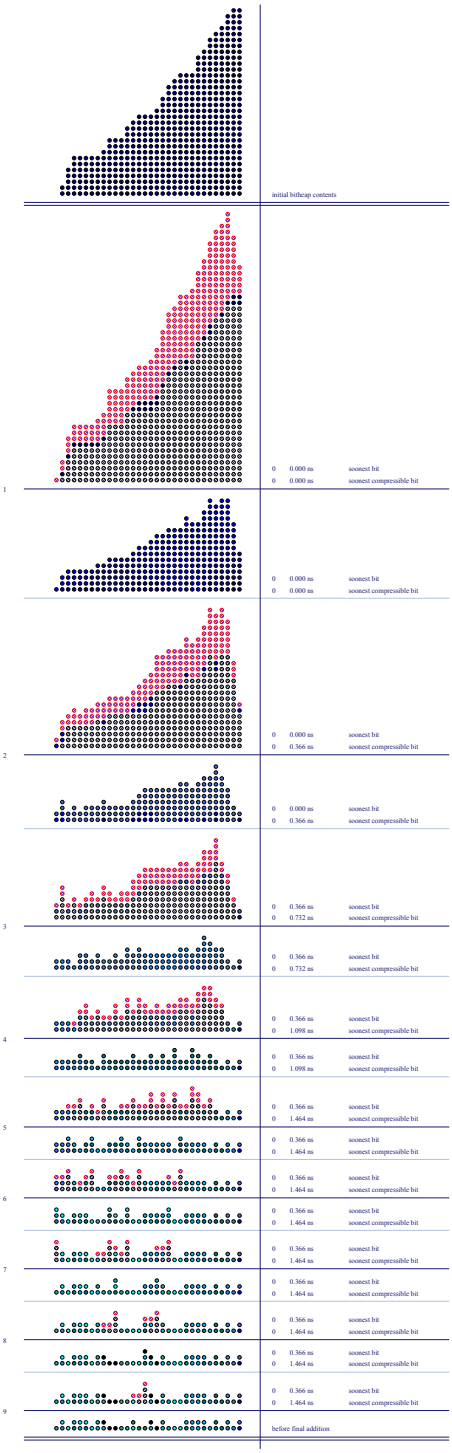


Figure 6.15: Compression stages for the bitheap of Figure 6.13b

stage must be within a given delay from this bit. This is done so as to have bits with comparable delays compressed by the same GCNT. The delay is initially set to that of a GCNT, and subsequently increased, if necessary.

The overall heuristic. In the first stages of the compression, the more efficient GCNTs are preferred. This is done with the hope that subsequently produced bits will allow the continued use of the efficient GCNTs. The less efficient GCNTs are only used in the subsequent stages, to avoid the increase of the delay of the compression. This is a heuristic decision, of course. The compression strategy used for the later stages is inspired by the classical approach of [30]. The idea is to first evaluate the minimum achievable delay for the compression, and subsequently use the minimal number of compressors to achieve it. Again, this is a heuristic, which can help speed-up the last compression stages.

The final addition. The GCNT-based compression stops as soon as the height of the bitheap is equal to 3. The last three rows of bits of the bitheap can be dealt with in two ways. The first is a ternary adder, on target platforms that support it. When not supported, two high-speed adders (like those of [95]) can be used. In some situations, the compression of the last three rows can be achieved by first using a row of $GCNT(3:2)$, followed by a fast adder. This last addition stage is sometimes referred to in literature as a vector merge adder (VMA) [71]. Of course, only columns with heights greater than 1 are included in the final additions, the rest being already compressed, and can be sent directly to the final result of the compression.

Figures 6.14 and 6.15 show the initial bitheap at the top, before the start of the compression stages. The next image in the figures shows the state of the bitheap after the first compression stage. In it, the bits that have been compressed have the contours in a color corresponding to the time at which they are created, and the interior hashed diagonally, in the same color. This color is that of the bits in the previous image. The bits just added to the bitheap are illustrated with red contours, and the interior hashed diagonally, in a different color corresponding to a different moment in the circuit time (which is more visible in the subsequent image). The third image in the figures represents the bitheap once the compressed bits are removed from the bitheap. It contains the newly created bits, plus the bits that could not be compressed in the previous stages. The compression stages are marked to the left of the thicker horizontal lines. The rest of the images in the two figures present the succession of compression stages. Figure 6.16 shows a zoom-in on the top part of Figure 6.14, with the three images explained above.

These images are generated automatically by the bitheap framework in FloPoCo. They are in SVG format, and hovering the mouse over a bit provides detailed timing information about the bit (more precisely about the signal it encapsulates).

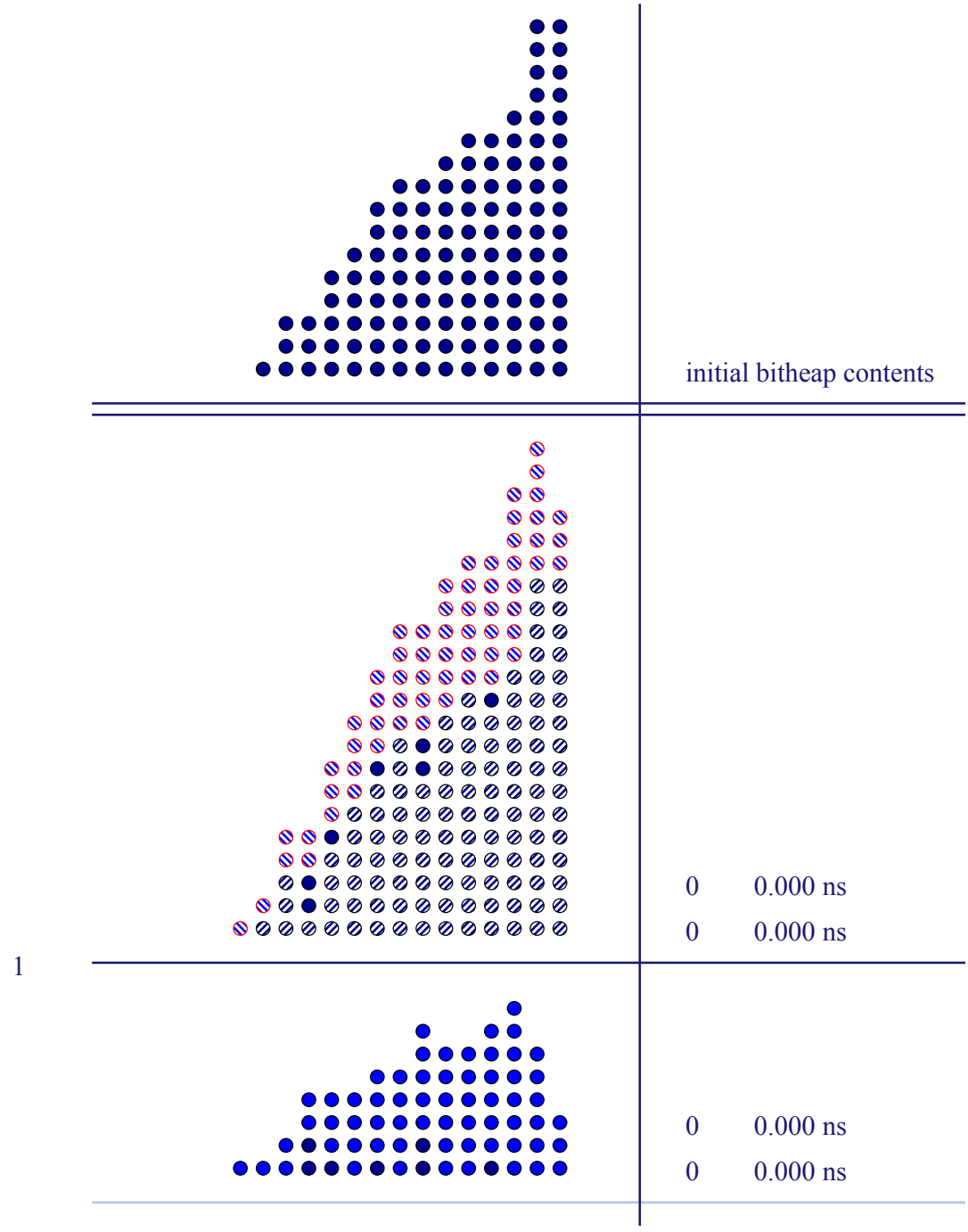


Figure 6.16: Zoom in on the top part of Figure 6.14

Results on logic-based integer multipliers

Section 6.3 presents and discusses some results on the implementation of arithmetic operators using the bitheap framework in FloPoCo. These results are by no means exhaustive, and are meant to show

Multiplier Specification	Cost of Partial Products	Cost of Compression (parallel counters)	(ternary adders)
24x24	384	505	283
32x32 truncated	510	379	287

Table 6.2: ALM costs for logic-based multipliers (parallel counter versus ternary adders) on Altera Stratix IV

Approach	Performance cycles @ frequency	LUTs	Registers	DSPs
precision = 23 bits				
Classical	1 cycles @ 179 MHz	798	135	0
Bitheap	1 cycles @ 255 MHz	771	78	0
precision = 32 bits				
Classical	1 cycles @ 168 MHz	1571	168	0
Bitheap	2 cycles @ 249 MHz	1469	114	0
precision = 53 bits				
Classical	1 cycles @ 137 MHz	4259	368	0
Bitheap	4 cycles @ 244 MHz	4002	497	0

Table 6.3: Integer Multiplications on Xilinx Virtex-6

some of the capabilities of the bitheap framework. More data is presented in the following sections, which also discuss the implementation details of their corresponding operators.

Table 6.2 illustrates the points made in Section 6.1.4 and in Table 6.1 on the efficiency of the ternary adders used as compressors. The table presents results for the implementation of logic-only based multipliers (based on the look-up table resources, not using the multiplier blocks in the target device). For both sizes of the multiplier it is obvious that the ternary adders are more effective for the compression. The data is limited to Altera devices, as Xilinx devices do not natively ternary adders. There is a performance penalty of about 10% (as compared to classical ripple-carry two-input adders) when implementing ternary adders on Altera devices [71]. Implementing ternary adders on Xilinx devices requires the use of device-specific primitives, which goes against FloPoCo's wish to maintain the resulting VHDL code portable. Even when using specific implementations, ternary adders on Xilinx devices have a performance penalty of about 50% [71].

Table 6.3 shows a comparison of two different implementations of integer multipliers on Xilinx devices. The multiplications have their inputs on 23, 32 and 53 bits, respectively, and compute the output on its full precision. The first one, denoted *Bitheap* uses a bitheap for implementation, as its name suggests. The second one, denoted *Classical*, is implemented using the $*$ VHDL multiplica-

tion operator, and relies on the synthesis tools to be replaced with the vendor's implementation of choice. It can be seen that the bitheap-based implementations consistently result in better LUT and register usages, all the while running at a higher frequency. The bitheaps corresponding to the three multipliers, as well as their compressions, are illustrated in Figures 6.17, 6.18 and 6.19.

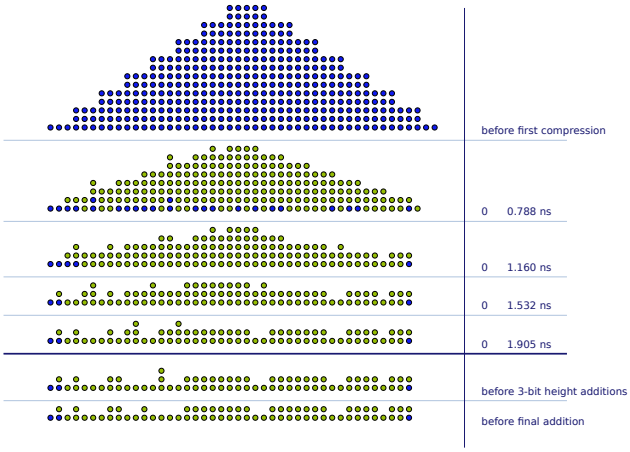


Figure 6.17: The bitheap of a 23-bit IntMultiplier and its compression

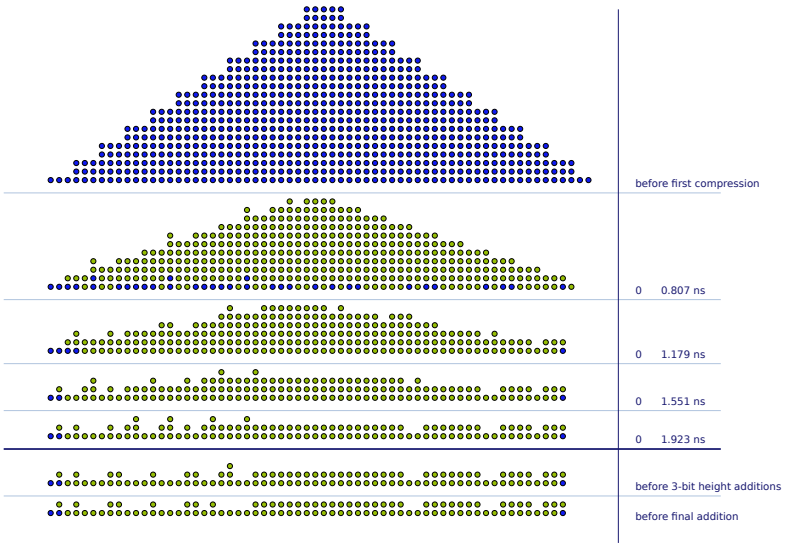


Figure 6.18: The bitheap of a 32-bit IntMultiplier and its compression

The example of Table 6.3 also show that the multipliers based on the bitheap framework offer the opportunity of a custom pipeline. More details on multiplications are presented in Chapter 7.

The last example operator presented is the product of two complex numbers. An illustration of the operator is provided in Figure 6.20. Figure 6.20a shows a block diagram of the operator, while

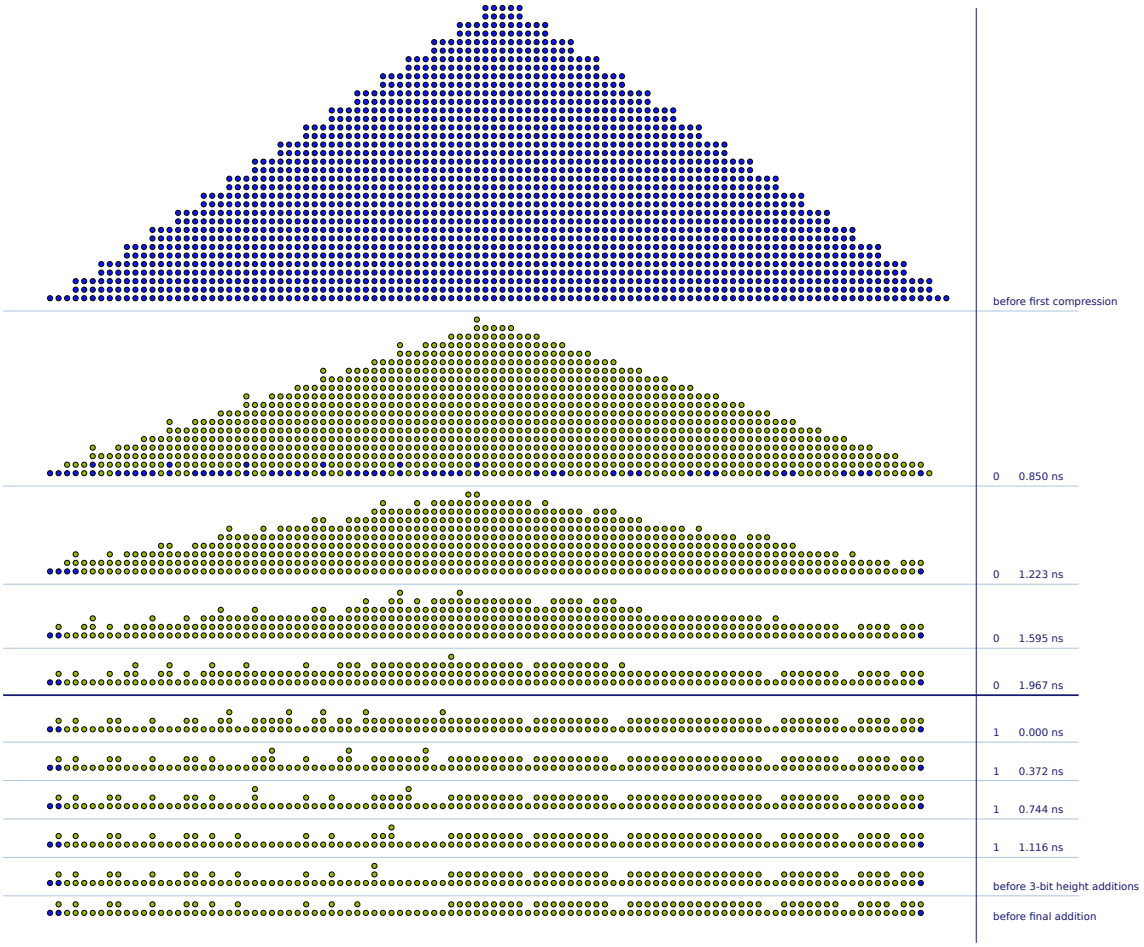


Figure 6.19: The bitheap of a 53-bit IntMultiplier and its compression

Figures 6.20b and 6.20c show the bitheaps corresponding to the complex product with inputs on 12 and 32 bits respectively. The architecture of the operator makes use of two bitheaps for computing the product of two complex numbers: one corresponding to the real part of the result, and the other corresponding to the imaginary part. The comparisons target a Xilinx Virtex6 device.

Table 6.4 shows some synthesis results for complex products of varying sizes. The size of the inputs to the multiplier are varied, from 12 to 32 bits, and the outputs are computed faithfully to the input precision. As in the case of Table 6.3, a comparison is made between the method based on bitheaps, denoted *Bitheap* in the table, and a classical implementation using the $+$ and $*$ operators. The latter method leaves the implementation of the addition and multiplication operators to the synthesis tools. For each input precision two comparisons are made: one using the DSP blocks and the other not using them.

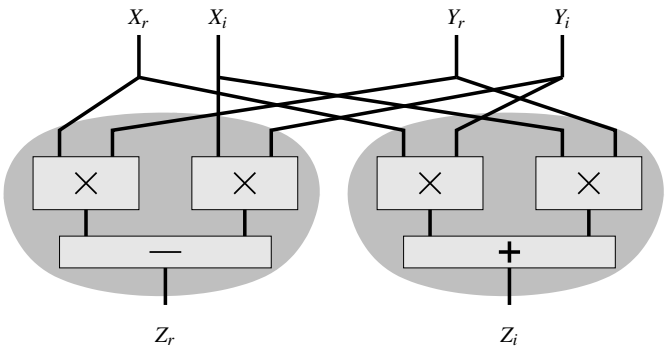
As with the findings of Table 6.3, for the logic-only architectures, the bitheap-based solutions

Approach	Performance cycles @ frequency	LUTs	Registers	DSPs
precision = 12 bits				
Mult+Add	1 cycles @ 238 MHz	48	16	4
Mult+Add	1 cycles @ 192 MHz	918	64	0
Bitheap	2 cycles @ 245 MHz	84	104	4
Bitheap	1 cycles @ 270 MHz	598	58	0
precision = 16 bits				
Mult+Add	1 cycles @ 238 MHz	63	21	4
Mult+Add	1 cycles @ 187 MHz	1623	84	0
Bitheap	2 cycles @ 245 MHz	131	142	4
Bitheap	1 cycles @ 258 MHz	1180	82	0
precision = 24 bits				
Mult+Add	4 cycles @ 251 MHz	469	611	8
Mult+Add	1 cycles @ 156 MHz	3630	120	0
Bitheap	4 cycles @ 256 MHz	320	214	8
Bitheap	1 cycles @ 257 MHz	2223	138	0
precision = 32 bits				
Mult+Add	2 cycles @ 273 MHz	2048	960	8
Mult+Add	1 cycles @ 148 MHz	6414	152	0
Bitheap	2 cycles @ 234 MHz	1816	328	8
Bitheap	1 cycles @ 185 MHz	3962	138	0
precision = 64 bits				
Mult+Add	5 cycles @ 250 MHz	4048	4584	44
Mult+Add	1 cycles @ 114 MHz	25297	284	0
Bitheap	5 cycles @ 250 MHz	1144	1028	44
Bitheap	1 cycles @ 136 MHz	14970	334	0

Table 6.4: Faithful complex products on Xilinx Virtex-6

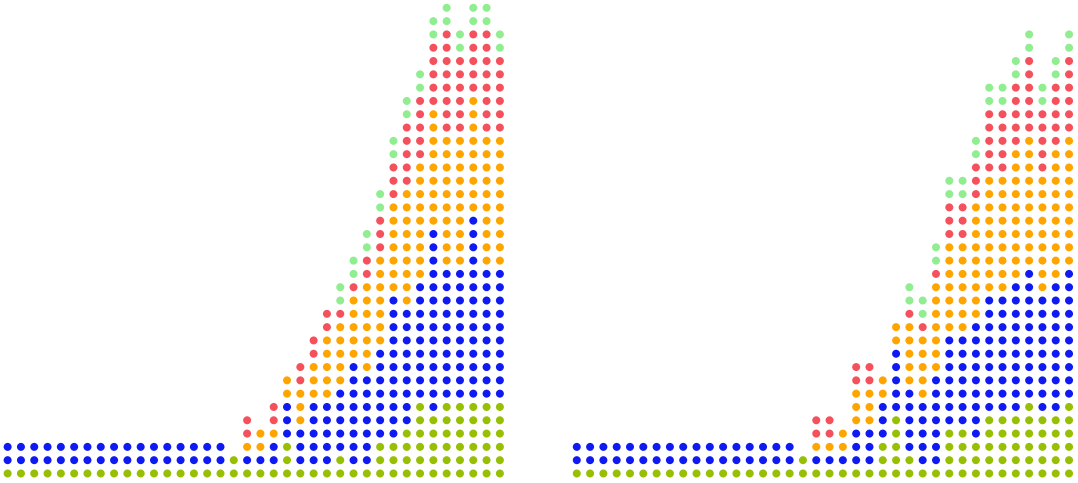
outperform their counterparts at all testes precisions. The differences in LUT consumptions increase with the size of the operators, in favor of the bitheap-based method. The same is true for frequency, where a better frequency is always achieved.

Tables 6.3 and 6.4 show that the bitheap-based solutions can be more efficient in terms of required resources and maximum frequency than their classical equivalent operators. These gains become more obvious as the operators grow larger, where the bitheaps have more opportunities for low-level optimizations.



(a) Operator block diagram

(b) The two bit heaps for 12-bit faithful results.



(c) The two bit heaps for 32-bit faithful results, using one DSP block for each multiplication.

Figure 6.20: Complex multiplication as two bit heaps.
Different colors in the bit heap indicate bits arriving at different instants.

Concluding Remarks on Bitheaps, Compressors and Compression Strategies

Section 6.4 deals with some comments on the work presented in Sections 6.1 through 6.3, and their corresponding subsections.

A very positive remark that can be made concerning the the work presented throughout this chapter on bitheaps is that it has restarted and motivated research on bitheap compressions.

The discussions and works of Section 6.1.4 have been further improved by [70] and even further by [71]. These works are subsequent to those that are part of FloPoCo's bitheap framework. They are specific to Xilinx FPGAs and are based on exploring all the available resources inside a slice. They do not limit themselves to using the LUTs, and manipulate all te other resources such as the carry chain or the multiplexers between the LUTs and the CLBs. The authors provide carry-based GCNTs that have high efficiency, which they measure as the number of bits per LUT (equivalent to $\frac{1}{\eta}$ using the notations of Section 6.1.4). The best GCNTs of Section 6.1.4 have an efficiency of 1, while the ones of [70] have efficiencies of 1.33 and 1.5. In [71] the authors propose GCNTs with efficiencies of 1.5 and even 1.75. The latter seems to be the most efficient that can be achieved, when using up to 4 LUTs. For reference, the ternary adder has an efficiency (with the connotation of [71]) of $2 - \frac{2}{k}$, where k is the number of columns in the adder.

A second remark can be made concerning the compression strategy of Section 6.2. The compression strategy chosen for FloPoCo's bitheap framework is a greedy one. The authors of [80] and [71] present optimal strategies, which, again, are subsequent to FloPoCo's solution. The solution of [71] seems to be the state of the art in bitheap compressions, and is implemented using FloPoCo. It is based on modeling the compression as an optimization problem, which can be solved using integer linear programming techniques. Their method results in around a 20% improvement in terms of the LUTs required and 6% in terms of the maximum running frequency. However, the tests they use are relatively small (with a maximum of $\approx 200 - 250$ LUTs needed for the designs). There is also a remark to be made concerning the designs tested, which are themselves particular, being sums of integers which create bitheaps with rectangular shapes. Multipliers, for example, result in triangular shapes for the bitheaps, which may affect the compression process. The solution proposed in [71] requires more registers to obtain similar maximum running frequencies as the bitheap framework in FloPoCo. This is due to an idea which they promote, that the compressors should take advantage of the registers which follow the LUTs. This means that more registers are used, but at the same time the umber of slices used is lower.

Another point to be made is relative to carry propagation inside the bitheap compression. As a matter of fact another criteria could be introduced for the compressors, in this respect: the virtual carry-propagation. It shows the number of columns which the compressor covers, thus avoiding the need for an adder to perform a carry propagation. As a general rule of thumb, carry-propagation inside a compression is to be avoided, as it has a negative impact on the maximum achievable frequency. This phenomenon can sometimes be observed for greedy compression strategies. Take, for example, the last compression stages for the 53x53-bit integer multiplier in Figure 6.19. However, this is an aspect which is difficult to model, in both the compressors themselves, as well as in the compression

strategy. The most efficient compressors of [71] also have a high virtual carry propagation. It might also be that in optimal compression strategies this factor is compensated for by the compression strategy itself. In greedy strategies, like the one presented in this chapter, this is not the case, on the other hand.

7 Multiplication

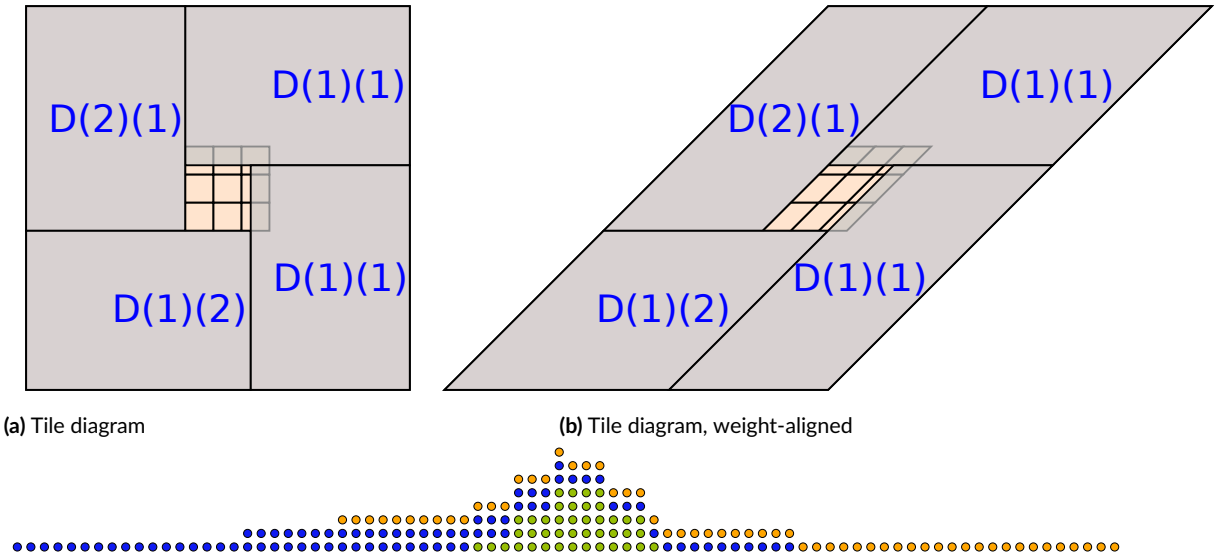
Multiplication is one of the basic building blocks inside an arithmetic operator generator. The performance and efficiency of the generator is tightly connected to those of the basic operations, such as the addition, multiplication or tabulation. Therefore, it is of the utmost importance to have the best possible implementation for the multiplication. Any improvement here translates to gains in any existing operator, as well as future operators based on multiplications.

It should also be pointed that there is a strong connection between the multiplication operation and the bitheap framework of Chapter 6. Indeed, this should come as no surprise, as the bitheaps (more specifically **compressor trees**) come from the fast multiplier and multi-input addition literature. As such, the improvement of large and fast multiplications for FPGAs are one of the original motivations for the bitheap framework.

FPGAs and Their Support for Multiplications

LUT-based Multipliers. Most legacy and current FPGAs are based on look-up tables. Therefore, the partial products can be computed using the look-up tables, be they 4-input LUTs on older devices, or 6-input LUTs on newer devices. This means that an $n \times n$ product can be computed using n^2 LUTs, hence using only the logic resources on the device. In addition, the fast carry-ripple additions can be used to accelerate the multiplications, at the cost of 1 LUT per bit of addition.

On the current generation devices, the 6-input LUTs can be used for implementing 3×3-bit multiplications. These can be tabulated into 6 LUTs, instead of being accumulated in 9 LUTs. This is at the same time more resource-efficient and faster, as it only incurs the delay of one look-up table.



(a) Tile diagram

(b) Tile diagram, weight-aligned

(c) The bitheap

Figure 7.1: A 41×41-bit to 82-bit integer multiplier for Xilinx Virtex 6 devices.

It is also the most efficient way of implementing the partial products for a bitheap-based logic-only multiplier. This idea was first introduced in [16].

DSP Blocks. Aside from LUTs, FPGAs provide direct support for multiplications through the embedded multiplier blocks (DSPs). On Xilinx devices these support signed multiplications of up to 25×18-bits. On Altera devices they support 36×36-bit multiplications, which can be split into multiple 18×18, 16×16, 12×12 or 9×9-bits ones. The number of blocks and the modes into which the multiplications can be split depends on the device itself and on the family of devices.

Figure 7.1 shows a representation of a 41×41-bit multiplier, with the output on its full precision, together with its corresponding bitheap, for a Xilinx Virtex 6 device. The large rectangles represent 25×18-bit DSP blocks and the small rectangles correspond to 3×3-bit LUT-based multipliers.

Multipliers, Tiles and Bitheaps

Section 6.3 used for comparison implementations of large multipliers where the architecture was left up to the synthesis tools, where the user has limited control. This solution could be improved by forcing the synthesis tools towards the desired solution, by splitting large multiplications into several smaller ones.

A multiplication can be split into smaller ones, tailored according to the available resources of

the target device: the logic-only multipliers and the embedded multipliers. Once decomposed, the multiplication can be seen as a sum, and is a perfect match for implementation using a bitheap. Take, for example, one of the possible implementations of the product of two 41-bit numbers X and Y:

$$\begin{aligned} XY = &(X_{0:16} \cdot Y_{0:23} + 2^{17} \cdot X_{17:40} \cdot Y_{0:16}) \\ &+ 2^{23}(X_{0:23} \cdot Y_{24:40} + 2^{17} \cdot X_{24:40} \cdot Y_{17:40}) \\ &+ 2^{34} X_{17:23} \cdot Y_{17:23} \end{aligned} \tag{7.1}$$

where $X_{n:m}$ represents the range of bits of variable X from index n to index m. This decomposition is also illustrated in Figure 7.1.

In the decomposition of Equation 7.1, every $X_{a:b} \cdot Y_{c:d}$ partial product except the 5×5-bit one on the last line is mapped to multiplier blocks (which are the large blocks in Figure 7.1). The sub-product on the last line is implemented using logic-based multipliers (and is shown as the small blocks in the middle of Figure 7.1).

The example of Equation 7.1 also shows how another feature of the target FPGA devices can be exploited. These are the internal adders contained in the multiplier blocks, which can be used for summing-up the results of several multipliers. Such groups of multiplier blocks are referred to as **supertiles** in [36] and in [100]. In Equation 7.1, the terms surrounded by parentheses on the first and second line represent supertiles.

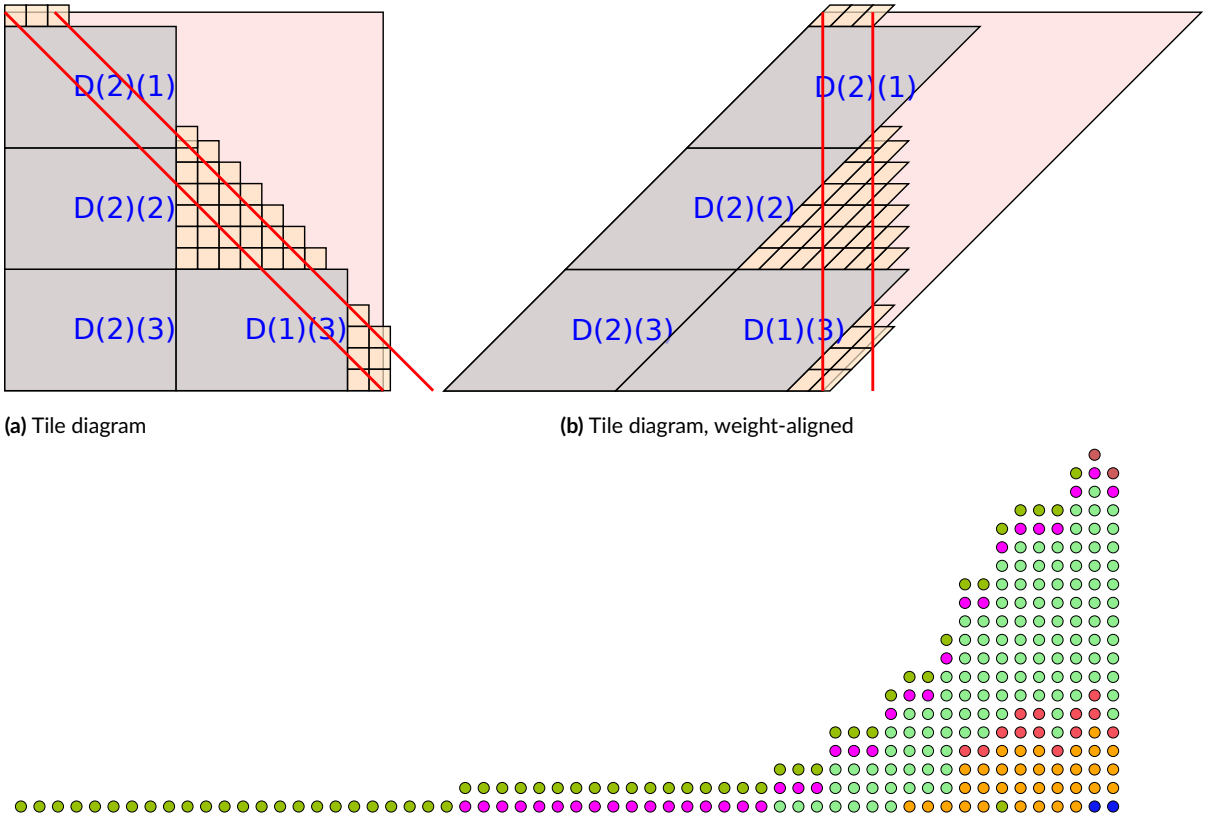
The bitheap-based multiplier generator is versatile, and offers customization opportunities in terms of target device, size, truncated/full result, signedness etc.. Figures 7.1, 7.2 and 7.3 are all generated together with the architectures. A proof of its versatility is represented by Figures 7.2 and 7.3, which are both representations of 53×53-bit multipliers, outputting their results faithfully to 53 bits. While the specification is similar, the two results are very different, as the multiplier blocks have different dimensions on the two target platforms.

The red line on the left, in both Figure 7.2 and 7.3, represents the truncation line; only the bits to its left should make it to the final output of the multiplier. However, in order to ensure the faithful result, some extra guard bits need to be used for the computations, into which errors can accumulate before the final rounding. This is the red line to the right, in the two figures.

Also worth mentioning is the fact that in Figure 7.2 there is a potential supertile grouping together three DSP blocks. In Figure 7.3, as well, two supertiles of size two can be formed, as described in [100], out of the two 18-bit, and 9-bit respectively, multiplier blocks at the top left and bottom right.

There is an equivalence between splitting the product $X \cdot Y$ in Equation 7.1 and the process of covering a rectangle representing $X \cdot Y$ with tiles corresponding to sub-multipliers in Figure 7.1.

Of course, none of the two concepts for implementing multipliers is new, and their sources of in-



(a) Tile diagram

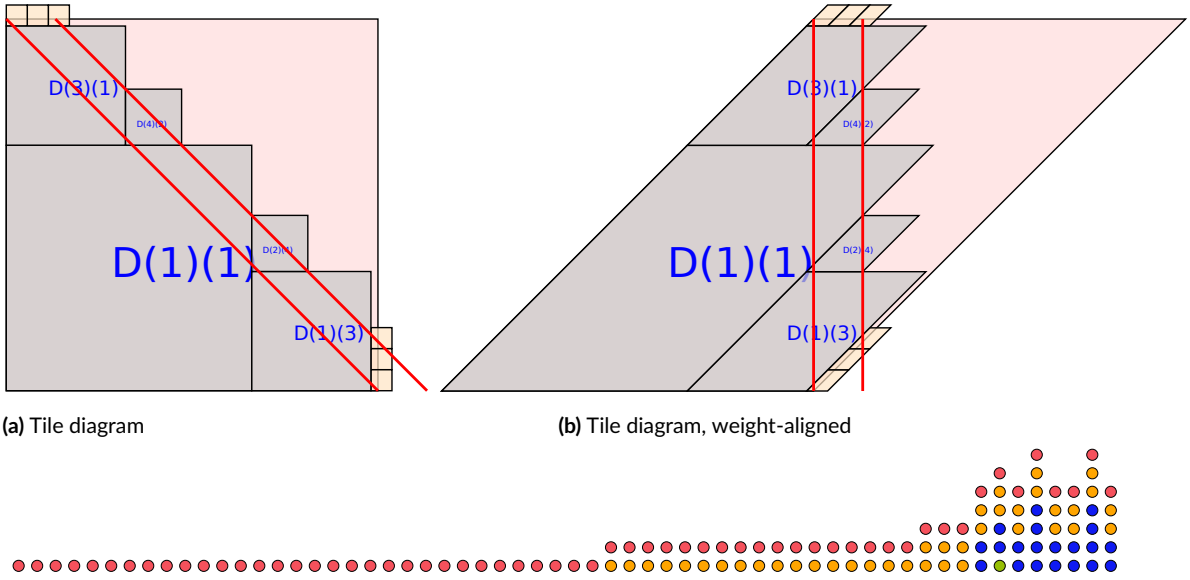
(b) Tile diagram, weight-aligned

(c) The bitheap

Figure 7.2: A 53×53-bit to 53-bit truncated multiplier for Xilinx Virtex 6 devices.

spiration can be traced throughout the literature. The association of the two concepts is introduced in [36] and further refined by [16]. Splitting Equation 7.1 is reminiscent of works such as [65] and [88], to mention just a few, from which [36] and [16] draw their inspiration.

The process of covering a board representing the multiplier (with the two multiplicands on the horizontal and vertical axes, respectively) is referred to in [16] as **tiling** . The tiles used there are either the embedded multipliers, or logic-only multipliers. The first to have associated tiling to the implementation of multipliers seem to be [36]. They are, however, far from being to be the first to study the tiling problem, which is very popular and has attracted much attention in various fields from mathematics/computer science (geometry, combinatorics, complexity theory) to industrial processes. A very comprehensive survey on tiling is presented in [118], where the problem is presented as a *packing* and *cutting* problem. The works surveyed there span over five decades and various fields, ranging from methods for solving the problem, to the analysis of its complexity and some of its possible applications.



(a) Tile diagram

(b) Tile diagram, weight-aligned

(c) The bitheap

Figure 7.3: A 53×53-bit to 53-bit truncated multiplier for Altera StratixV devices.

In terms of the complexity of the problem, it has been shown that variations of the tiling problem are NP-complete. In [74] and then in [75], tiling is even proposed as the problem to which to reduce other ones, when trying to prove their NP-completeness. In their context, the tiling problem consists of covering a board with a set of authorized tiles that have letters on each corner. There are rules as to which tiles can be placed on next to the other. Although less constrained, the tiling of a multiplier using square multiplier blocks can be reduced to the problem of [74]. In [122] the authors use a similar variation of the tiling problem for analyzing its complexity, while reaching the same conclusions. In their formulation, the tiles have colors along the edges. In [89] it is shown that tilings using trominos and tetrominos are also NP-complete. This is relevant, as the groups of DSPs, denoted **supertiles** in [16], can be modeled as such.

Looking at the tiling problem as a *partitioning problem*, it is shown in [18] that the problem is, again, NP-complete. Their context is to partition the unit square into a set of rectangles of given dimensions. The tiling of a multiplier is an even more difficult problem, as the number of tiles is not known in advance.

It is also curious to see how some of the designs for multipliers uncovered in [16] have shown up throughout literature, in completely different contexts and domains. Take, for example, the 41×41-bit multiplier of Equation 7.1 and Figure 7.1. In mathematics, this type of tiling is part of a class of shapes called simple tilings [29], which are tilings where no two adjacent tiles form a rectangle. The

particular pattern of the multiplier blocks of Figure 7.1 is called a pinwheel in [93]. Larger shapes of the same style can be created, for tiling larger surfaces. The same kind of shape as the pinwheel can be found using a completely different approach, such as the partitioning. In [32] a *double order* is used for finding the pinwheel design. It consists of two partial orderings inside the set that represents the surface to be tiled: the 'below' and the 'to the left of' relationships.

Bitheaps vs. Components

As the multiplier implementations of [16] predate the bitheap framework in FloPoCo, they relied on three main components: the embedded multiplier blocks, logic-based multipliers and multi-input adders. This is inefficient from several reasons:

- multiple individual compressions, one for each logic-based multiplier, instead of unique one.
- underutilization of some of the resources available; the adders in the first DSP blocks are left unused.
- decreased performance of the operator; some of the lower weight bits are delayed, when they could be sent directly to the next stage of the computations.
- multiple local optimizations, instead of a single global one; for example, combining the compression of the partial products and their summation.
- it makes the designer's job more difficult due to the need for manual synchronization and alignment of the bits coming in and out of the partial products.

The advantages of using bitheaps over individual components are multiplied when the multipliers are part of a larger design. In such cases, there are even more optimization opportunities inside a bitheap. Take as an example a sum of products (for example when computing a digital filter) or a complex product, like the one of Figure 6.20 on page 96.

One possible optimization concerns the supertiles. For large designs, it is possible to group multiplier blocks coming from different multipliers in order to obtain supertiles. For example the product of two complex numbers, where there are two multiplications for computing the real and imaginary parts, respectively. Indeed, Altera's multiplier blocks have support for these kind of operations, being able to compute $a \times b \pm c \times d$ with operands of up to 18 bits. It should be noted that grouping together multiplier blocks in supertiles can be used as well for larger multipliers, which are split into several sub-products. The sub-products corresponding to the same $a \times b \pm c \times d$ operation can be paired in the same supertile.

Another opportunity for optimization is to fully make use of the resources inside the multipliers, such as the adders. The adders can be used for compressing bits from the bitheap, which are ready for

compression timing-wise. These bits' origins are not important; they can even come from a different operation than the multiplication at hand. For example, for multiplier blocks that are not part of supertiles, the adders are unused. The same is true for the first multiplier block in a supertile. Xilinx's multiplier blocks can accept an external input, while Altera's do not support such a thing.

There are a couple of observations to be made concerning the management of multipliers and multiplier blocks. First, When used as a subcomponent of a larger operator, a multiplier is not an actual component itself. It is a virtual component; it does not correspond to a VHDL entity, in the usual way an operator does, it is part of one. For example, in the case of the complex product of Figure 6.20 the multipliers throw bits to one of the two shared bitheaps. The same can be said for an addition/subtraction. Relating again to the complex product, the adder and the subtracter do not have a corresponding VHDL entity either. Instead, they are part of the compressions of the two bitheaps.

If multiple operators can share the same bitheap, the converse is true as well. An operator can have multiple bitheaps.

The second observation concerns the embedded multiplier blocks. An efficient way of managing them in a bitheap is to:

- treat the multiplier block as unevaluated multiplications
- consider the adders in the multiplier blocks as compressors, when otherwise unused
- consider the supertiles as a step of the compression process

Multiplying Signed Numbers

The example of the complex multiplication requires subtracting one of the products, which brings up another issue: managing the product of signed numbers. To start, the product of two two's complement numbers X and Y of m and n bits, respectively, is to be added to the bitheap. The two numbers can be written as:

$$\begin{aligned} X &= -2^{m-1} \cdot x_{m-1} + \sum_{i=0}^{m-2} 2^i \cdot x_i \\ Y &= -2^{n-1} \cdot y_{n-1} + \sum_{j=0}^{n-2} 2^j \cdot y_j \end{aligned} \tag{7.2}$$

Therefore, their product can be written as:

$$\begin{aligned} X \cdot Y = {} & 2^{m+n-2} \cdot x_{m-1} \cdot y_{n-1} \\ & - 2^{m-1} \cdot x_{m-1} + \sum_{j=0}^{n-2} 2^j \cdot y_j \\ & - 2^{n-1} \cdot y_{n-1} + \sum_{i=0}^{m-2} 2^i \cdot x_i \\ & + \left(\sum_{i=0}^{m-2} 2^i \cdot x_i \right) \cdot \left(\sum_{j=0}^{n-2} 2^j \cdot y_j \right) \end{aligned} \tag{7.3}$$

Equations 7.2 and 7.3 are the signed equivalents of Equations 6.1 and 7.1, respectively, from the beginning of Chapter 6. Figure 7.4 shows an illustration of Equation 7.3.

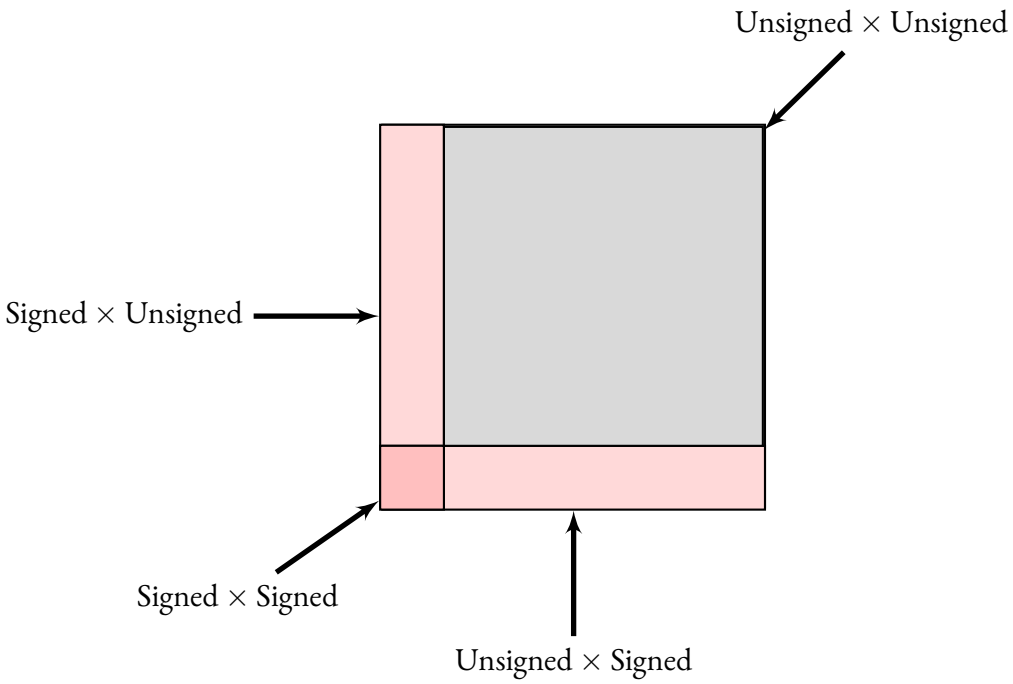


Figure 7.4: Multiplication of two signed numbers X and Y

The $X \times Y$ product of Equation 7.3 has its MSB at weight $m + n - 1$, even if this term is not present directly in the formulation. The bit at this weight is the carry-out of the sum. The way in which the sum can be handled is to handle each term separately. The first and last term are unsigned numbers, so they can just be added to the bitheap. The two sums in the middle need to be subtracted from the bitheap, so they are complemented and sign-extended before being added to the bitheap, as

shown in Section 6.1.2. Subtracting a product is a similar process. A good starting point is Equation 7.3, where the signs need to be adjusted. There is, however, one catch: negating the product of two unsigned numbers is not necessarily a negative number, the exception being represented by zero, which has a positive sign.

Using Equation 7.3 brings the same problem evoked in the beginning of Chapter 7: the designer's control over the implementation of the architecture. In order to take advantage of the multiplier blocks, for example, a different splitting is required. Equation 7.5 is a generalization of this idea. X and Y are split into their high and low parts:

$$\begin{aligned} X &= 2^{m-k} \cdot X_H + X_L \\ Y &= 2^{n-k'} \cdot Y_H + Y_L \end{aligned} \tag{7.4}$$

Therefore:

$$\begin{aligned} X \cdot Y = {} & 2^{m+m-k-k'} \cdot X_H \cdot Y_H \\ & + 2^{m-k} \cdot X_H \cdot Y_L \\ & + 2^{n-k'} \cdot X_L \cdot Y_H \\ & + X_L \cdot Y_L \end{aligned} \tag{7.5}$$

X_H and Y_H are formed of the k and k' most significant bits of X and Y, respectively, and can be seen as signed numbers. X_L and Y_L are formed of the $m-k$ and $n-k'$ least significant bits of X and Y, respectively, and can be seen as unsigned numbers. The values of k and k' are chosen according to the target platform. Xilinx supports 25×18-bit signed multiplications, while on Altera the 36×36-bit multipliers can be split in smaller sizes.

Equation 7.5 only shows a two-way split, but dividing the multiplicands in more blocks might be required, so as to maximize the use of the resources available on the target device.

Results on multipliers

This section shows some results of syntheses of multipliers designed using bitheaps. Table 7.1 is an extended version of Table 6.3, with more accent on the flexibility of the multiplier framework. It also adds some results for a multiplier implementation which is based on multi-input addition, instead of using a bitheap. It is denoted as *Components* in the table.

As can be seen in Table 7.1, the bitheap-based implementation is the most efficient in terms of resources out of the three methods compared. The exception is the DSP-based 53×53-bit multiplier, where the component-based method is better. This is probably due to a good match between the

Approach	Performance cycles @ frequency	LUTs	Registers	DSPs
precision = 23 bits				
Classical	1 cycles @ 215 MHz	0	17	2
Classical	1 cycles @ 179 MHz	798	135	0
Components	5 cycles @ 476 MHz	17	29	2
Components	7 cycles @ 411 MHz	991	563	0
Bitheap	2 cycles @ 477 MHz	0	5	2
Bitheap	2 cycles @ 254 MHz	248	138	1
Bitheap	1 cycles @ 255 MHz	771	78	0
precision = 32 bits				
Classical	1 cycles @ 133 MHz	0	34	4
Classical	1 cycles @ 168 MHz	1571	168	0
Components	7 cycles @ 404 MHz	184	120	4
Components	7 cycles @ 381 MHz	1717	1006	0
Bitheap	3 cycles @ 445 MHz	57	81	4
Bitheap	3 cycles @ 250 MHz	348	265	3
Bitheap	3 cycles @ 280 MHz	449	266	2
Bitheap	3 cycles @ 244 MHz	969	311	1
Bitheap	2 cycles @ 249 MHz	1469	114	0
precision = 53 bits				
Classical	1 cycles @ 96 MHz	211	120	15
Classical	1 cycles @ 137 MHz	4259	368	0
Components	13 cycles@ 454 MHz	324	390	12
Components	9 cycles @ 381 MHz	6047	2794	0
Bitheap	7 cycles @ 250 MHz	458	437	12
Bitheap	7 cycles @ 250 MHz	1060	621	6
Bitheap	4 cycles @ 244 MHz	4002	497	0

Table 7.1: Integer Multiplications on Xilinx Virtex-6 (using FloPoCo 4.0)

size of the terms that need to be added by the multi-input adders. It should also be remarked that the component-based method obtains better maximum running frequencies, but this is at the cost of a vastly increased latency.

8

Constant Multiplication

Having seen the opportunities that FPGAs offer in terms of multiplications, illustrated throughout Chapter 7, the utility of dedicated multipliers by constants could be argued. In order to motivate it, a very basic example is considered, with the help of Figures 8.1 through 8.3.

Figure 8.1 presents the dot diagram of the multiplication between two 8-bit fixed-point numbers X and Y. This is similar to the illustration in Figure 6.3. In a naive implementation of the multiplication, the figure shows the 64 partial products that have to be first computed, and then added together.

However, as this is a constant multiplication, one of the multiplicands can be replaced with its value, known in advantage. For example, Y can be replaced with the 8-bit constant `10110101`. This is illustrated in Figure 8.2, where the white-filled bits represent constant 0 bits and the black-filled bits represent constant 1 bits. It is clear that many of the lines containing partial products are zero and could therefore be eliminated from the heap.

Figure 8.3 illustrates the multiplication, where the partial products that are equal to zero have been eliminated. A comparison between Figure 8.1 and Figure Figure 8.3 shows the basic optimizations that can be made on a multiplications with a constant. Comparing them also shows the savings which can be achieved in terms of resources required for the implementation.

Of course, this is only a toy example, and more complex techniques for multiplications by constants can be even more efficient. The rest of Chapter 8 presents such a table-based technique.

Consider the multiplication between a fixed-point number X and the real constant C. As there are a finite number of values that X can take (2^w, to be precise, where w is the bit-width of X), the simplest method for multiplying by a constant is to directly tabulate all the possible results. Using

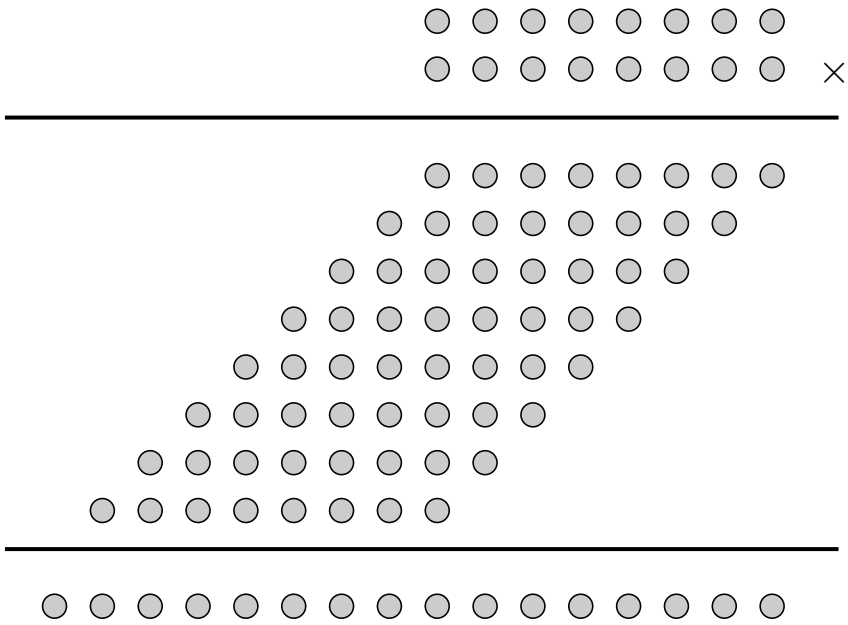


Figure 8.1: Dot diagram for the multiplication of two fixed-point variables X and Y

multi-precision software, the output values can be computed with a sufficiently large precision so that the operator returns the correctly rounded results.

For look-up table based FPGAs, this is the best as it gets, both in terms of delay, as well as resource consumption. The underlying assumption is, of course, that the width of $w \leq \alpha$, where α denotes the number of inputs to the LUTs (6 on most contemporary target FPGA devices). Therefore, for $w \leq 6$, the constant multiplier consumes one LUT per output bit. It should also be noted that for a real-valued constant C this method is more accurate than using a correctly rounded multiplier which inputs the constant rounded to a given target output precision p. This is because the latter method accumulates two rounding errors: one for the constant and one for rounding the output to p bits.

However, this method has one major disadvantage. In general, for an input of size $w = 6 + w'$ it requires $2^{w'}$ LUTs per output bit. For obvious reasons, this method does not scale well with the increase of the input size.

One answer to this problem is a method inspired from the KCM method in [26], which was later improved by [134], and extended to multiplications by a real constant in [38]. In [26], only integer constant are targeted. In the following, the method presented also handles real constants.

The problem with direct tabulation is the exponential increase of the size of the table with the size of the input. To get around this problem, the method here splits the input X into n chunks $X_0, X_1, X_2, \ldots X_{n-1}$ of sizes equal to α, where $n = \lceil \frac{w}{\alpha} \rceil$. Therefore, X can be written as:

$$X = 2^{-p} \cdot (2^{(n-1)\alpha} \cdot X_{n-1} + \ldots + 2^{2\alpha} \cdot X_2 + 2^{\alpha} \cdot X_1 + X_0) \tag{8.1}$$

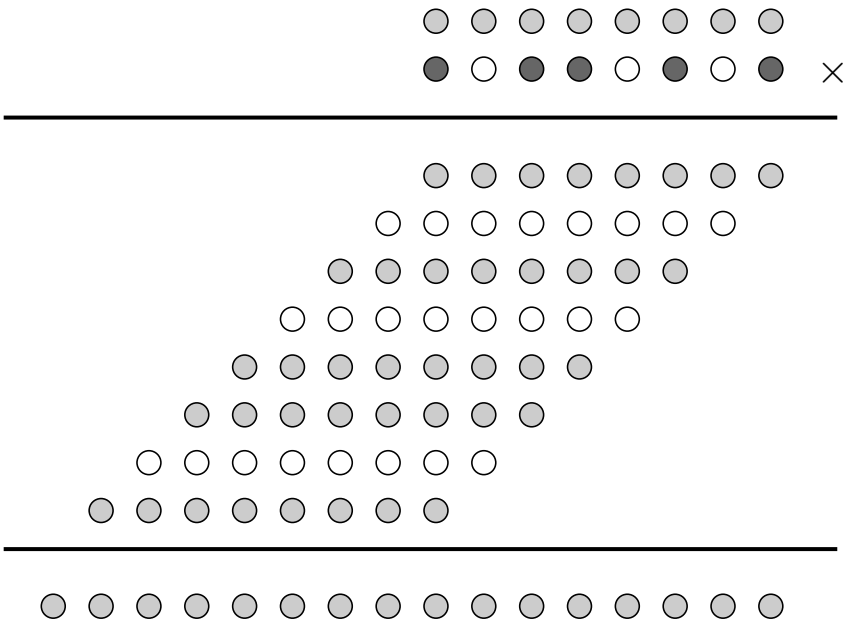


Figure 8.2: Dot diagram for the multiplication of a fixed-point variables X and the constant 10110101, with the zeros in the constant outlined

where $X_i \in \{0, \ldots, 2^{\alpha-1}\}$. The product between X and the constant C becomes:

$$C \cdot X = 2^{-p} \cdot (2^{(n-1)\alpha} \cdot C \cdot X_{n-1} + \ldots + 2^{2\alpha} \cdot C \cdot X_2 + 2^{\alpha} \cdot C \cdot X_1 + C \cdot X_0) \tag{8.2}$$

Seeing as each of the sub-products in Equation 8.2 has the variable multiplicand of size α, they can each be efficiently tabulated. Each of the sub-products consumes one LUT per output bit. Figure 8.4 shows the alignment of the sub-products. As the constant is real, each of the sub-products has an infinite number of trailing bits, and thus needs to be rounded to an intermediary precision $p+g$, where g represents the number of extra guard bits. The error of such a multiplier is therefore:

$$\varepsilon = n \cdot 2^{-p-g-1} \tag{8.3}$$

Equation 8.3 shows the reason for computing the product with the extra guard bits: the error is proportional to 2^{-g} and can therefore be made as small as needed by increasing g.

Unlike the classical KCM multiplier, not all the tables have the same size, as shown in Figure 8.5. The $2^{-k\alpha}$ factors shift the sub-products to the right. The leading zeros need not be stored, and can be compensated for through the alignment of the sub-product in the final sum.

Adding the sub-products together is done with the help of a bitheap. All that is left to do is to through the results of the tabulations to the bitheap, as the products are already all aligned to the

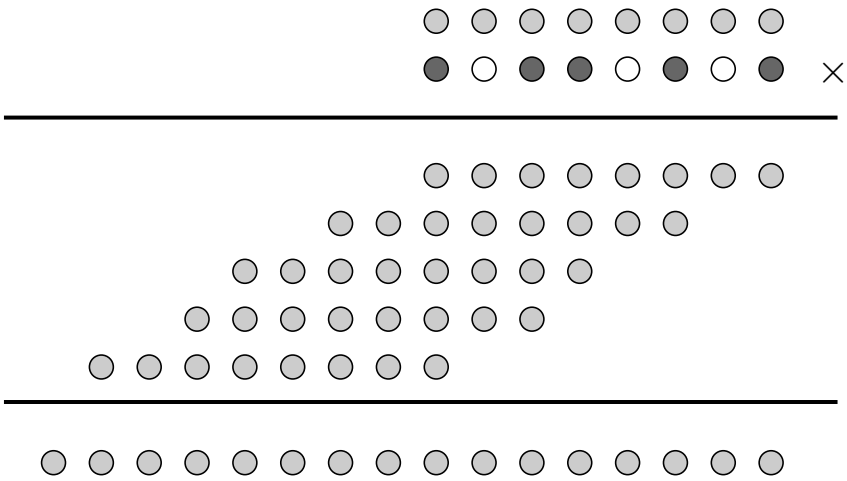


Figure 8.3: Dot diagram for the multiplication of a fixed-point variables X and the constant 10110101, without the zero rows

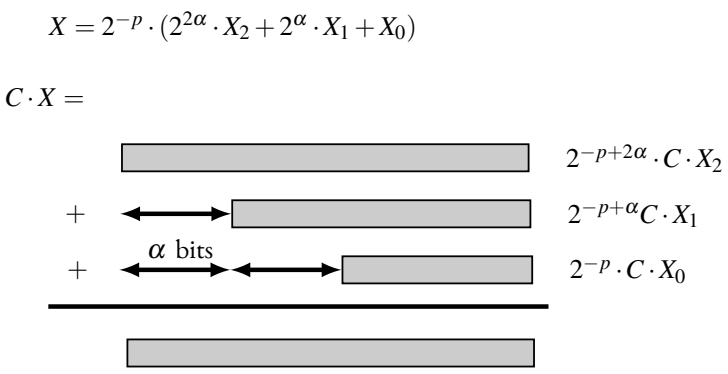


Figure 8.4: Alignment of the terms in the FixRealKCM multiplier

same LSB.

Synthesis results are shown in the following sections, where the constant multipliers are the basis of larger operators, such as sums of products by constants or digital filters (FIR and IIR). The larger operators are more relevant to show the optimizations that can be achieved, than just individual tests on the multipliers.

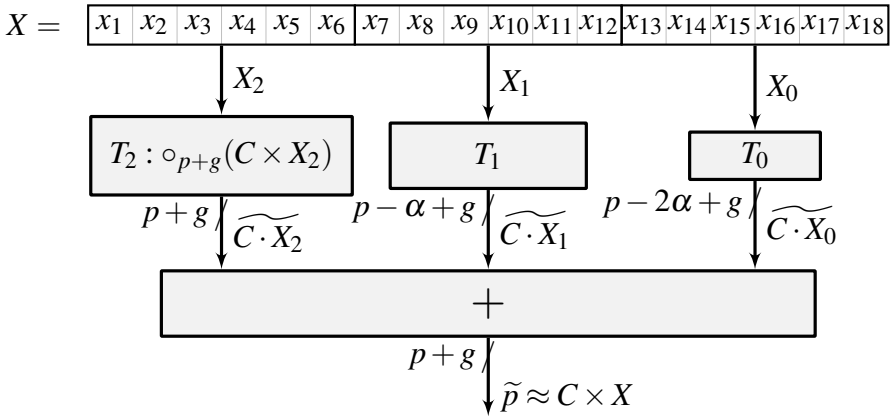


Figure 8.5: The `FixRealKCM` operator when X is split in 3 chunks

Part III

Arithmetic Functions

While Parts I and II focus on low-level problems concerning the back-end of a generator for arithmetic operators, the level of abstraction is raised in Part III. It focuses on taking advantage of the features introduced in the two previous parts for creating state of the art implementations for new arithmetic operators. The creation of each new operator added to FloPoCo's library is guided by its motto. Operators should be designed specific to their context (application and target platform), and should not mimic other implementations (typically those for CPUs).

Part III is divided into two main parts. Chapters 11 and 9 deal with univariate functions, while Chapter 10 considers the challenges of designing hardware implementations for multivariate functions. Among the single-input functions are algebraic functions such as the reciprocal, the square root of a number or the reciprocal square root, in Chapter 11. Some elementary functions are studied in Chapter 9: the sine and cosine trigonometric functions. Another trigonometric function, the two-input arctangent (commonly referred to as $\mathrm{atan}\,2$), is the a use-case for the study of multi-input functions, presented in Chapter 10. This exposes the difficulties of designing for this class of functions.

It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience.

often reduced to:

Everything should be made as simple as possible, but not simpler.

Albert Einstein

9 Sine and Cosine

The sine and the cosine functions are very common and very useful in disciplined such as signal processing, high-performance computing or computer graphics, to mention only a few. As such, this chapter surveys and compares three state-of-the-art methods for computing these functions. The context is that of modern FPGAs, with large look-up tables, embedded multipliers and embedded memory blocks. The focus is on high-throughput and resource-efficient implementations of the sine and/or the cosine functions for these platforms. The resulting architectures are implemented as new FloPoCo operators, and the framework is used for their testing and comparison. Some applications require the computation of one of the two functions, but there are many instances where both of them are required, for example computations that require rotations. Therefore, two of the methods explore the opportunities for optimization when computing a fused sine-cosine implementation.

Many of the available methods for computing the sine/cosine functions are based or use one of the properties of the complex exponential function:

$$e^{j(x+y)} = e^{jx} + e^{jy} \tag{9.1}$$

The geometrical interpretation of Equation 9.1 is that a rotation by the angle $x + y$ can be obtained by two rotations, one by the angle x and one by the angle y.

Euler's identity can be used to establish the connection between the complex exponential function and the trigonometric functions (illustrated in Figure 9.1):

$$e^{jx} = \cos(x) + j \cdot \sin(x) \tag{9.2}$$

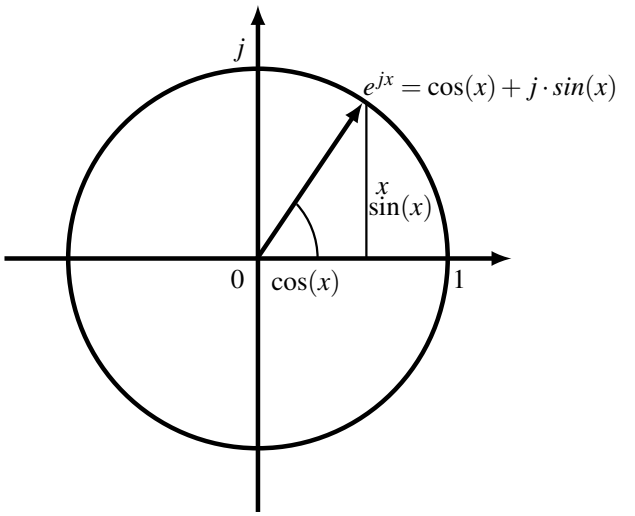


Figure 9.1: Euler's identity

Identifying the real and complex parts, the real version of Equation 9.1 becomes:

$$\begin{cases} \sin(x+y) &= \sin(x)\cos(y) + \cos(x)\sin(y) \\ \cos(x+y) &= \cos(x)\cos(y) - \sin(x)\sin(y) \end{cases} \tag{9.3}$$

Hence, the computation of the sine/cosine functions for a composite angle can be done by a composition of rotations of the complex number 1. These formula as known as the *prosthaphaeresis formulas*, and are sometimes referred to as *Simpson's formulas*.

A method based on this idea of decomposing the rotation by a composite angle into several micro-rotations is the classic CORDIC algorithm, introduced in [125] and extended in [130]. The method computes both the sine and the cosine functions, and was designed for use in aviation and to be implemented using integrated circuits. It also relies solely on additions and shifts, and is suitable for target platforms that do not have direct support for multiplications, or where multiplications are considerably more expensive than additions.

Most FPGAs have dedicated support for additions, through the fast-carry propagation chains. As such, the CORDIC method has attracted a lot of attention on this platform and has been well studied. For example, the survey of [13] reviews many useful techniques. There have also been many variations developed from the original, like the ones presented in [92] or in the survey of [84], which came out on the 50th anniversary of the CORDIC algorithm. And yet, due to the fact that most FPGAs are LUT-based and due to their better support for additions than for multiplications, most preferred implementations remain the simpler ones, like that of [121]. In general, CORDIC imple-

mentations have a nice property, which is that their resource consumption is quite easy to estimate. For a sine/cosine computation faithful to w bits it requires roughly $3 \cdot w$ additions, each of size w.

A variant of the CORDIC algorithm with "reduced iterations" is introduced in [11], and then generalized in [120]. The idea is to replace approximately half of the iterations in the CORDIC algorithm with two multiplications and two additions, which, after having performed half of the iterations, should be fairly small.

In order to truly take advantage of the embedded multipliers and the embedded memory blocks different approaches and decompositions of the input angle are required. Several techniques for dividing the input angle and the ensuing rotations have been explored in [47] and [92]. In the following sections, a method for computing the sine and cosine using a coarser decomposition is explored. The input angle is divided into its high and low parts, whose sizes are influenced by the sizes of the multiplier and memory blocks. If the sizes are chosen appropriately, the high part can be used for addressing a table that stores the values for the functions. The sine/cosine of the lower part can be computed using polynomial approximation, which is typically of a small degree, and thus incurs low resource requirements.

A Common Specification

In order to have a correct and relevant comparison between methods for computing the sine and cosine, a common interface and specification for the operators must first be established.

As such, the functions computed are the $\sin(\pi \cdot x)$ and/or $\cos(\pi \cdot x)$. The argument x is a fixed-point number belonging to $[-1, 1)$ and using two's complement notation. This input format is chosen because it maps the periodicity of the trigonometric functions to the periodicity of the two's complement representation.

There is, however, one problem with the output range: the asymmetry of the $[-1, 1)$ range, where -1 is representable, but 1 is not. The sine and cosine have the co-domain $[-1, 1]$. In order to avoid overflow, the value 1 must actually represented as $0.11\cdots1_2 = 1 - 2^{w_{out}}$, where w_{out} is the width of the output. The value -1 should consequently be represented as $-1 + 2^{w_{out}}$.

With these considerations in mind, the functions computed in the rest of Chapter 9 are scaled, and are actually:

$$\begin{cases} (1 - 2^w) \cdot sin(\pi x) \\ (1 - 2^w) \cdot cos(\pi x) \end{cases} \tag{9.4}$$

where w is the bitwidth of the input x and of the output. This scaling is not a problem in practice. Applications using the trigonometric function can compensate, if required, for the error introduced by the scaling by increasing the precision of the computations w. It will be shown that from an implementation stand point, the scaling does not incur any extra cost, as it can be incorporated in the

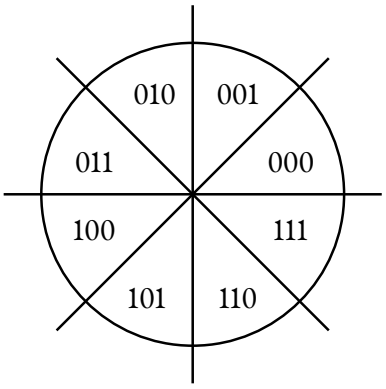


Figure 9.2: The values for *sqo* in each octant

existing computations.

All the methods of Chapter 9 use fixed-point representations. Between the trigonometric functions' periodicity and their output in the $[-1, 1]$ interval, most implementations should benefit and are a good match for a fixed-point datapath. As such, a floating-point implementation of the sine/-cosine functions should benefit from a conversion to fixed-point. The overhead incurred by the floating-point implementation of these functions has already been studied in the literature. Take for example [40], which deals with this problem.

In terms of the accuracy of the computed results, all the operators should be *faithful*, which means that they are last-bit accurate. This ensures that all the computed bits are correct, and therefore there are no wasted resources. The error (ε), with respect to the true mathematical result is less that the weight of the LSB of the result: $\varepsilon < 2^{-w}$. Using more precision would compute bits that the user cannot benefit from. Using less precision would mean that some of the output is wrong, and therefore, the user cannot benefit from a part of the output either.

As a last specification, all the architectures considered are pipelined implementations, which output one result each cycle (once the pipeline is full, of course). An alternative would be to have sequential architectures, which output a result after a given number of cycles. Some of the methods to be presented could be adapted to function this way, but this is out of the scope of this chapter.

Argument Reduction

The computation of the sine/cosine function can be sped-up using an argument reduction, which is based on some of the basic trigonometric identities. This allows for a faster and less demanding architecture in terms of resources, at the cost of a few comparisons. The argument reduction applies for all of the methods analyzed in Chapter 9. It relies on dividing the input x into two parts, its 3 most significant bits and the rest of the number. The most significant bits are denoted s, for sign,

sqo	Reconstruction
000	$\begin{cases} \sin(\pi x) = \sin(\pi y') \\ \cos(\pi x) = \cos(\pi y') \end{cases}$
001	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = \sin(\pi y') \end{cases}$
010	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = -\sin(\pi y') \end{cases}$
011	$\begin{cases} \sin(\pi x) = \sin(\pi y') \\ \cos(\pi x) = -\cos(\pi y') \end{cases}$

Table 9.1: The reconstruction for the sine/cosine function values using *sqo* (only positive input values illustrated)

q, for quadrant and o for octant, while the lower bits of x are denoted as y, with $y \in [0, \frac{1}{4})$. This is illustrated in Figure 9.2. Therefore, the sine/cosine functions can now be computed only for $\sin(\pi \cdot y')$ and $\cos(\pi \cdot y')$, where

$$y' = \begin{cases} \frac{1}{4} - y \text{ if } o = 1 \\ y \text{ otherwise.} \end{cases} \tag{9.5}$$

with $y' \in [0, \frac{1}{4})$. The final values for the sine/cosine functions can be reconstructed using Table 9.1, which is based on some basic trigonometric identities, such as $\sin(\pi \cdot x) = \cos(\frac{\pi}{2} - \pi \cdot x)$ etc..

The CORDIC method

The first method for computing the sine/cosine functions that is analyzed in Chapter 9 is the classic CORDIC method. The focus in Section 9.3 is on its *rotation mode*. Chapter 10 shows how the *vectoring mode* can be used for the computation of the arctangent.

The CORDIC algorithm decomposes the input angle y' in a base formed by the angles $x_i = \arctan(2^{-i})$, using a non-restoring algorithm [92]. As such, y' can be expressed as:

$$y' = \sum_{i=0}^{\infty} d_i \cdot x_i \tag{9.6}$$

What is left to do is to express the rotation by y' as a series of rotations by the angles $d_i \cdot x_i$, using Equations 9.3 and 9.6. The starting point is the point (c_0, s_0), and at each rotation the point (c_{i+1}, s_{i+1}) is obtained from (c_i, s_i), by a rotation of $d_i \cdot x_i$ (as illustrated in Figure 9.3):

$$\begin{pmatrix} c_{i+1} \\ s_{i+1} \end{pmatrix} = \begin{pmatrix} \cos(d_i \cdot x_i) & -\sin(d_i \cdot x_i) \\ \sin(d_i \cdot x_i) & \cos(d_i \cdot x_i) \end{pmatrix} \cdot \begin{pmatrix} c_i \\ s_i \end{pmatrix} \tag{9.7}$$

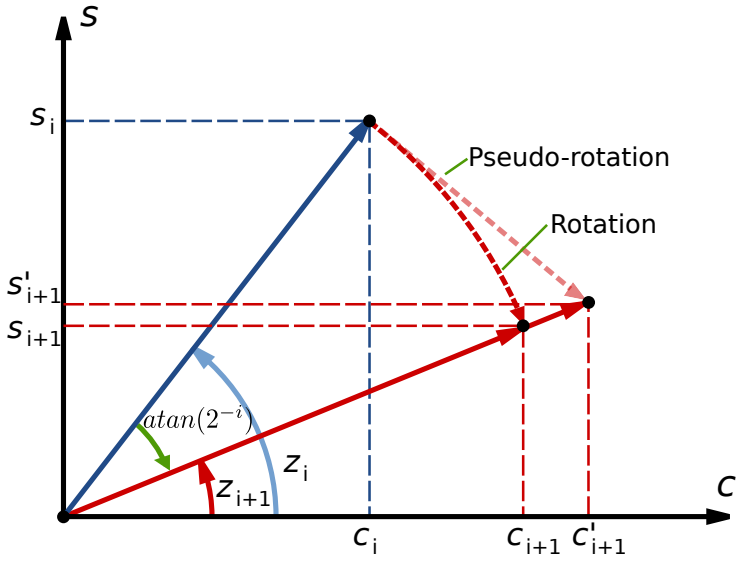


Figure 9.3: An illustration of one iteration of the CORDIC algorithm

where the direction of the rotation is determined as:

$$\begin{aligned} t_0 &= 0 \\ t_{i+1} &= t_i + d_i \cdot x_i \\ d_{i+1} &= \begin{cases} 1 & when \quad t_i \leq y' \\ -1 & otherwise \end{cases} \end{aligned} \tag{9.8}$$

Equation 9.7 can be simplified based on two remarks. First of all, the variables $d_i = \pm 1$, and the cosine and sine functions are even and odd, respectively. Therefore, $\cos(d_i \cdot x_i) = \cos(x_i)$ and $\sin(d_i \cdot x_i) = d_i \cdot \sin(x_i)$. Second, $x_i = \arctan(2^{-i})$, therefore $\tan(x_i) = 2^{-i}$. By taking $\cos(x_i)$ as a factor, Equation 9.7 can be reduced to:

$$\begin{pmatrix} c_{i+1} \\ s_{i+1} \end{pmatrix} = \cos(x_i) \cdot \begin{pmatrix} 1 & -d_i \cdot 2^{-i} \\ d_i \cdot 2^{-i} & 1 \end{pmatrix} \cdot \begin{pmatrix} c_i \\ s_i \end{pmatrix} \tag{9.9}$$

A multiplication by a 2^{-i} factor is just a shift, while the d_i factors only influence the sign of the addend. To obtain a multiplier-less algorithm, CORDIC does not multiply by $\cos(x_i) = \frac{1}{\sqrt{1+2^{-2i}}}$, and instead of Equation 9.9 computes a pseudo-rotation, as shown in Figure 9.3:

$$\begin{pmatrix} c_{i+1} \\ s_{i+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_i \cdot 2^{-i} \\ d_i \cdot 2^{-i} & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{i+1} \\ s_{i+1} \end{pmatrix} \tag{9.10}$$

Equation 9.8 can also be simplified, by performing a variable change: $z_i = y' - t_i$:

$$\begin{array}{rcl} z_0 & = & y' \\ z_{i+1} & = & z_i - d_i \cdot x_i \\ d_{i+1} & = & \begin{cases} 1 & when \quad z_i \geq 0 \\ -1 & otherwise \end{cases} \end{array} \tag{9.11}$$

Therefore, d_i is given by the sign of z_i. Finally, the sine/cosine functions can be computed using the following iteration:

$$\begin{cases} c_0 & = & \frac{1}{\Pi_n^{i=0}\sqrt{1-2^{-2\cdot i}}} \\ s_0 & = & 0 \\ z_0 & = & y' \quad \text{the reduced argument} \end{cases} \tag{9.12a}$$

$$\begin{cases} c_{i+1} & = & c_i - 2^{-i} \cdot d_i \cdot s_i \\ s_{i+1} & = & s_i + 2^{-i} \cdot d_i \cdot c_i \\ z_{i+1} & = & z_i - d_i \cdot w_i, \quad \text{where } w_i = \arctan(2^{-i}) \\ d_i & = & +1 \text{ if } z_i > 0, \quad \text{else } -1 \end{cases} \tag{9.12b}$$

which converge to:

$$\begin{cases} c_{n\longrightarrow\infty} & = & \cos(y') \\ s_{n\longrightarrow\infty} & = & \sin(y') \\ z_{n\longrightarrow\infty} & = & 0 \end{cases} \tag{9.13}$$

As can be seen in Equation 9.12, the multiplications by the $\cos(x_i)$ factors that are dropped between Equations 9.9 and 9.10 are compensated for by initializing suing the factor $K = \prod_{i=0}^{n} \sqrt{1-2^{-2\cdot i}}$. Therefore, they incur no actual cost. An illustration of the evolution of the z angle throughout the iterations of the CORDIC algorithm is presented in Figure 9.4.

A fully unrolled implementation of the CORDIC method in its rotation mode has been implemented, and is used for discussions and comparisons in the following. The implementation is fully pipelined and illustrated in Figure 9.5.

There are a few remarks to be made regarding the implementation, in the context of FPGAs. First of all, the $x_i = \arctan(2^{-i})$ terms used for the iterations on z_i are constants. As such, they are folded into the logic on the FPGA, and need not be stored in dedicated tables. Second of all, Equation 9.12 in its current form is designed for input arguments given in radians. This means that a scaling of the constants used in Equation 9.12 is required, namely for the x_i factors. Fortunately, this can be done a priori, and does not incur any additional cost. A scaling by $1 - 2^{-w}$ (where w is the accuracy) is also required, as mentioned in Section 9.1. This can also be integrated in the initial state, in the top

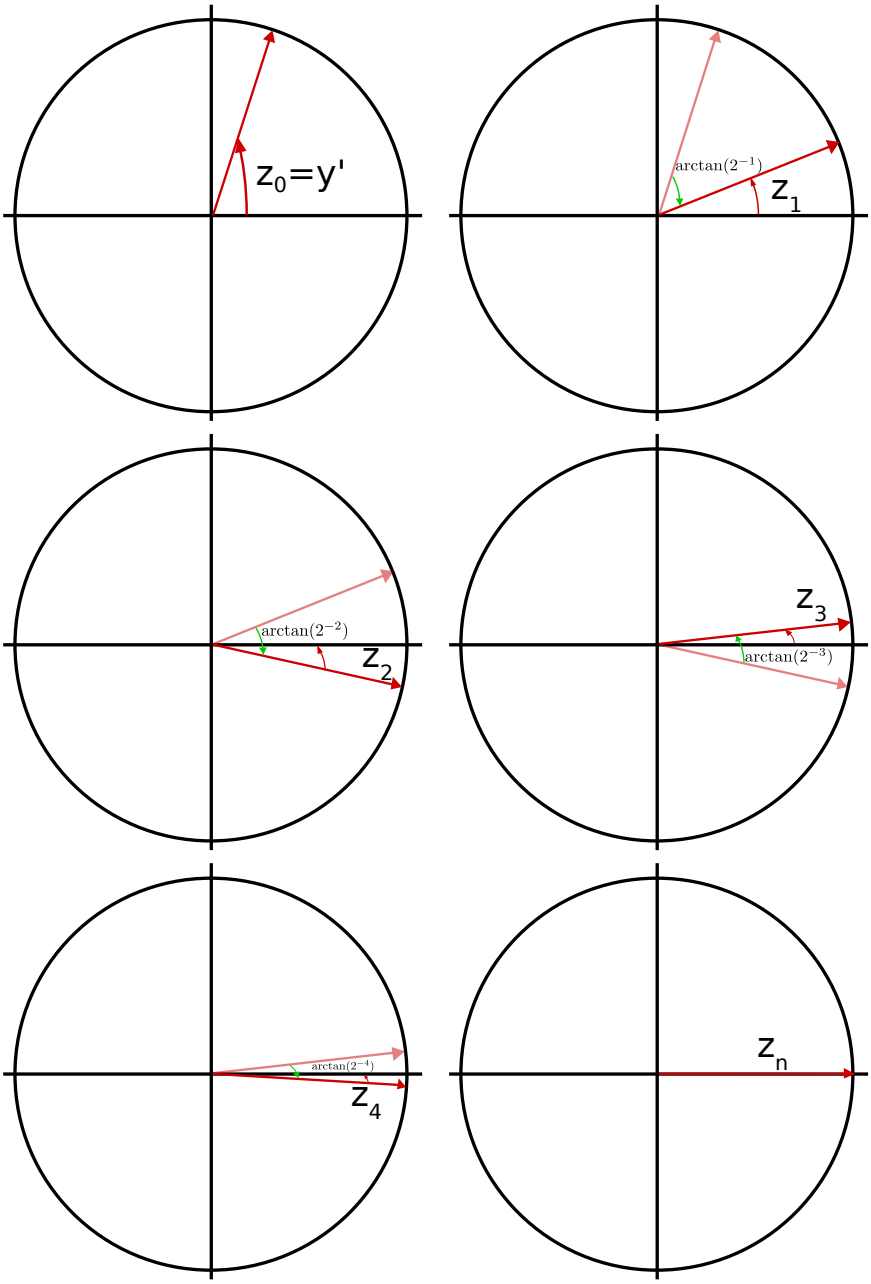


Figure 9.4: An illustration of the evolution of the angle z throughout the iterations

part of Equation 9.12. A last remark relates to number of iterations required for convergence. In Equation 9.13, the convergence is given for $n \longrightarrow \infty$. However, only a bounded number of iterations n is required for obtaining a faithful result on the output accuracy w. Usually, n is not much larger than w, and more details on how it is determined are given in the following sections.

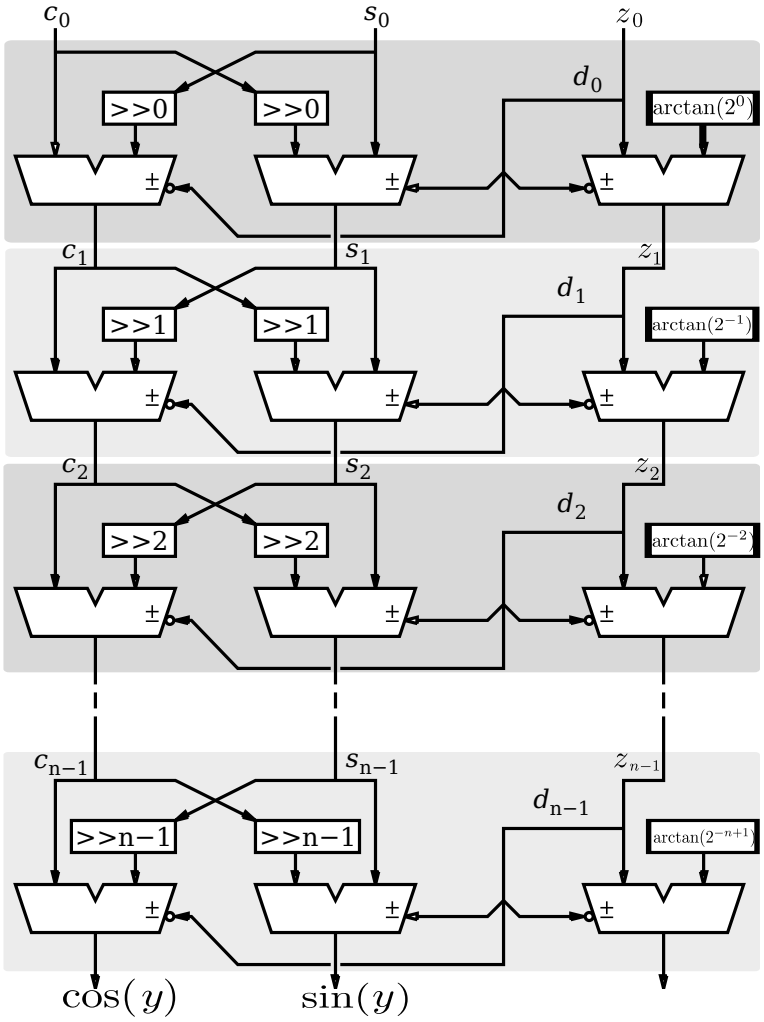


Figure 9.5: An unrolled implementation of the CORDIC method. Bitwidths for the c and s datapaths remain constant, while the bitwidth of the z datapath decreases by 1 bit at each level.

An Error Analysis For the CORDIC Method

In order to ensure the most efficient architecture in terms of resources and in terms of speed, while still meeting the accuracy specification, the possible errors performed during the computations need to be determined. There are two possible types of errors: the method error and the rounding errors. The rounding errors are themselves coming from two sources: the rounding/truncation operations performed during the computations and the final rounding of the outputs to the target accuracy 2^{-w}. Hence, the total error can be expressed as:

$$\varepsilon_{total} = (\varepsilon_{final\_round} + \varepsilon_{round}) + \varepsilon_{method} \tag{9.14}$$

Classically, an error budget is established when performing the error analysis. As the accuracy specification is faithful rounding, or last bit accuracy, the maximum allowed error must be smaller than 2^{-w}:

$$\varepsilon_{total} < 2^{-w}$$

The output values c_n and s_n are rounded to the output precision 2^{-w}. This operation entails an error $\varepsilon_{final\_round} \leq 2^{-w-1}$. This therefore requires that $\varepsilon_{round} + \varepsilon_{method} < 2^{-w-1}$.

The method error ε_{method} of the CORDIC method, as per Equation 9.12 is proportional to the magnitude of z_i, at the moment when the iterations are stopped. A bound on ε_{method} can be computed using numerical methods. This is how the number of iterations n is determined. It represents the smallest integer for which $\varepsilon_{method} < 2^{-w-2}$. The remaining part of the error budget is allocated towards the rounding errors ε_{round}.

In order to compensate for the inevitable rounding errors, the computations on the c, s and z datapaths are performed using an increased accuracy. The computations are done using g extra guard bits, which will accumulate the errors, instead of them accumulating in the bits returned at the output.

Concerning the z datapath, the errors introduced at each iteration i come from the $x_i = \arctan(2^{-i})$ constant factors. As they cannot be stored with infinite accuracy, the next best thing to do is to compute them using multi-precision software and then store the correctly rounded on $w+g$ bits. Therefore, they introduce an error which is bounded: $\varepsilon_{x_i} < 2^{-w-g-1}$. After n iterations, these accumulate and can be bounded by $n \cdot 2^{-w-g-1}$.

Concerning the c and s datapaths, a recurrence formula for the rounding errors performed when computing the sine and the cosine can be formulated. The lower part of Equation 9.12 can be expressed as follows, when taking into consideration the errors:

$$\begin{cases} \widetilde{c}_{i+1} &= \widetilde{c}_i - 2^{-i} \cdot d_i \cdot \widetilde{s}_i + \varepsilon_{trunc} \\ \widetilde{s}_{i+1} &= \widetilde{s}_i + 2^{-i} \cdot d_i \cdot \widetilde{c}_i + \varepsilon_{trunc} \end{cases} \tag{9.15}$$

where the tilde symbols denote terms with error. Expanding Equation 9.15 gives:

$$\begin{cases} \widetilde{c}_{i+1} &= c_i + \varepsilon_{c\_i} - 2^{-i} \cdot d_i \cdot (s_i + \varepsilon_{s\_i}) + \varepsilon_{trunc} \\ \widetilde{s}_{i+1} &= s_i + \varepsilon_{s\_i} + 2^{-i} \cdot d_i \cdot (c_i + \varepsilon_{c\_i}) + \varepsilon_{trunc} \end{cases} \tag{9.16}$$

Therefore, the errors on the c and s datapaths can be defined as:

$$\begin{cases} \varepsilon_c &= \widetilde{c}_i - c_i \\ \varepsilon_s &= \widetilde{s}_i - s_i \end{cases} \tag{9.17}$$

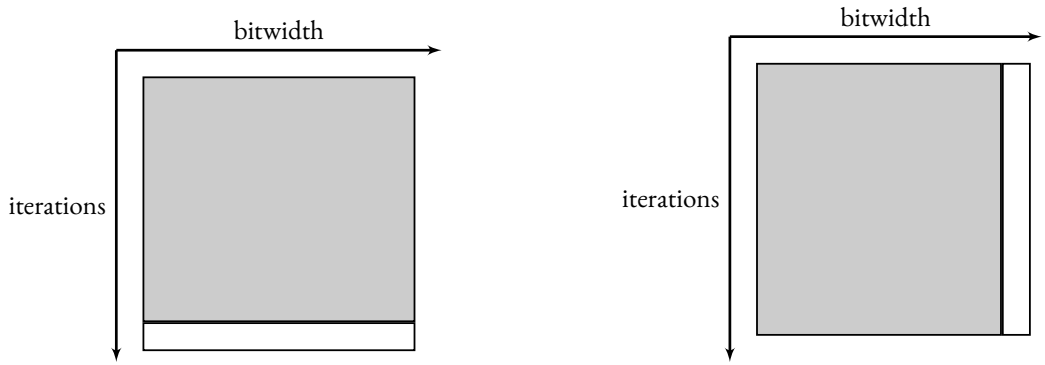


Figure 9.6: The trade-of between using roundings and truncations for the c and s datapaths

From Equations 9.16 and 9.17, and considering a common error bound for ε_c and ε_s denoted ε_{cs}, an iterative definition can be deduced:

$$\begin{cases} \varepsilon_{cs\_0} &= 2^{-w-g-1} \\ \varepsilon_{cs\_i+1} &= \varepsilon_{cs\_i} + 2^{-i} \cdot \varepsilon_{cs\_i} + \varepsilon_{trunc} \end{cases} \tag{9.18}$$

The term ε_{trunc} represents the error entailed by computing the $2^{-i} \cdot d_i \cdot c_i$ and $2^{-i} \cdot d_i \cdot s_i$ terms in Equation 9.12 using a truncation and is bounded by 2^{-w-g}.

Indeed, this error could be diminished by using a rounding instead of a truncation. This has an impact of requiring one extra bit for the adders used for the computations. In return, it divides the error by a factor that is less than two, and therefore requiring one less iteration of the CORDIC algorithm. In the grand scheme of things, there is no actual benefit in doing this: one set of additions is saved, which is proportional to $\approx 3(w+g)$ LUTs, while adding one extra bit to all the remaining operations, which has a cost proportional to $\approx 3(w+g)$ LUTs. This trade-of is illustrated in a graphical manner in Figure 9.6. The rectangle on the left represents the solution using roundings, while the one of the right corresponds to the solution using truncations. In both images, the gray areas represent the computations performed (and hence the required resources) and the white areas represent the ones which are saved (computations as well as resources).

The rounding errors due to computations on the c and s datapaths ε_{cs_n} can be determined using numerical methods. It can be observed that ε_{cs_n} is generally larger than ε_z. Therefore, it is ε_{cs_n} that will be used to determine the number of guard bits g, and thus the accuracy of the computations: the smallest integer that ensures $\varepsilon_{cs\_n} < 2^{-w-2}$.

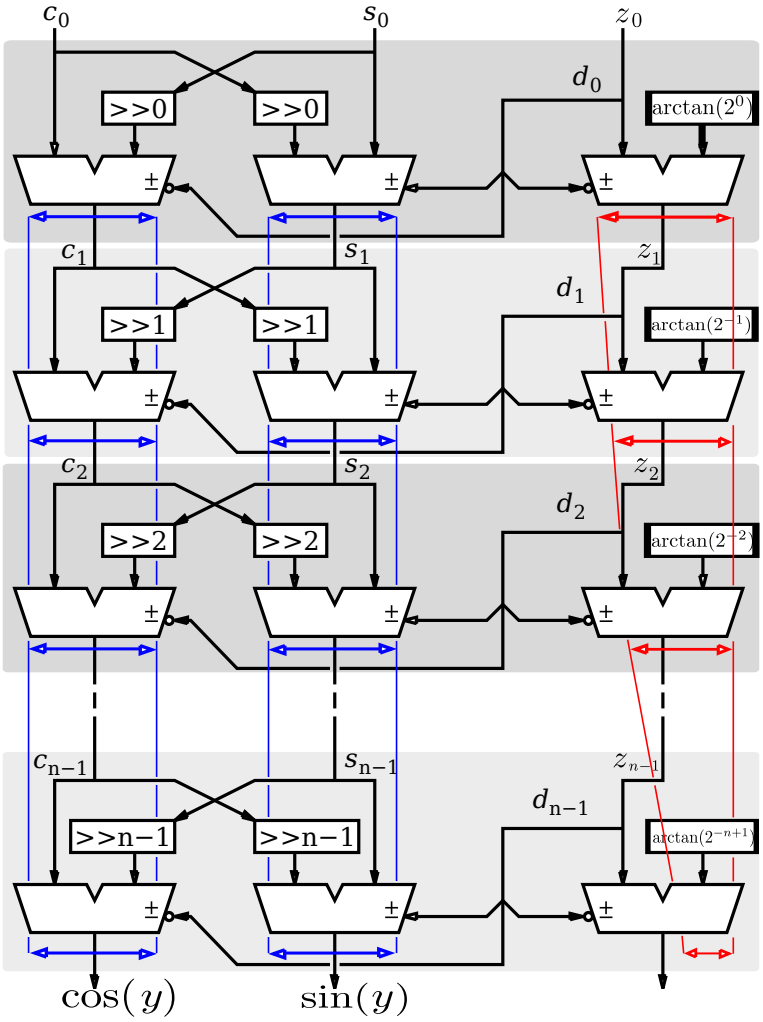


Figure 9.7: The z datapath can be reduced by its most significant bit at each iteration

Reducing the z Datapath

The CORDIC implementation can be further improved based on an remark concerning the z datapath. It can be observed that the $x_i = \arctan(2^{-i})$ constant factors are of the same order of magnitude at each iteration i as the z_i terms. Therefore, z_i has $i-1$ leading zeros. Taking into consideration that a fixed-point format is used, the computations on the z datapath can be performed on one less bit at each iteration, without impacting the correctness of the results. The resources required for the CORDIC implementation are reduced by approximately $\frac{1}{6}$. An illustration of the decreasing size of the z datapath is provided in Figure 9.7.

Another beneficial side-effect is the reduction of the critical path. The data dependency between

iterations follows the path $z_i \longrightarrow d_i \longrightarrow z_{i+1}$. As z_i gets shorter, d_i can be computed faster and a new iteration on c_{i+1} and s_{i+1} can be started before the previous one has ended. The moment the bits of c_i and s_i that are required for computing c_{i+1} and s_{i+1} are ready, the computations can be launched and execute in parallel. It is unfortunate that later iterations, when d_i can be computed the fastest, can benefit less from this speed-up, as c_i and s_i get shifted increasingly to the right.

These subtleties would require a delicate manipulation and skillfulness in managing the pipeline specification. The CORDIC implementation uses the pipelining framework that presents the one introduced in Chapter 4. However, a nice side-effect of porting the operator to the new pipeline framework is that this inherent synchronization inside the z and the c and s datapaths follows naturally and requires no effort on the side of the designer.

Reduced Iterations CORDIC

Half of the iterations in the CORDIC method can be traded-of for a single rotation by the remaining angle z_i at that point.* The key here is to stop the rotations, as soon as $\sin(z_i)$ can be approximated by z_i and $\cos(z_i)$ can be approximated by 1 on the target output accuracy. At which point the remaining rotations can be replaced by:

$$\begin{cases} c_{i+1} &= c_i + z_i \cdot s_i \\ s_{i+1} &= s_i - z_i \cdot c_i \end{cases} \tag{9.19}$$

With the considerations of Section 9.3.2, it can be remarked that z_i is of a size slightly larger than $\frac{w+g}{2}$.

On the implementation side, in order to have an optimal solution c_i and s_i can be truncated to the same accuracy as z_i, before the multiplication. This means that the two multipliers required are each of size $\frac{w+g}{2} \times \frac{w+g}{2}$. For all practical purposes, for input sizes of up to 32 bits the last rotation can be implemented using two 18×18-bit multiplier blocks, which can also integrate the two additions. An illustration of the diagram schematic of the implementation is presented in Figure 9.8.

In order to ensure the correctness of the results, the error analysis has to be adapted accordingly. For example, the scaling factor has to be modified to account for a smaller number of iterations. The approximation of $\sin(z_i) \approxeq z_i$ and $\cos(z_i) \approxeq 1$ can be justified by considering the Taylor series for the two factors:

$$\begin{cases} \sin(z) &\approxeq z - \frac{z^3}{3!} + \frac{z^5}{5!} + \dots \\ \cos(z) &\approxeq 1 - \frac{z^2}{2!} + \frac{z^4}{4!} + \dots \end{cases} \tag{9.20}$$

*As a side-note, remark that [11], which introduced the idea of reduced iterations, also showed how the scale factor compensation can be avoided by choosing a specific shift sequence that compensates for the scale factor. This was also remarked in [85]

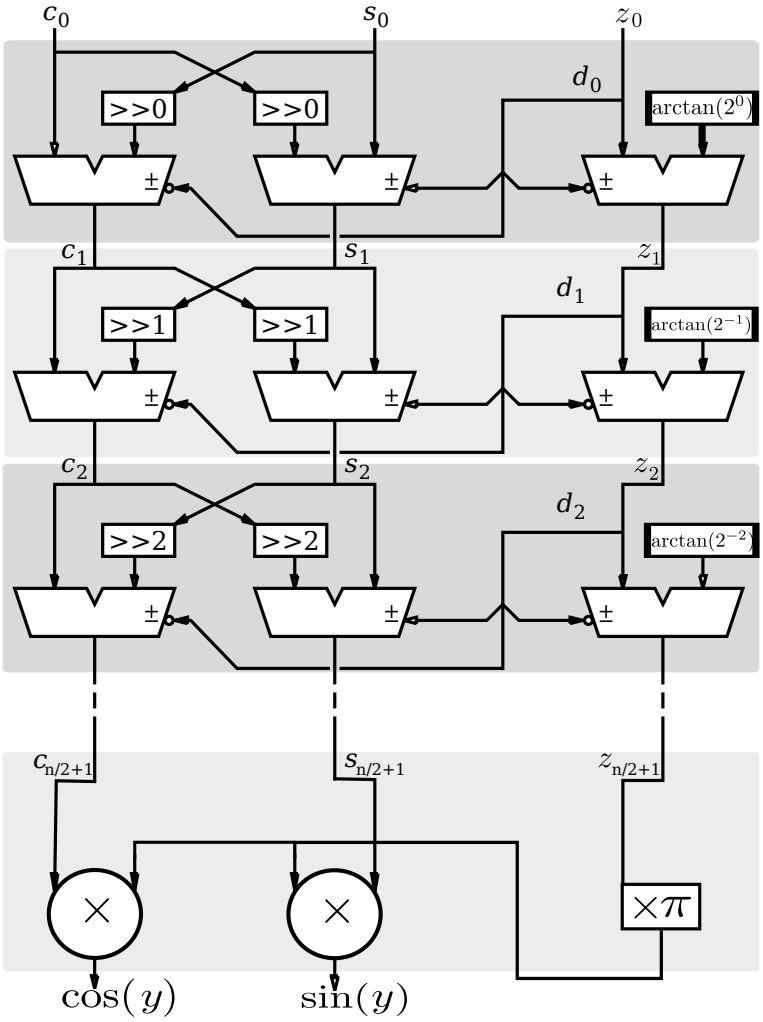


Figure 9.8: An implementation of the unrolled CORDIC method with reduced iterations

As soon as $|z| < 2^{\frac{-w-g}{2}}$, the two functions can be approximated using only the first term in their corresponding series. Notice, as well, that the iterations on the $\sin(z)$ datapath could actually be stopped even faster, at approximately one third of the total initial number of iterations $\frac{w+g}{3}$, once $|z| < 2^{\frac{-w-g}{3}}$. This could result in savings of about $\frac{1}{6}$ of the resources required for computing the sine datapath.

The Table- and Multiply-Based Parallel Polynomial Method

The second method for computing the sine/cosine functions is based on a coarser decomposition of the input angle. The main goal is to take advantage as efficiently as possible of *all* the available

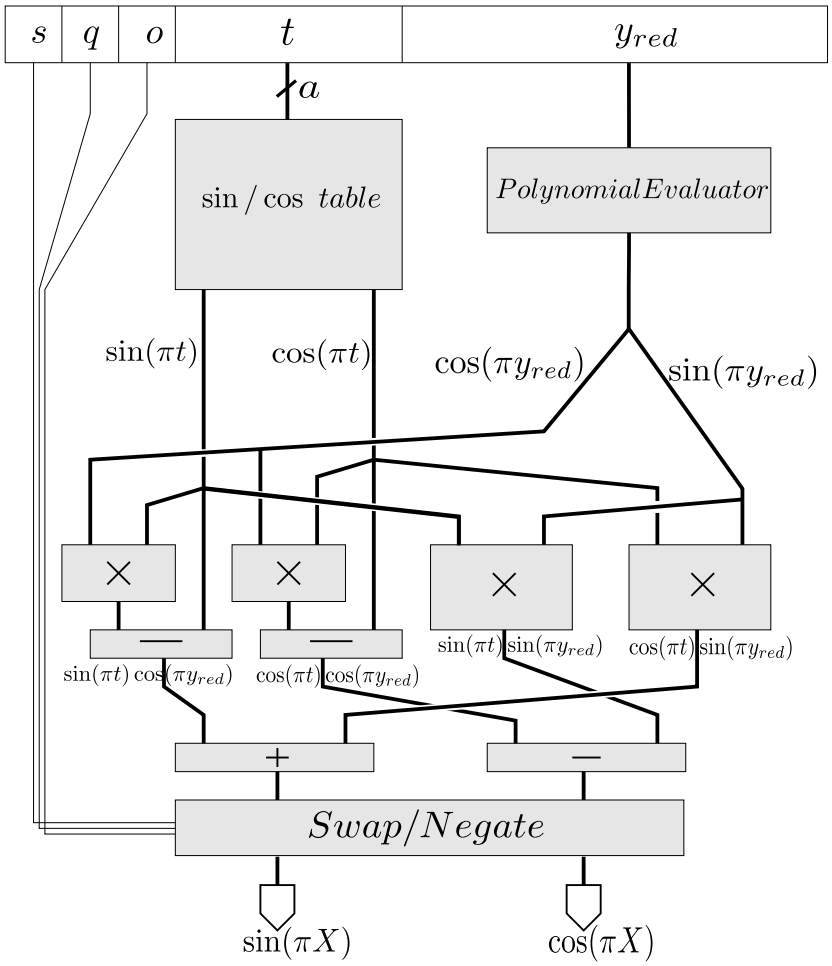


Figure 9.9: An illustration of the tabulate and multiply method for computing the sine and cosine

resources on a modern FPGA: look-up tables, embedded multipliers and embedded memories.

The reduced input y' is further split into two chunks: its most significant a bits, denoted hereafter t, and its least significant bits, denoted y_{red}. The LSB part y_{red} therefore belongs to $y_{red} \in [0, 2^{-(a+2)})$. The idea is to use t in order to address a table that stores $\sin(\pi \cdot t)$ and $\cos(\pi \cdot t)$. The sine and cosine corresponding to the remaining part of the input y_{red} is computed by first multiplying y_{red} by π. Then, the sine/cosine of

$$z = \pi \cdot y_{red}$$

are obtained through a polynomial approximation technique. The final results, $\sin(y')$ and $\cos(y')$ can be restored from the previously computed terms, based as well on Equation 9.3 for the sine/cosine of a sum of angles.

The technique chosen in the context of the sine/cosine functions for the polynomial approxima-

tion is based on the Taylor series about the point 0, also known as the Maclaurin series. Another option is the use of a generic polynomial approximation, as, for example, the one described in [35]. However, using the Taylor series allows taking advantage of two important properties of the sine/cosine functions: the sine is an odd function and the cosine is an even function. Therefore half of the coefficients are zero. Another property that is beneficial is that the first two terms in the series corresponding to the sine/cosine functions involve multiplications by powers of 2, or by $\frac{1}{3}$. The former can be implemented through shifts by a constant quantity, and do not require any resources. The latter can be implemented through efficient methods, like the one described in [34], which have low resource demands.

In order to simplify the evaluation of $\sin(z)$ and $\cos(z)$ the number of terms of the Taylor series used should be kept as small as possible. In this case, a reasonable trade-off between the size of the tables for $\sin(\pi \cdot t)$ and $\cos(\pi \cdot t)$ and the number of terms in the Taylor series (and therefore the number and size of the required multiplications) leads to neglecting terms beyond $\frac{z^3}{3!}$:

$$\begin{cases} \sin(z) & \approxeq & z - \frac{z^3}{6} \\ \cos(z) & \approxeq & 1 - \frac{z^2}{2} \end{cases} \tag{9.21}$$

In order for the polynomial approximation to be precise enough, there is one constraint that must be satisfied:

$$\frac{z^4}{24} \leq 1.02 \cdot 2^{-w-g}$$

which implies:

$$4 \cdot (a+2) - 2 \geq w + g$$

This expression can be used to determine the size a of the input of the tables storing $\sin(\pi \cdot t)$ and $\cos(\pi \cdot t)$. Using Equations 9.3 and 9.21, the values for $\sin(\pi \cdot y')$ and $\cos(\pi \cdot y')$ can be computed as:

$$\begin{cases} \sin(\pi \cdot y') & \approxeq & \sin(\pi \cdot t) \cdot \cos(z) + \cos(\pi \cdot t) \cdot \sin(z) \\ \cos(\pi \cdot y') & \approxeq & \cos(\pi \cdot t) \cdot \cos(z) - \sin(\pi \cdot t) \cdot \sin(z) \end{cases} \tag{9.22}$$

Equation 9.22 can be evaluated in a more efficient way by observing that $\cos(z)$ is close to 1, as the argument z is quite small ($z \in [0, 2^{-a-2})$). Therefore, the multiplication $\sin(\pi \cdot t) \cdot \cos(z)$ can be traded-of for a smaller multiplication and addition, and be evaluated as $\sin(\pi \cdot t) + \sin(\pi \cdot t)(\cos(z) - 1)$. As such, Equation 9.22 can be replaced by:

$$\begin{cases} \sin(\pi \cdot y') & \approxeq & \sin(\pi \cdot t) - \sin(\pi \cdot t) \cdot \frac{z^2}{2} + \cos(\pi \cdot t) \cdot \sin(z) \\ \cos(\pi \cdot y') & \approxeq & \cos(\pi \cdot t) - \cos(\pi \cdot t) \cdot \frac{z^2}{2} - \sin(\pi \cdot t) \cdot \sin(z) \end{cases} \tag{9.23}$$

An illustration of the method is presented in Figure 9.9.

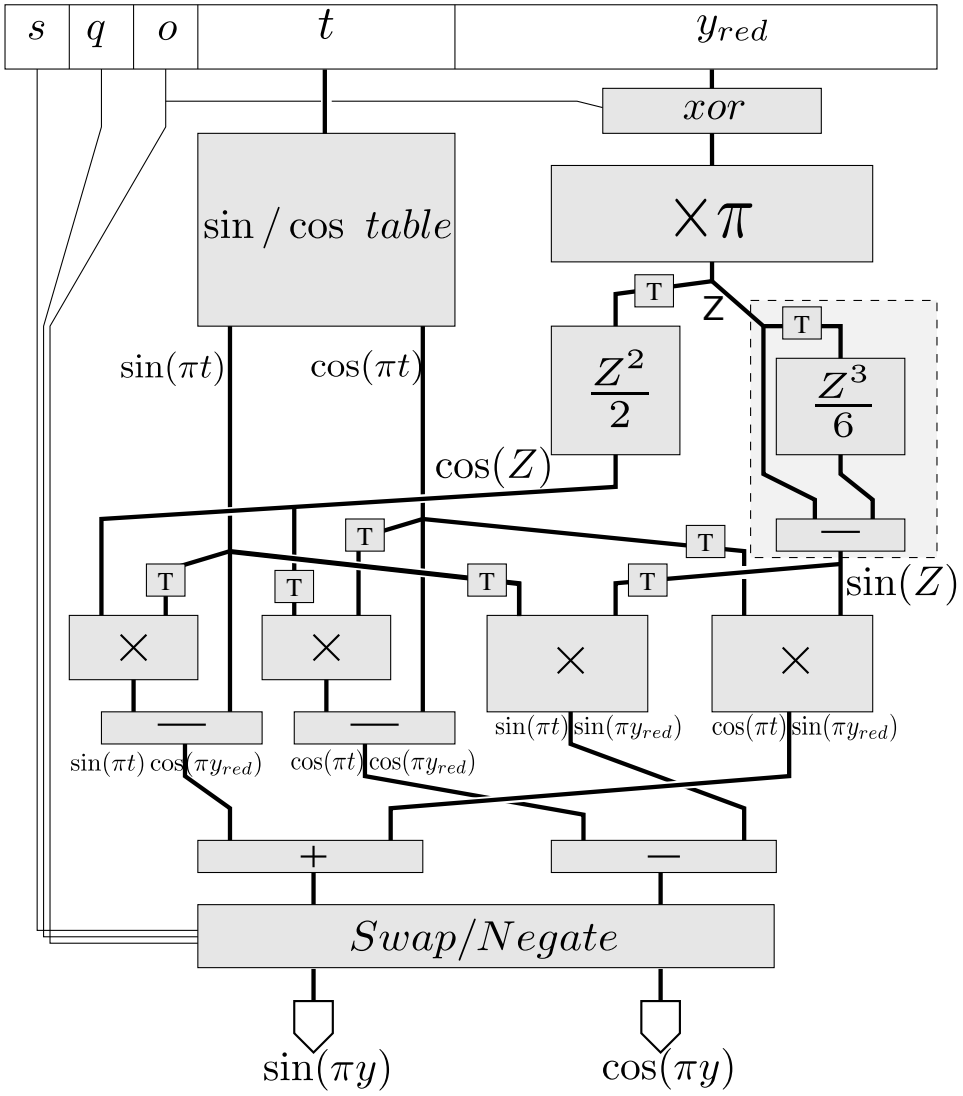


Figure 9.10: A possible implementation for the tabulate and multiply method for computing the sine and cosine

A possible implementation of the method of Section 9.4 tuned for an efficient mapping on FPGAs is presented in the following.

The first remark is that the term $y' = \frac{1}{4} - y$ can actually be obtained as the one's complement of y ($\neg y$). This has the disadvantage of introducing an error, managed in the error analysis below. As an advantage, it avoids one extra addition and a carry propagation.

Figure 9.10 shows an illustration of the proposed implementation. As can be seen in the figure, the datapaths are sized so as to compute just right, therefore satisfying the accuracy constraints when

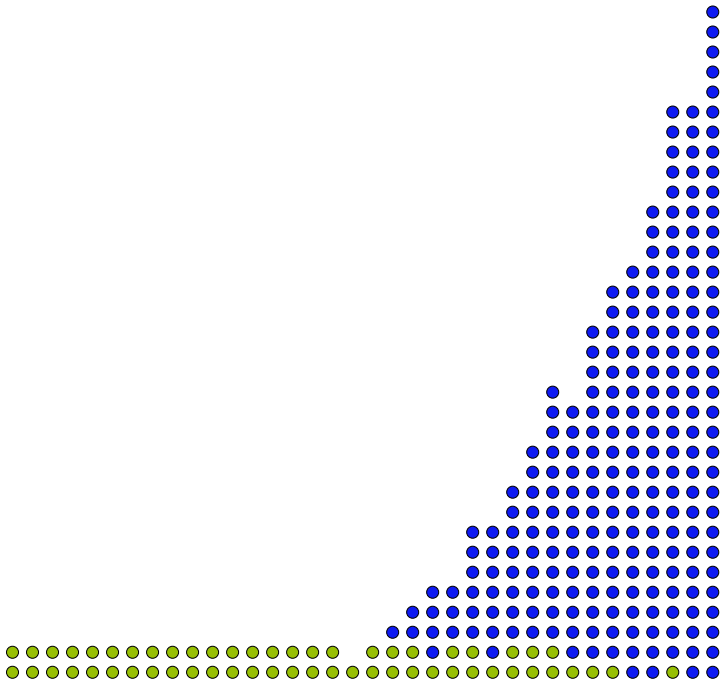


Figure 9.11: A bitheap corresponding to $z - \frac{z^3}{6}$, for a sine/cosine architecture with the input on 40 bits

using the least amount of resources. The implementation uses truncated multipliers. Also, the inputs to the multipliers are truncated to the the accuracy of the results, which is indicated in Figure 9.10 by the gray squares that contain a T inside.

Concerning the computation of $\sin(\pi \cdot y_{red})$ and $\cos(\pi \cdot y_{red})$, the term z^2 is obtained by using a dedicated squarer or a table, which is more efficient than the equivalent multiplier.

In order to compute $\sin(z) = z - \frac{z^3}{6}$, two approaches are possible for the $\frac{z^3}{6}$ term. As only a few bits are needed, for small sizes of z tabulation can be used. Currently, this is done for z up to 10 bits. For larger sizes, the architecture described in Chapter 6, on page 72, for illustrating the opportunities brought about by bitheaps can be used. In Figure 9.10 this is shown as the dashed box, filled with gray. It can be implemented as a bitheap, an example of which is shown in Figure 9.11.

Table 9.2 presents a breakdown of the required precisions for sizing the datapath in the implementation for Figure 9.10, considering an input and output of size w. The $\frac{z^2}{2}$ column also defines the size of the two left multipliers on the figure. The y_{red} column defines the size of the two rightmost multipliers. All the data of Table 9.2 is computed by the operator generator.

w	a	y_{red}, z $\cos(\pi \cdot t) \cdot \sin(z), \sin(\pi \cdot t) \cdot \sin(z)$	$\frac{z^2}{2}$	$\frac{z^3}{6}$
16	4	16	11	6
24	6	22	15	8
32	8	28	19	10
40	10	34	23	12
48	12	40	27	14

Table 9.2: Internal precisions needed by the architecture of Figure 9.10

An Error Analysis For the Table and Multiplier-Based Parallel Polynomial Method

This section analyzes the possible errors produced during the computations of the sine and cosine functions with the method of Section 9.4. As in the case of Section 9.3.1, the errors can be split into two categories: method errors and rounding errors. The former have already been analyzed, to determine the value of the parameter a.

Concerning the rounding errors, there are, again two sources: the final rounding to the target output precision (which is of the order of 2^{-w-1}) and the computations that are part of the datapath. Each of the results that are obtained from a table introduce an error that is bounded by 2^{-w-g-1}, on an intermediary accuracy using g extra guard bits. Every truncation of the inputs to the multipliers, shown as the gray boxes with Ts inside the in Figure 9.10, introduce an error of the order of 2^{-w-g}. The same is true for the truncated multipliers and the squarer. Finally, computing y' as $\neg y$ and omitting the carry propagation introduces an error of the order of 2^{-w-g}. Adding all the errors together, it can be seen that there is a bound of $15 \cdot 2^{-w-g}$ on the total error. Therefore, a number of $g = 4$ extra guard bits suffices for the implementation of the datapath. It should be noticed that the value of g no longer depends on the input size w, contrary to CORDIC.

The Generic Polynomial Approximation-Based Method

The final method that is explored for computing the sine/cosine functions is based on the generic polynomial evaluation technique presented in [35]. This operator is capable of computing either the sine, or the cosine functions. The choice between the two is done using an extra input in the architecture: when it is set, the sine function is computed, else the output is the cosine function of the input. An illustration of the polynomial evaluation on which the method is based is shown in Figure 9.12.

This method uses a slightly different argument reduction. The input is reduced only to a quadrant. Therefore, $y' \in [0, \frac{1}{2})$, and the output of the operator is reconstructed using $\sin(\pi \cdot y')$ and the

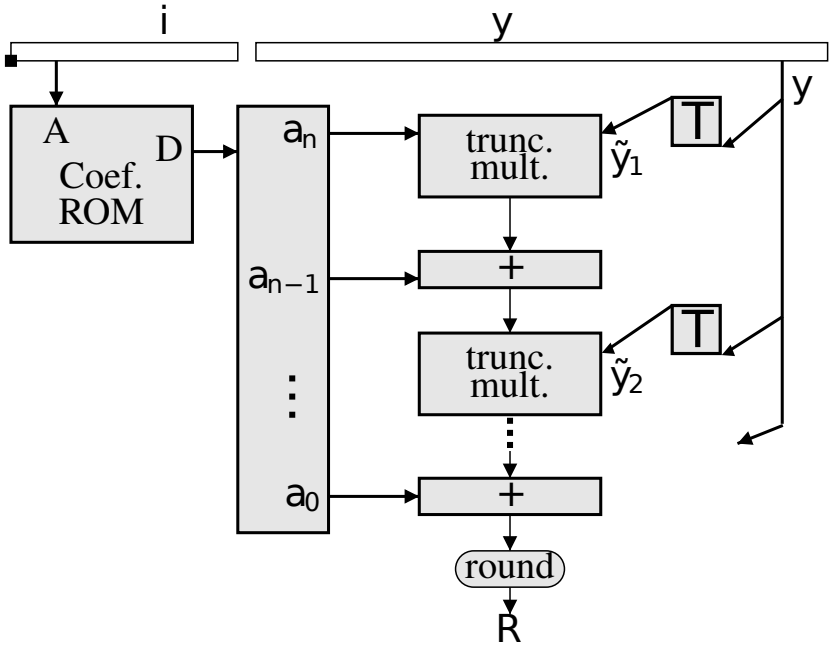


Figure 9.12: The generic polynomial evaluator-based method

bits s, q and the extra input which is denoted as $\overline{\sin\cos}$.

Results and Discussions on Methods for Computing the Sine and Cosine Functions

All the methods presented throughout Chapter 9 are implemented as new operators in FloPoCo's library. In the following, the method of Section 9.3 is denoted as *CORDIC* (prefixed by *Red.* for the reduced iterations version), the method of Section 9.4 is denoted as *SinAndCos*, and finally the method of Section 9.5 is denoted as *SinOrCos*.

Approach	latency	area
CORDIC 16 bits	30.3 ns	1034 LUTs
SinAndCos 16 bits	15.0 ns	1211 LUTs
CORDIC 24 bits	44.6 ns	2079 LUTs
SinAndCos 24 bits	17.0 ns	2183 LUTs
CORDIC 32 bits	62.1 ns	3513 LUTs
SinAndCos 32 bits	19.4 ns	3539 LUTs

Table 9.3: Synthesis results for logic-only implementations

Approach	latency	frequency	Reg. + LUTs	BRAM	DSP
CORDIC	18	478	969 + 1131	0	0
CORDIC	14	277	776 + 1086	0	0
CORDIC	7	194	418 + 1099	0	0
CORDIC	3	97	262 + 1221	0	0
Red. CORDIC	13	368	625 + 719	0	2
Red. CORDIC	4	238	106 + 713	0	2
SinAndCos	4	298	107 + 297	0	5
SinAndCos	3	114	168 + 650	0	2
SinOrCos (d=2)	9	251	136 + 183	1	2
SinOrCos (d=2)	5	115.3	87 + 164	1	2

Table 9.4: Synthesis Results on Xilinx Virtex5 FPGA, Using ISE 12.1, for 16 bit input precision

The first comparison being is made to answer a fundamental question: what is the cost of computing the sine/cosine functions on an FPGA? In order to answer this question, the *SinAndCos* and the *CORDIC* methods are compared, both of them generated using only the look-up tables available on the target device. For the *SinAndCos* method, a combinatorial version of the operator is generated, where the tables use the LUTs, and not the embedded block memories, and the multipliers use the truncated soft multipliers available in FloPoCo. The two approaches are comparable: they have the same inputs, the same outputs and achieve the same accuracy; they are functionally equivalent, yet they take two different approaches. Comparing the results in Table 9.3, it can be observed that the two approaches required approximately the same number of LUTs to be implemented. This is, indeed, a very interesting result. However, the *SinAndCos* method has a considerably lower latency, all throughout the range of precisions. It can happen, therefore, that the unrolled CORDIC implementation presents no interest over the multiplier-based methods. An iterative CORDIC implementation is still relevant, though, as the *SinAndCos* method cannot be used in such a way. It should also be mentioned that the *CORDIC* implementation is very much on par with the vendor provided cores of [3], used in parallel mode.

Table 9.4 illustrates the versatility of all the operators described throughout Chapter 9. The input size is set to 16 bits. It can be seen that various frequencies can be attained, trading a faster architecture for longer latencies. *Red. CORDIC* is a very interesting compromise, as it reduces the number of iterations to approximately half, all for the cost of two multiplier blocks, for precisions of up to 32 bits.

New families of devices are pushing even further the number of available embedded multipliers and block memories, with each new generation of devices. Methods that exploit these features are becoming more relevant. Table 9.5 illustrates the compromises that can be made when implementing

Approach	latency	frequency	Reg. + LUTs	BRAM	DSP
precision = 40 bits					
SinAndCos (bit heap)	11	266	895 + 1644	3	12
SinAndCos (table $z^3/6$)	8	232	500 + 949	4	12
SinAndCos (bit heap)	4	154	612 + 2826	0	12
SinAndCos (table $z^3/6$)	4	156	395 + 2268	2	12

Table 9.5: Trade-ofs for implementing $\frac{z^3}{6}$. Synthesis Results on Xilinx Virtex5 FPGA, Using ISE 12.1

the $\frac{z^3}{6}$ term, in the implementation of the *SinAndCos* method. As can be seen, the choice between the architectures can be done by the designer based on a trade-of between the available resources.

Finally, Tables 9.6 and 9.7 present a comprehensive set of results (for precisions between 16-32 bits and 40-48 bits, respectively), comprising all the methods, for various precisions and various target frequencies. As the precision increases, the multiplier- and table- based methods become more relevant. The lower maximum frequencies of the *SinAndCos* and *SinOrCos* methods are due to the a complex pipeline, and do not reflect some intrinsic limits of the algorithms themselves. Once all the features of the target devices are fully exploited, such as the registers inside the embedded multiplier blocks, similar frequencies can be achieved. A remark that has to be made is that when comparing the *SinAndCos* and *SinOrCos* methods, the former computes both functions, while the latter computes only one of them. So, two instance of *SinOrCos* are required in order to output the same results as *SinAndCos*. The gains of computing both functions at once therefore becomes evident.

Approach	latency	frequency	Reg. + LUTs	BRAM	DSP
precision = 16 bits					
CORDIC	18	478	969 + 1131	0	0
CORDIC	14	277	776 + 1086	0	0
CORDIC	7	194	418 + 1099	0	0
CORDIC	3	97	262 + 1221	0	0
Red. CORDIC	13	368	625 + 719	0	2
Red. CORDIC	4	238	106 + 713	0	2
SinAndCos	4	298	107 + 297	0	5
SinAndCos	3	114	168 + 650	0	2
SinOrCos (d=2)	9	251	136 + 183	1	2
precision = 24 bits					
CORDIC	28	439.9	1996 + 2144	0	0
CORDIC	23	251.0	1721 + 2114	0	0
CORDIC	12	180.5	919 + 2146	0	0
CORDIC	5	103.8	442 + 2089	0	0
Red. CORDIC	20	273.4	1401 + 1446	0	4
Red. CORDIC	11	222.6	674 + 1438	0	4
Red. CORDIC	5	222.6	224 + 1470	0	2
SinAndCos	5	262	197 + 441	3	7
SinAndCos	3	179	193 + 472	1	7
SinOrCos (d=2)	9	251	202 + 279	2	2
SinOrCos (d=2)	7	180	178 + 278	2	2

Table 9.6: Comparison of the three techniques on Virtex-5. Small precisions.

Approach	latency	frequency	Reg. + LUTs	BRAM	DSP
precision = 32 bits					
CORDIC	37	403.5	3495 + 3591	0	0
CORDIC	32	230.0	3120 + 3559	0	0
CORDIC	16	162.4	1532 + 3509	0	0
CORDIC	7	86.6	698 + 3508	0	0
Red. CORDIC	24	256.8	2160 + 2234	0	4
Red. CORDIC	15	217.1	1149 + 2295	0	4
Red. CORDIC	7	112.1	618 + 2225	0	4
SinAndCos	10	253	535 + 789	3	9
SinAndCos	4	110	311 + 996	1	9
SinOrCos (d=3)	14	251	444 + 536	4	5
SinOrCos (d=3)	9	146	306 + 534	4	5
precision = 40 bits					
CORDIC	45	375	5070 + 5289	0	0
CORDIC	25	149	2948 + 5245	0	0
Red. CORDIC	37	252	3695 + 3768	0	8
Red. CORDIC	22	211	2438 + 3476	0	8
Red. CORDIC	9	123	931 + 3339	0	8
SinAndCos (bit heap)	11	266	895 + 1644	3	12
SinAndCos (table $z^3/6$)	8	232	500 + 949	4	12
SinAndCos (bit heap)	4	154	612 + 2826	0	12
SinAndCos (table $z^3/6$)	4	156	395 + 2268	2	12
SinOrCos (d=3)	15	251	628 +725	4	8
SinOrCos (d=3)	9	132	376 +675	4	8
precision = 48 bits					
SinAndCos (bit heap)	13	232	1322 + 2369	12	17
SinAndCos (bit heap)	6	132	972 + 2133	12	17
SinOrCos	15	250	734 + 879	17	10
SinOrCos	9	124	431 + 823	17	10

Table 9.7: Comparison of the three techniques on Virtex-5. Large precisions.

10
Atan2

The **atan2** function is widely used in digital signal processing. Examples of applications include recovering the phase of a signal, or rectangular to polar coordinates conversion (computing the polar angle $\arctan(\frac{y}{x})$, of a number given by its coordinates (x,y)). In the following, the implementation of atan2 with fixed-point inputs and outputs is studied.

The classical CORDIC shift-and-add algorithm is compared to two multiplier-based techniques. The first method computes the atan2 function using two single-variable functions: the reciprocal and the arctangent, each evaluated using bipartite or polynomial approximation methods. The second method uses a bivariate piecewise polynomial approximation of degree one or two. Each of the method benefits from an argument reduction, which is also discussed. All the algorithms produce faithfully rounded results.

Definitions and Notations

The function which is implemented in the following is $\text{atan2}(y,x) = \arctan(y/x)$. It is part of the standard mathematical library and returns an angle $\alpha \in [-\pi,\pi]$. A difference between it and the composition of the division and the arctan function (that returns an angle in $[-\pi/2,\pi/2]$) is that $\text{atan2}(y,x)$ keeps track of the respective signs of x and y. It is used to compute the phase of a complex number $x+iy$, as shown in Figure 10.1.

The inputs and outputs of the function are both fixed-point and on w bits.

A straightforward simplification of the fraction can be used, $\arctan(\frac{ky}{kx}) = \arctan(\frac{y}{x})$, therefore the range of the inputs is not really relevant. It is important, however, that the inputs x and y both

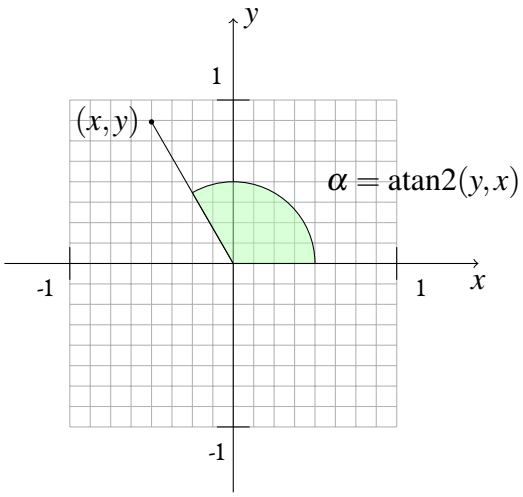


Figure 10.1: Fixed-point $\arctan(\frac{y}{x})$

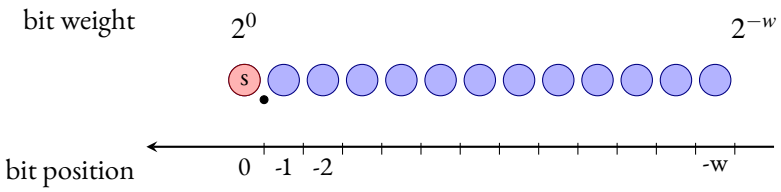


Figure 10.2: The bits of the fixed-point format used for the inputs of atan2

have the same format. Both inputs are fixed-point numbers in the range $[-1,1)$, represented classically in two's complement, as shown in Figure 10.2.

There are two possible ways for expressing the output:

- in radian, from $-\pi$ to π
- or in $[-1,1)$, in which case the function being computed is $\frac{1}{\pi}\text{atan2}(y,x)$. This option will be referred to as "binary angles" in the following.

There are several reasons for preferring binary angles when using fixed-point notations. The first is that it fully uses the representation space. An output on $[-\pi,\pi]$ requires a fixed-point format which represent the interval $[-4,4)$. This leaves the format underutilized, as all the codes between π and 4 (and $-\pi$ and -4, respectively) are never used.

The second reason is that it simplifies the implementation: the modulo-2π periodicity of radian angles is mapped to the modulo-2 periodicity of the two's complement binary arithmetic. Take, for example, the slight asymmetry of the two's complement output range $[-1,1)$ (where -1 is representable on the format, but 1 is not). It is not a problem, as the modulo-2 arithmetic maps the angle

$1 \times \pi$ to the angle $-1 \times \pi$. This allows for savings to be made during the range reduction. It also allows it to be exact, compared to radian angles which require computations with the constant $\pi/2$ that cannot be represented exactly in binary.

A third reason for advocating the use of binary angles is that it simplifies the application context. In QPSK decoding (commonly used in telecommunications), for example, the atan2 of incoming samples are averaged to estimate a phase shift. Computing this average is simpler with binary angles than with radian angles, as it is a modulo operation.

Therefore, in this chapter the focus is on the function:

$$f(x,y) = \frac{1}{\pi}\,\text{atan2}(y,x)$$

sometimes called atan2Pi. However, adapting all these algorithms to radian angles is possible.

Following FloPoCo's philosophy, the objective of this chapter is to compute the function with last-bit accuracy. The target function is computed with last-bit accuracy, is less than the weight of the LSB of the result. Another option would be correct rounding, but it may require three times the internal precision [92], which is not justified in the current context.

Overview of the Proposed Methods For Hardware Implementation of the Atan2

To ensure a meaningful comparison between the different methods, each technique is implemented with the same level of dedication towards the possible optimizations, and with the same accuracy objective.

The most classical technique for fixed-point hardware implementation of atan2 is **CORDIC**, introduced in Section 9.3, but used here in the **vectoring mode**. This approach is studied in Section 10.4, which presents an error and range analysis that guarantees last-bit accuracy at the lowest cost in terms of the implementation.

With the experience of the techniques of Chapter 9, the first approach that might spring to mind is the tabulation of all the values of atan2 in a table addressed by a concatenation of x and y. This would result, however, in a table with 2^{2w} entries, which accelerates even further the exponential growth of the table, and quickly becomes unmanageable, even for small values of w. It should also be remarked that classical table-compression techniques, like the bipartite or multipartite methods of Section 11.4 cannot be applied, since they are developed for single variable functions, whereas atan2 is a function of two variables.

The next method considered, studied in Section 10.5 and illustrated in Figure 10.8, tries to overcome these shortcomings and therefore reduces atan2 to two functions of one variable, evaluated in a sequence. These are the reciprocal $1/x$, used to compute the division $z = \frac{y}{x}$, and the arctangent, used

to compute $\arctan(z)$. This method will be referred to as RecipMultAtan in the following. There are a multitude of methods throughout the literature which can be used for computing these single variable functions. There is, however, the problem that the output of the reciprocal z can become very large, as the input x approaches 0. One solution would be the use of a floating-point format, or a very large fixed-point format for z, but these are both impractical solutions. This can be avoided by using a scaling-based range reduction, described in Section 10.3.

The last method considered, and the most original one, presented in Sections 10.6 and 10.7, is the use of a piecewise approximation by bivariate polynomials. The presented experiments are limited to polynomials of the first and second degree because the number of multipliers in bivariate polynomials increases quadratically with the degree of the polynomial. As such, $\text{atan2}(y,x)$ can be approximated by the polynomials $P_1(y,x) = ax + by + c$ and $P_2 = ax + by + c + dx^2 + ey^2 + fxy$, respectively. These methods are referred to as Taylor 1 and Taylor 2, in the following. Their architectures are depicted by Figures 10.11 and 10.12.

Some interesting comparisons can be made regarding the asymptotic complexity of each of the approaches, with respect to the output precision. The area and delay of CORDIC are known to be quadratic in the precision. The other methods are table-based: their area is exponential in the precision. However their delay should be sub-quadratic, with the bivariate approach having shorter latency, but larger tables. One of the objectives is, thus, to study the relevant precision domains of each of the methods, in Section 10.8.

This work presented is designed specifically with FPGAs in mind, and integrated as new operators in the FloPoCo framework.

Range reductions

In the following, the inputs and the outputs are fixed-point numbers in $[-1,1)$ on w bits. The notations of Section 2.2.1 are used. As such, the inputs/outputs are on the format $\text{sfix}(0,-w+1)$: the sign bit has weight 2^0 and the LSB has weight 2^{-w+1}.

Parity and Symmetry

The arctan is an odd function, and this property can be exploited to easily reduce the inputs to the positive quadrant. There is a symmetry between x and y:

$$\arctan(y/x) = \pi/2 - \arctan(x/y) \tag{10.1}$$

If $y > x$, the two inputs x and y can be swapped, so the computation is reduced to the first octant ($x \geq 0,\ y \geq 0,\ y < x$). Similar range reduction formulas can be used on the other quadrants. Figure

$$\begin{cases} x < 0 \\ y < 0 \\ y < x \end{cases}$$

$$\Rightarrow \text{atan2}(y,x) = -\tfrac{\pi}{2} - \text{atan2}(|x|,|y|)$$

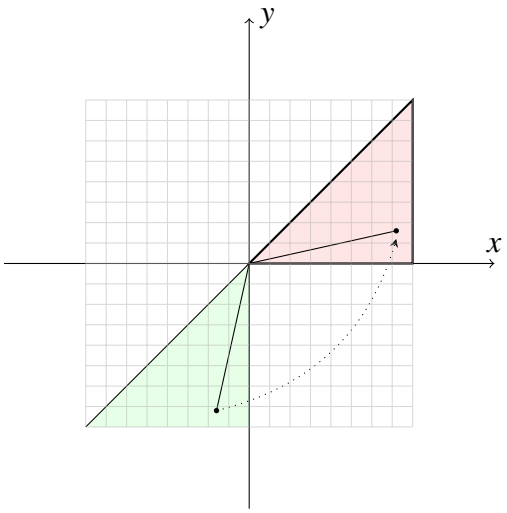


Figure 10.3: One case of symmetry-based range reduction. The 7 other cases are similar.

10.3 exemplifies one such process.

This type of argument reduction is much easier when using binary angles. For example, the final addition with $\frac{\pi}{2}$ becomes an addition of $\frac{1}{2}$ to a number in $[-1,1)$. This is an exact operation, with a carry propagation of only one bit. On the other hand, adding $\frac{\pi}{2}$ would require a full-width carry propagation, and entail a systematic error due to the rounding of the irrational number $\frac{\pi}{2}$.

In order to obtain the absolute values $|x|$ and $|y|$, the negative input values must be negated. One exception is the value -1, which has no opposite in two's complement arithmetic, and must be saturated. There are two possible implementations for this operation: a full-size subtraction, which makes them exact, or a bitwise complement, which is smaller and has lower latency, but comes at the cost of an error of the order of magnitude of the last bit. The two options are relevant, depending on the subsequent algorithm.

Scaling range reduction

The value of $z = \frac{y}{x}$ can become very large due to the division by x, as x goes to 0. On the left side of the plot of atan2 over a quadrant, shown in Figure 10.4, this translates to much larger slopes when y is small.

This is problematic for RecipMultAtan and Taylor 1 and 2 methods. This range reduction can be achieved using an architecture similar to the one depicted in Figure 10.5 [51].

The architecture consists of first counting the number of leading zeros which are in common between x and y. This amount is denoted s. Then, both x and y can be scaled up by a shift left of s

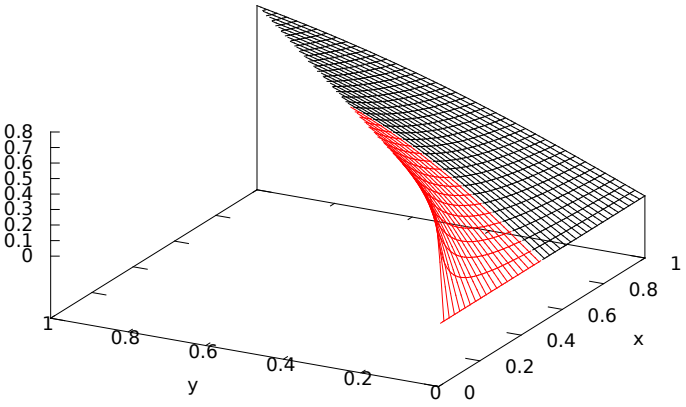


Figure 10.4: 3D plot of atan2 over the first octant.

bits, using the identity:

$$\arctan\left(\frac{2^s \cdot y}{2^s \cdot x}\right) = \arctan\left(\frac{y}{x}\right)$$

which also implies that the resulting x satisfies $x \geq \frac{1}{2}$.

The CORDIC Method

The CORDIC method for computing the arctangent was introduced in [125], together with the method for the computation of the sine and cosine, discussed in Chapter 9. Another relevant and comprehensive reference is [92]. In the vectoring mode, the algorithm is initialized as:

$$\begin{cases} x_1 &= x_r \\ y_1 &= y_r \\ \alpha_0 &= 0 \end{cases} \tag{10.2}$$

with the main iteration being:

$$\begin{cases} x_{i+1} &= x_i - 2^{-i} s_i y_i \\ y_{i+1} &= y_i + 2^{-i} s_i x_i \\ \alpha_{i+1} &= \alpha_i - s_i \arctan 2^{-i} \end{cases} \tag{10.3}$$

where s_i is chosen so as to answer the comparison $y_n \leq 0$ and therefore is defined as $s_i = -\text{sgn}(y_i)$.

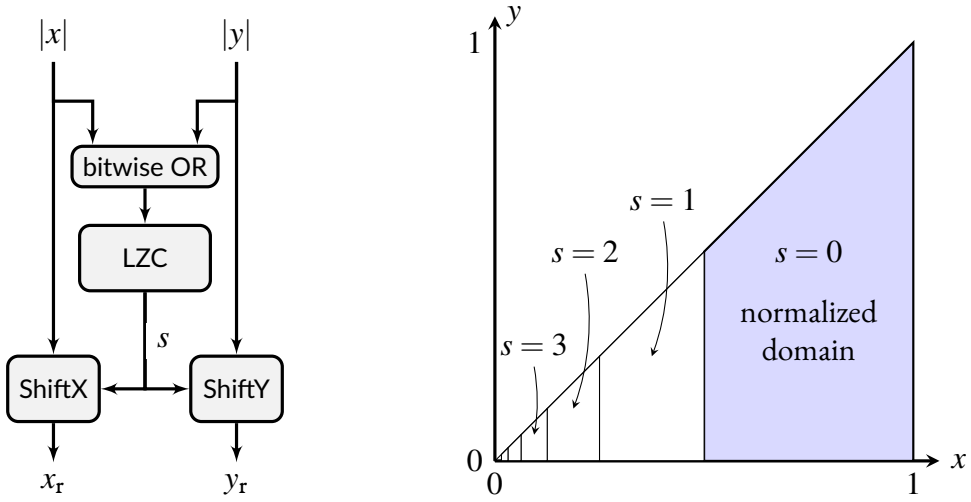


Figure 10.5: Scaling-based range reduction

The iteration in Equation 10.3 converges as follows:

$$\begin{cases} x_i & \longrightarrow & K\sqrt{x^2+y^2} & \text{, with } K = \prod_{i=1}^{\infty}\sqrt{1+2^{-2i}} \approx 1.1644... \\ y_i & \longrightarrow & 0 & \\ \alpha_i & \longrightarrow & \arctan\frac{y_r}{x_r} & \end{cases} \tag{10.4}$$

The method is designed to work with radian angles, but can be modified to produce a binary angle if the constants $\arctan 2^{-i}$ are expressed in this format. As a small optimization, the iterations can start from iteration 1 instead of iteration 0, thanks to the initial reduction of the argument to the $[0, \pi/4]$ octant. This also leads to a smaller value for the K scaling factor, than the one found in most textbooks.

A choice was made to ignore CORDIC variants which use carry-save or other forms of redundant arithmetic in order to accelerate the iterations. Several overviews of these methods can be found in works such as [92], [13] or [84], to name only a few. The main reason for this choice is connected to the fact that addition carry propagation is very fast on FPGAs, as compared to generic logic routing. This choice is also supported by the data analysis in Section 10.8.

In the following, the unrolled implementation is discussed, which, once pipelined, produces one result each cycle. An illustration of such an architecture is shown in Figure 10.6.

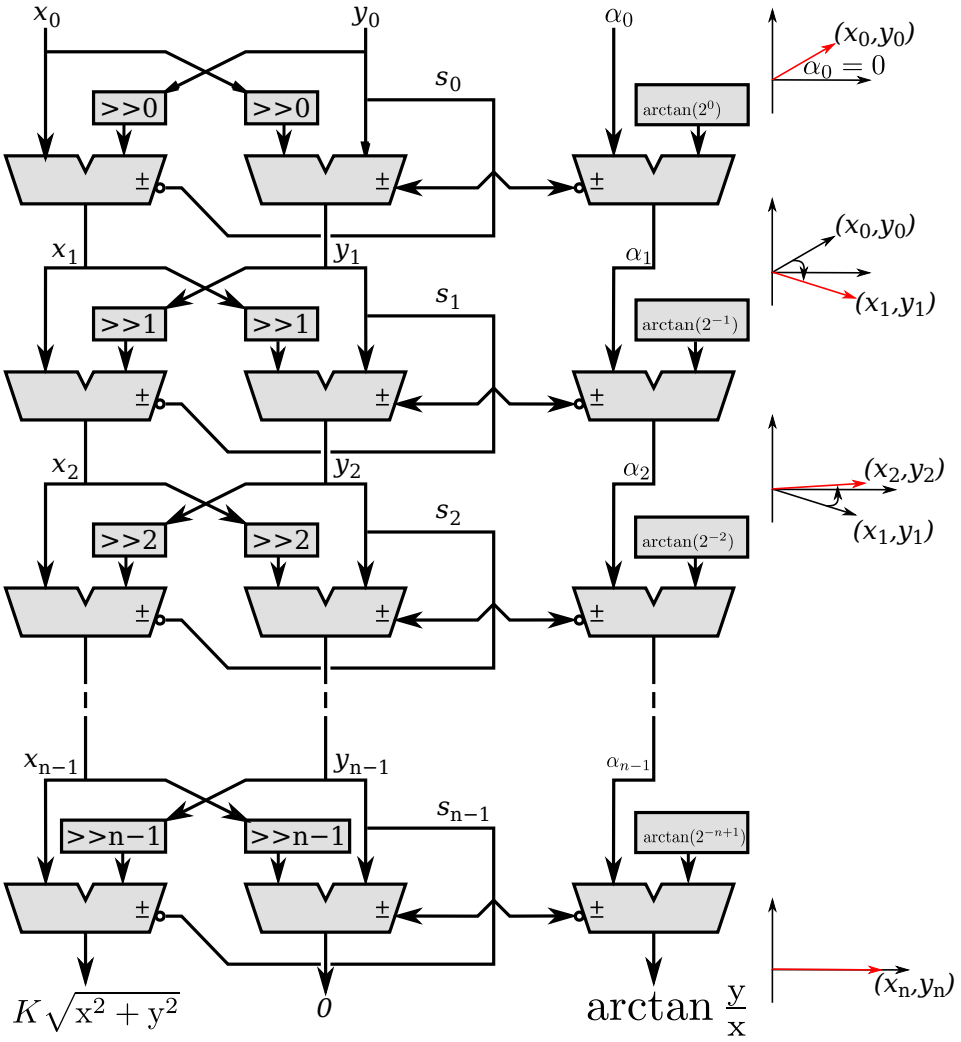


Figure 10.6: An implementation of the unrolled CORDIC method in vectoring mode

Error analysis and datapath sizing

Computing the iteration of Equation 10.3 with infinite accuracy, produces a method error ε_{method} on α_i which is smaller than 2^{-i} radians, after i iterations [92]. In order to make this error on the binary angle smaller than 2^{-w} (*i.e.* smaller than half of an ulp of the result), the iterations can be stopped at iteration $w-1$.

The discussions are first focused on rounding errors, and more generally on fixed-point implementation issues. The weight of the least significant bit on the x_i and y_i datapath is denoted p (this will be $p=-w-g$, where g is the number of extra guard bits). In order to define the fixed-point format for the x_i datapath, an observation can be made that $0 \leq x_i < K\sqrt{2} \approx 1.646$. This gives the weight

of the most significant bit of x_i, and means that a ufix$(1,-p)$ format can be used.

On the y_i datapath, the following relation holds: $y_i < K \sin \alpha_i$. The bound on $\alpha_i \leq 2^{-i}$ results in $|y_i| < 2^{-i+1}$, which gives the most significant bit of y_i. Therefore, a sfix$(-i+2,-p)$ format can be used for y_i.

With the data formats determined, the rounding errors performed when implementing the CORDIC iterations can be analyzed next. In terms of notations, the values which are actually computed are denoted $\tilde{x}_i$ and $\tilde{y}_i$. The corresponding errors are define as $\varepsilon_i^x = \tilde{x}_i - x_i$, and $\varepsilon_i^y = \tilde{y}_i - y_i$ respectively. The errors introduced in each iteration, u_i^x and u_i^y, are due to the shift operations, which cause some of the least significant bits to be discarded. They can be expressed as:

$$\begin{aligned}
\tilde{x}_{i+1} &= \tilde{x}_i - s_i 2^{-i} \tilde{y}_i + u_i^x \\
&= x_i + \varepsilon_i^x - s_i 2^{-i}(y_i + \varepsilon_i^y) + u_i^x \\
&= x_{i+1} + \varepsilon_i^x - s_i 2^{-i} \varepsilon_i^y + u_i^x
\end{aligned}$$

from which a recurrence formula for the accumulation of errors on the x_i datapath can be deduced:

$$\varepsilon_{i+1}^x = \varepsilon_i^x - s_i 2^{-i} \varepsilon_i^y + u_i^x$$

Using a common upper bound $\overline{\varepsilon}_i$ for both ε_i^x and ε_i^y, and the bound 2^{-p} for u_i^x, the recurrence relation becomes:

$$\overline{\varepsilon}_{i+1} = \overline{\varepsilon}_i (1 + 2^{-i}) + 2^{-p} \tag{10.5}$$

The initial error is $\overline{\varepsilon}_1 = 0$, when using a carry propagation while computing the negation in the range reduction. If a simple one's complement is used, the error increases to $\overline{\varepsilon}_1 = 2^{-p}$. This only increases the overall error by a small amount.

In order to ensure a faithfully rounded result, on the output precision, the recurrence relation of Equation 10.5, is computed up to $i = w-1$. The precision of the x_i and y_i datapaths can then be defined as $p = w - 1 - \lceil log_2 \overline{\varepsilon}_{w-1} \rceil$. This ensures that y_i is computed accurately enough so as to ensure that the rounding errors do not change its sign in a way that cannot be corrected by CORDIC.

There is also a small optimization that can be made on the y_i datapath. Considering the fact that $|y_i| < 2^{-i+1}$, it follows that the term $2^{-i} y_i$ which is added or subtracted to x_i is smaller than 2^{-2i+1}. Therefore, the iterations on x_i can be stopped as soon as $2i - 1 > p$.

The error analysis on the α_i datapath is quite straightforward. The errors are introduced by the $\arctan 2^{-i}$ terms. If these terms are correctly rounded to the precision p_α, they each contribute an error which is bounded by $2^{-p_\alpha - 1}$. The final rounding introduces an error bounded by 2^{-w}. Therefore, the computations need to be performed using g_α extra guard bits, which can absorb these er-

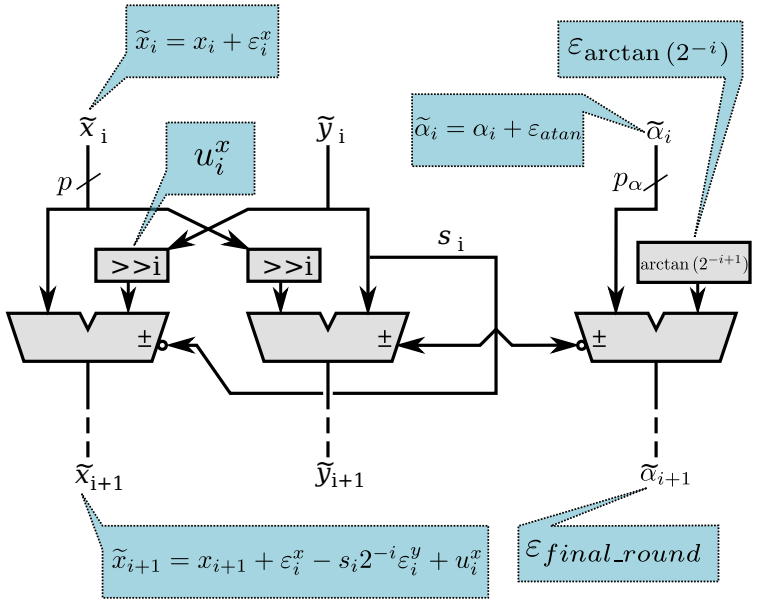


Figure 10.7: The errors on the x_i, y_i and α_i datapaths

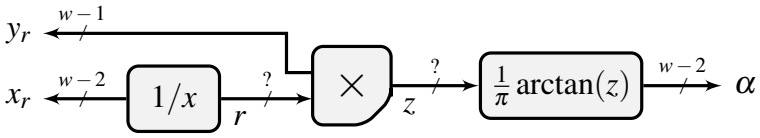


Figure 10.8: Architecture based on two functions of one variable

rors, where $g_\alpha = 1 + \lceil log_2((w-1) \times 0.5) \rceil$.

Another small optimization is that the final rounding can be replaced by a truncation, as long as 2^{-w} is added to one of the $\arctan 2^{-i}$ constants.

The Reciprocal-Multiply-Tabulate Method

The second method considered is a classical one, and comes quite natural. It is illustrated in Figure 10.8 and requires the prior reduction of the inputs (x_r and y_r represent the reduced inputs in the figure). After this scaling $0.5 \leq x_r \leq 1$, and since the input is transposed to the first octant, it can be remarked that $y \leq x$, and therefore $z = \frac{y}{x} \in [0, 1]$. On these input ranges, the reciprocal as well as the arctangent are regular, as can be seen in Figure 10.9. They can be approximated using tabulation or polynomials. It remains to determine the size of the datapath, given by the bitwidths of r and z, in such a way so as to ensure a faithful result at the minimal cost.

In terms of implementation, it is again convenient to use binary angles. The output of the block computing the arctangent in Figure 10.8 belongs to $[0, 1/4]$, so the ufix$(-2, -w+1)$ format can be

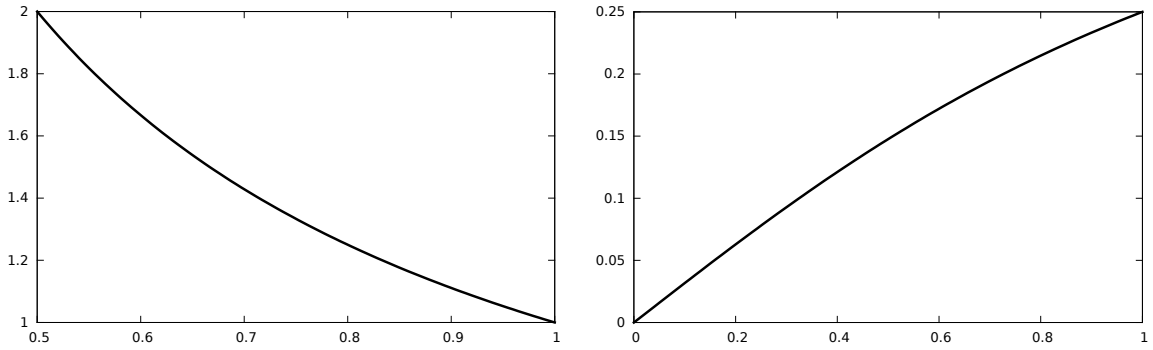


Figure 10.9: Plots of $\frac{1}{x}$ on $[0.5, 1]$ (left) and $\frac{1}{\pi}\arctan(z)$ on $[0, 1]$ (right)

used. The two most significant bits of the result can be added, error-free, during the reconstruction stage.

Related work

The method of Figure 10.8 is classically used in software [48, 116]. The division $z = \frac{y}{x}$ can be computed in the working precision, followed by a high-degree, odd polynomial, which is used for the accurate approximation of $\arctan(z)$.

In terms of methods designed for hardware, some early works use piecewise linear approximations of unspecified accuracy for the computation of $\arctan(z)$ to some precision [104, 106]. In the context of FPGAs, this technique is used in [51] with multipartite tables [39] to remove the need for multipliers. Comparisons to the CORDIC method are shown for 12-bit precision; their main outcome seems to be a reduction of the power consumption to nearly half. Similarly, the accuracy of the architecture is only measured for 12-bit precision (10.3 bits in the worst case). In a follow-up article [50], the division is replaced by a subtraction in the logarithmic domain. However the architectural requirements are changed to three function evaluations: two logarithmic conversions on the input, and one computation of the $\arctan(2^z)$. For the evaluation of $\arctan(2^z)$, a non-uniform decomposition is used.

The contributions of the work presented in this chapter over [51] are:

- a generalization to more precisions
- a rigorous error analysis, so as to achieve last-bit accuracy
- polynomials of larger degrees, as in [35], are used for the approximation of the unary functions when needed.

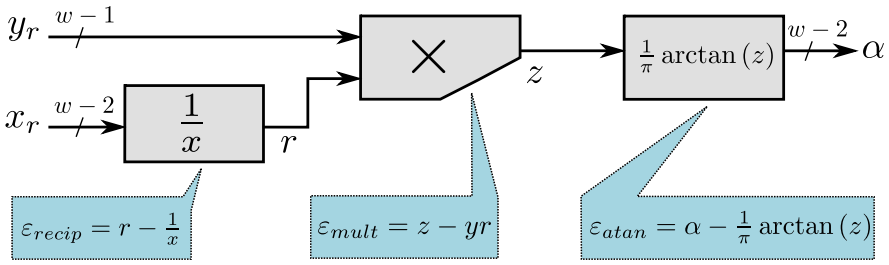


Figure 10.10: The errors on the reciprocal-multiply-tabulate method datapath

Error analysis

The errors for the computation of the reciprocal and arctangent can be defined as:

$$\varepsilon_{\text{recip}} = r - 1/x$$
$$\varepsilon_{\text{atan}} = \alpha - \frac{1}{\pi}\arctan(z)$$

Some of the least significant bits of the product $y \times \frac{1}{x}$ are truncated/rounded, so as to keep the input to the arctangent small. Therefore, the multiplier can be truncated, and its error can be defined as:

$$\varepsilon_{\text{mult}} = z - yr$$

Using the above relations, the total error on z is can be defined as follows:

$$\begin{aligned}\varepsilon_z &= z - \frac{y}{x} \\ &= z - yr \quad + \quad yr - \frac{y}{x} \\ &= \varepsilon_{\text{mult}} + y\varepsilon_{\text{recip}}\end{aligned}$$

As $0 \leq y \leq 1$ it results that:

$$|\varepsilon_z| \leq |\varepsilon_{\text{mult}}| + |\varepsilon_{\text{recip}}|$$

In order to obtain a faithfully rounded result, the total error at the output ε_{total} must satisfy

$\varepsilon_{total} < 2^{-w+1}$. The total error can be defined as:

$$\begin{aligned}\varepsilon_{\text{total}} &= \alpha - f(x,y)\\ &= \left(\alpha - \frac{1}{\pi}\arctan(z)\right) + \left(\frac{1}{\pi}\arctan(z) - f(x,y)\right)\\ &= \quad \varepsilon_{\text{atan}} \quad + \quad \varepsilon_2\end{aligned}$$

From $\arctan'(x) = \frac{1}{1+x^2}$ it can be deduced that:

$$\arctan(z+\varepsilon) - \arctan(z) \approx \varepsilon \frac{1}{1+x^2}.$$

From this, and with $z \in [0,1]$, the following inequality results:

$$\left|\frac{1}{\pi}\arctan(z+\varepsilon) - \frac{1}{\pi}\arctan(z)\right| < \frac{1}{3}\varepsilon$$

where the rounding up of $\frac{1}{\pi}$ to $\frac{1}{3}$ accounts for the higher order error terms. The previous inequality may also be observed on Figure 10.9. Applying it to $z = \frac{y}{x} + \varepsilon_z$, gives:

$$|\varepsilon_2| = \left|\frac{1}{\pi}\arctan(z) - \frac{1}{\pi}\arctan(\frac{y}{x})\right| < \frac{1}{3}|\varepsilon_z|$$

and finally

$$|\varepsilon_{\text{total}}| < \frac{1}{3}|\varepsilon_{\text{recip}}| + \frac{1}{3}|\varepsilon_{\text{mult}}| + \varepsilon_{\text{atan}} \tag{10.6}$$

Datapath dimensioning

Consider first the simpler version of the implementation, where the two functions are implemented through tabulation. The arctangent table is correctly rounded to its output format. Therefore $|\varepsilon_{\text{atan}}| \leq 2^{-w}$, and what is required for last-bit accuracy is $|\varepsilon_{\text{total}}| < 2^{-w+1}$. Unfortunately, a correctly-rounded reciprocal table cannot be used, because its output has to be in $[1,2)$, and not in $(1,2]$ when the input is in $[\frac{1}{2},1)$. What is tabulated for r is therefore the correct rounding of $\frac{1}{x} - 2^{-w}$ to the format $\text{ufix}(1,-w)$. This entails $|\varepsilon_{\text{recip}}| \leq 2^{-w}$.

It is not necessary to use a correctly rounded multiplier. A truncated multiplier, faithful to an output format $\text{ufix}(0,-w)$ for z will entail $|\varepsilon_{\text{mult}}| \leq 2^{-w}$. This combination of errors satisfies Equation 10.6. Interestingly, however, this error bound is not reached: exhaustive tests (up to $w = 12$) show that last-bit accuracy is still achieved with a truncated multiplier faithful to $\text{ufix}(0,-w+1)$.

Using a correctly rounded multiplier increases its cost, requiring the computation and the summation of all its partial products. However, it provides the same accuracy for z using one less bit. A one bit on z reduces significantly the size of the block computing the arctangent, possibly halving it. This trade-off between the multiplier and the arctangent block has not been fully studied yet.

Using a black-box polynomial approximation similar to the one in [35], or the multipartite method [39], entails that the two functions are themselves last-bit accurate. In this case, the constraint $|\varepsilon_{\text{atan}}| \leq 2^{-w}$ can no longer be ensured at a reasonable cost, due to the Table Maker's dilemma [92]. What is possible is a more accurate implementation that will ensure $|\varepsilon_{\text{atan}}| \leq \frac{3}{4} \cdot 2^{-w+1}$. An error budget of $\frac{1}{4}2^{-w+1}$ is distributed among the multiplier and the reciprocal table. With the same reasoning, using a ufix$(1, -w-1)$ format for r and a ufix$(0, -w)$ for z manages to satisfy Equation 10.6.

The analysis is implemented in the FloPoCo `FixAtan2` operator. An architecture can be built for any size w and degree d. For $d = 1$, the bipartite method or the multipartite method is used, depending on which result in the best savings.

FPGA-specific Considerations

For small precisions, the following optimizations may apply on FPGAs which contain embedded memory and multiplier blocks.

The hard multipliers compute the full product, so truncation is faithful, and correct rounding comes at the cost of only one addition which can be mapped in the post-adder included in all recent DSP blocks. The latter is relevant if it saves one hard memory block, for instance in the situations depicted below.

Memory blocks are of fixed size and dual-ported, which means they can store two tables in a single block. The largest architecture that fits in a single 20Kb block (Xilinx devices) is for $w = 11$, with a reciprocal table of $2^9 \times 11$ bits and an arctangent table of $2^{10} \times 9$ bits, both packed in a dual-port block configured as $2^{10} \times 20$. In a 36Kb block (Altera devices), the largest architecture that fits is for $w = 12$. This can implement a $2^{10} \times 12$ reciprocal table and a $2^{10} \times 10$ arctangent table. In both cases the multiplier must be correctly rounded.

First-order Bivariate Polynomial-based Method

While polynomial approximation is a fairly common approach (and Section 10.5 stands as testimony), what sets the method considered in this section apart is the use of a piecewise bivariate polynomial approximation of degree 1 or 2. We are not aware of comparable work in the literature.

The main idea is to decompose the domain of the $\arctan(\frac{y}{x})$ function illustrated in Figure 10.5 on page 149, into a grid of squares/rectangles. The sub-domains are indexed by the first few most

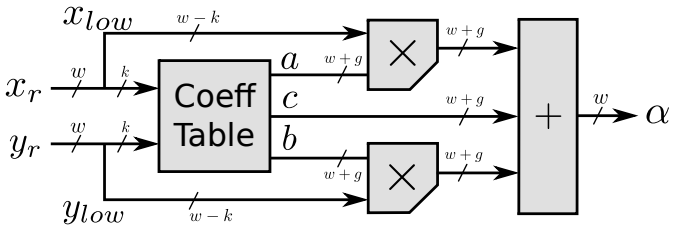


Figure 10.11: Architecture based on a first order bivariate polynomial

significant bits of x and y. Then a bivariate polynomial in the remaining bits is evaluated on each of the sub-domains.

Therefore, let x_h and y_h be the numbers formed using the k most significant bits of x_r and y_r, respectively. Also, let δx and δy denote the numbers formed of their respectively remaining $w-k-1$ least significant bits:

$$\begin{cases} x_r &= x_h + \delta x \\ y_r &= y_h + \delta y \end{cases}$$

The bits (x_h, y_h) are used to index the table of coefficients of the bivariate polynomials approximating $\text{atan2}(y,x)$, on each sub-domain. These are illustrated in Figures 10.11 and 10.12:

$$\text{atan2}(y,x) \approx P_{x_h y_h}(\delta x, \delta y)\ .$$

As can be seen from Figure 10.4, on the left part of the function's co-domain, it is beneficial to apply the scaling range reduction of Figure 10.5. This allows avoiding the problematic areas, in terms of the accuracy required for the approximation.

The first degree bivariate polynomial $P_{x_h y_h}$ is of the form:

$$P_{x_h y_h}(\delta x, \delta y) = a \cdot \delta x + b \cdot \delta y + c$$

Two different ways of obtaining the a, b and c coefficients are explored. The first uses the Taylor series approximation around point (x_0, y_0), which is the center of each of the sub-domains indexed by (x_h, y_h):

$$\begin{aligned} T_1(x_r, y_r) &\approx f(x_0, y_0) \\ &+ \frac{\partial}{\partial x} f(x_0, y_0)(x - x_0) + \frac{\partial}{\partial y} f(x_0, y_0)(y - y_0) \end{aligned}$$

The coefficients a, b and c of the polynomial can be identified by replacing $f(x,y)$ with the arctan-

gent and developing the equations:

$$\begin{aligned} a &= \frac{-y_h}{x_h^2+y_h^2} \\ b &= \frac{x_h}{x_h^2+y_h^2} \\ c &= \text{atan2}(y_h, x_h) \end{aligned}$$

The polynomial T_1 becomes:

$$T_1(x_r, y_r) = \frac{-y_h}{x_h^2+y_h^2}\delta x + \frac{x_h}{x_h^2+y_h^2}\delta y + \text{atan2}(y_h, x_h)$$

Some small optimizations can be made for the implementation noticing that the multiplications $a \cdot \delta x$ and $b \cdot \delta y$ are of reduced sizes. This is due to $\delta x = x - x_h < 2^{-k}$ (idem for δy).

The second way of obtaining the coefficients a, b and c takes a different approach, which is more geometrical. A first degree bivariate polynomial is a plane. Three points from the surface of f define a unique plane that goes through these three points. This plane is a bivariate degree-1 approximation of the function.

Therefore, if we chose three points of the form $(x, y, f(x, y))$ with (x, y) inside the square sub-domain, the equation of the plane defined by these three points provides the candidate coefficients a, b and c. The equation of the plane is given in the form:

$$\alpha \cdot x + \beta \cdot y + \gamma \cdot z + \delta = 0$$

and, thus, the value of the function is computed as:

$$\text{atan2}(y_r, x_r) = z = -\frac{\alpha}{\gamma}x_r - \frac{\beta}{\gamma}y_r - \frac{\delta}{\gamma}$$

which gives the expressions for coefficients: $a = -\frac{\alpha}{\gamma}$, $b = -\frac{\beta}{\gamma}$ and $c = -\frac{\delta}{\gamma}$.

The parameters four parameters are approximated using the k most significant bits of the inputs x_r and y_r. The same variable change is performed for x_r and y_r. The constant c must then be updated to: $c = -\frac{\delta}{\gamma} + ax_h + by_h$.

If the three points are very close to (x_0, y_0) this methods gives the Taylor coefficients. Choosing a larger triangle inside the square sub-domain actually provides a better approximation to the function.

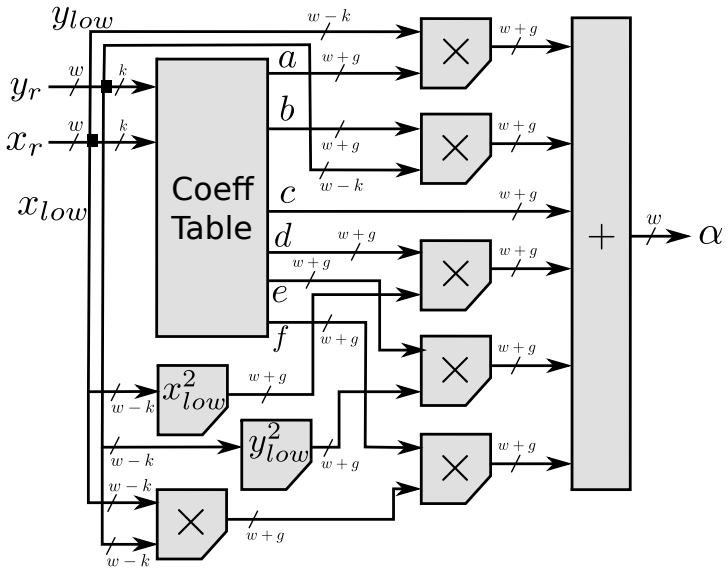


Figure 10.12: Architecture based on a second order bivariate polynomial

Second-order Bivariate Polynomial-based Method

The approach is very similar to the first-order one. The main reason for using a second-order polynomial is to obtain a better approximation. This, in turn, enables a smaller value of k, which significantly decreases the size of the coefficient table.

For the second-order method, $\text{atan2}(y,x)$ is approximated as:

$$\begin{aligned} T_2(x_r,y_r) &\approx P_{x_h y_h}(\delta x,\delta y) \\ &\approx a\cdot\delta x + b\cdot\delta y + c + d\cdot(\delta x)^2 + e\cdot(\delta y)^2 + f\cdot\delta x\cdot\delta y \end{aligned}$$

The coefficients a to f are again obtained by using the Taylor series for $\text{atan2}(y_r,x_r)$, around a point (x_0,y_0):

$$\begin{aligned} T_2(x_r,y_r) &= f(x_0,y_0) \\ &+ \tfrac{\partial}{\partial x}f(x_0,y_0)\cdot(x_r-x_0) + \tfrac{\partial}{\partial y}f(x_0,y_0)\cdot(y_r-y_0) \\ &+ \tfrac{1}{2}\tfrac{\partial^2}{\partial x^2}f(x_0,y_0)\cdot(x_r-x_0)^2 \\ &+ \tfrac{1}{2}\tfrac{\partial^2}{\partial y^2}f(x_0,y_0)\cdot(y_r-y_0)^2 \\ &+ \tfrac{\partial^2}{\partial x\partial y}f(x_0,y_0)\cdot(x_r-x_0)\cdot(y_r-y_0) \end{aligned}$$

By replacing f as $\text{atan2}(y_r,x_r)$, and performing the same variable change $x_r = x_h + \delta x$ and $y_r =$

$y_h + \delta y$, results in:

$$\begin{aligned} T_2(x_r, y_r) &= \text{atan2}(y_h, x_h) + \frac{-y_h}{x_h^2+y_h^2}\delta x + \frac{x_h}{x_h^2+y_h^2}\delta y \\ &+ \frac{x_h y_h}{(x_h^2+y_h^2)^2}\delta x^2 + \frac{-x_h y_h}{(x_h^2+y_h^2)^2}\delta y^2 + \frac{y_h^2-x_h^2}{(x_h^2+y_h^2)^2}\delta x^2 \delta y^2 \end{aligned}$$

Error analysis

The discussions concerning the error analysis are limited to the approximation of $\text{atan2}(x_r, y_r)$ by the second-order Taylor polynomial. For the first-order method the analysis is actually simpler, and the same approach can be taken. Illustrations for the sources of errors on both datapaths are provided in Figures 10.13 and 10.14, respectively. The goal is again to obtain a faithful result, meaning $\varepsilon_{total} < 2^{-w}$, with ε_{total} denoting the total error. There are three main sources of error:

$$\varepsilon_{total} = \varepsilon_{method} + \varepsilon_{final\_round} + \varepsilon_{round}$$

The magnitude of ε_{method}, the method error, is determined by the parameter k. The Taylor series approximation contains an infinite number of terms:

$$\begin{aligned} T_2(x, y) &= c_0 + c_{x\_1}x + c_{y\_1}y \\ &+ c_{x\_2}x^2 + c_{x\_1y\_1}xy + c_{y\_2}y^2 \\ &+ c_{x\_3}x^3 + c_{x\_2y\_1}x^2y + c_{x\_1y\_2}xy^2 + c_{y\_3}y^3 + \ldots \end{aligned}$$

Limiting the approximation to the second order terms results in an error which is of the same order of magnitude as the terms that are omitted:

$$\varepsilon_{method} = \sum_{i=3}^{\infty}(c_{x\_i}x^i + c_{y\_i}y^i + \sum_{k,l,\ k+l=i} c_{x\_ky\_l}x^k y^l)$$

where the c_{x_i}'s and c_{y_i} are obtained by expanding the Taylor series. ε_{method} can therefore be bounded by:

$$\begin{aligned} \varepsilon_{method\_Max} &= c_{x\_max}\sum_{i=3}^{\infty}(2^{-k})^i + c_{y\_max}\sum_{i=3}^{\infty}(2^{-k})^i \\ &+ c_{xy\_max}\sum_{i=3}^{\infty}(2^{-k})^i \\ &< (c_{x\_max} + c_{y\_max} + c_{xy\_max})\frac{2^{-3k}}{1-2^{-k}} \end{aligned} \tag{10.7}$$

c_{x_max}, c_{y_max} and c_{xy_max} being the maximum values of c_x, c_y, and c_{xy}. Equation 10.7 can be used to determine the value of k, which must satisfy $k > \lceil \frac{w}{3} \rceil$. But this is a pessimistic bound, and it can be improved using numerical methods: set the value of k, the fill the table, then compute ε_{method} in each point of the domain using the actual coefficient values. If the error is above the error budget, then k

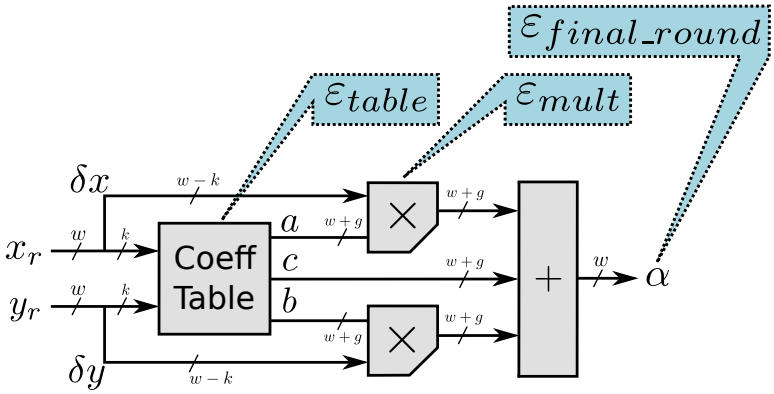


Figure 10.13: Errors on the first order bivariate polynomial approximation implementation

is increased. The obtained values for the coefficients can be used to provide a tight bound on ε_{method}.

The error due to the final rounding $\varepsilon_{final\_round}$ can be bounded by 2^{-w}, when the final result is rounded to the output format.

Each term of $T_2(x_r, y_r)$, except for c, adds an error due to the truncation/rounding of the multiplications/squares, and an error due to the tabulation of the real coefficients. The expressions for the error terms are given after the variable change, which is errorless. For $a \cdot \delta x$, the error is:

$$\begin{aligned}
\varepsilon_{a\cdot\delta x} &= \widetilde{\widetilde{a}\cdot\delta x} - a\cdot\delta x \\
&= (\delta x\cdot\widetilde{a} + \varepsilon_{mult_{a\delta x}}) - a\cdot\delta x \\
&= (\delta x(a+\varepsilon_{a_table}) + \varepsilon_{mult_{a\cdot\delta x}}) - a\cdot\delta x \\
&= \varepsilon_{mult_{a\cdot\delta x}} + \delta x\cdot\varepsilon_{a_table}
\end{aligned}$$

The same holds for $b \cdot \delta y$ term. The error due to c in the final sum comes from storing the coefficient in the table: $\varepsilon_c = \varepsilon_{c\_table}$. For the second order terms, the error can be found in a similar manner:

$$\begin{aligned}
\varepsilon_{d\cdot\delta x^2} &= \widetilde{\widetilde{d}\cdot\delta x^2} - d\cdot\delta x^2 \\
&= (\widetilde{\delta x^2}\cdot\widetilde{d} + \varepsilon_{mult_{d\cdot\delta x^2}}) - d\cdot\delta x^2 \\
&= ((\delta x^2 + \varepsilon_{sqr_{\delta x^2}})(d+\varepsilon_{d_table}) + \varepsilon_{mult_{d\cdot\delta x^2}}) - d\cdot\delta x^2 \\
&= \delta x^2\cdot\varepsilon_{d_table} + d\cdot\varepsilon_{sqr_{\delta x^2}} + \varepsilon_{sqr_{\delta x^2}}\cdot\varepsilon_{mult_{d\cdot\delta x^2}}
\end{aligned}$$

The same holds for $e \cdot \delta y^2$. Lastly, for $f \cdot \delta x \cdot \delta y$ the error can be expressed as:

$$\begin{aligned}
\varepsilon_{f\cdot\delta x\cdot\delta y} &= \widetilde{f\cdot\delta x\cdot\delta y} - f\cdot\delta x\cdot\delta y \\
&= (\widetilde{\delta x\cdot\delta y}\cdot\widetilde{f} + \varepsilon_{mult_{f\cdot\delta x\cdot\delta y}}) - f\cdot\delta x\cdot\delta y \\
&= ((\delta x\cdot\delta y + \varepsilon_{mult_{\delta x\cdot\delta y}})(f+\varepsilon_{f_table}) + \varepsilon_{mult_{f\cdot\delta x\cdot\delta y}}) - f\cdot\delta x\cdot\delta y \\
&= \delta x\cdot\delta y\varepsilon_{f_table} + f\cdot\varepsilon_{mult_{\delta x\cdot\delta y}} + \varepsilon_{mult_{\delta x\cdot\delta y}}\cdot\varepsilon_{mult_{f\cdot\delta x\cdot\delta y}}
\end{aligned}$$

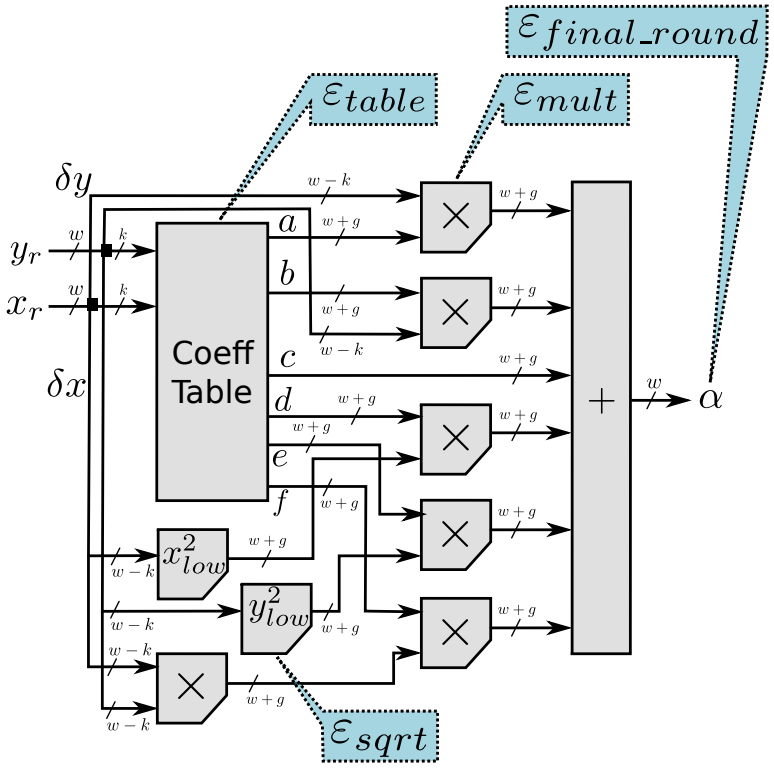


Figure 10.14: Errors on the first order bivariate polynomial approximation implementation

This gives the expression for ε_{round}:

$$\varepsilon_{round} = \varepsilon_{a\delta x} + \varepsilon_{b\delta y} + \varepsilon_c + \varepsilon_{d\delta x^2} + \varepsilon_{e\delta y^2} + \varepsilon_{f\delta x\delta y}$$

Assuming that the errors due to tabulation are all equal to ε_{table} and that the errors due truncation/rounding of multiplications and squares are ε_{mult} and ε_{sqr}, respectively, ε_{round} becomes:

$$\begin{aligned} \varepsilon_{round} &= \varepsilon_{mult}(2 + f + 2\varepsilon_{sqr} + \varepsilon_{mult}) \\ &+ \varepsilon_{table}(1 + \delta x + \delta y + \delta x^2 + \delta y^2 + \delta x \cdot \delta y) \\ &+ \varepsilon_{sqr}(d + e) \end{aligned} \tag{10.8}$$

Datapath dimensioning

As with the previous methods, the error analysis ensures that the final result is faithful with respect the mathematical result. It also ensures an architecture that computes with the minimal amount of resources.

Therefore, the required number of additional guard bits g (as illustrated in Figure 10.14) due to

$\varepsilon_{final\_round}$ and ε_{round} needs to be determined in order to establish the data format of the computations. The final rounding has an error that can be bounded as $\varepsilon_{final\_round} < 2^{-w}$, if a truncation is used.

From Equation 10.8, ε_{round} can be controlled by the precision of the multiplications and squares. It is also influenced by precision of the stored coefficients, which depends on the choice for the k parameter. This is due to the dependence of ε_{round} to δx and δy, which satisfy $\delta x < 2^{-k}$ and $\delta y < 2^{-k}$.

A choice on the value of g, is necessarily a compromise that tries to balance (by taking into consideration the size of the corresponding hardware) the size of the multiplications and that of the coefficient tables.

Results and Discussion

The synthesis results presented in the following are obtained using the Xilinx ISE 14.7 suite, and target a Virtex 6 (6vhx380tff1923) device.

Logic-only Synthesis

Tables 10.1 and 10.2 show results for logic-only implementations of the methods, where the multipliers and the tables are synthesized purely in logic. In this case, faithful truncated multipliers can be used so as to save resources and latency. The method denoted *degree 0* uses plain tabulation, *degree 1* uses a bipartite approximation [39], and *degree 2 and above* use a Horner scheme inspired by [35]. For the small precisions required in the architecture of Figure 10.12, the squaring operators also also use bipartite approximation. This allows for savings in terms of resources, as compared to a dedicated squarer.

The objective of this data is to provide a fair comparison between the methods, in absolute terms. A first observation is that CORDIC behaves extremely well on modern FPGAs. The multiplier- and table-based methods never perform better in area, and only rarely beat its latency. This is especially disappointing as the architecture of Figure 10.12 exhibits a lot of parallelism. The comparison between CORDIC and the RecipMultAtan method of Section 10.5 is consistent with the findings of [51].

It is also interesting to analyze this data in conjunction with the findings of Chapter 9, which studied CORDIC implementations for the sine and cosine functions. Curiously, it lead to the opposite conclusion: for sine/cosine, it was shown that CORDIC had longer latency than multiplier-based methods even when the multipliers were implemented in logic.

The bivariate approximation methods can be compared to a bivariate interpolation technique used in [131]. The technique used there resorts to two sequential interpolation processes (on x, then

Bitwidth	Method	LUT	Latency (ns)
CORDIC			
8		173	9.3
12		435	14.6
16		734	19.7
24		1504	31.0
32		2606	43.1
Taylor degree 1 / Plane			
8		207	12.64
12		1258	14.74
16		37744	20.20
Taylor degree 2			
8		356	13.72
12		469	14.75
16		1509	17.90
RecipMultAtan			
8	degree 0	175	11.8
12	degree 0	683	16.2
12	degree 1	443	19.0
16	degree 1	1049	19.1
24	degree 2	2583	35.2
32	degree 2	6190	40.7
32	degree 3	5423	50.8

Table 10.1: Logic-only (combinatorial) synthesis results

on y). The results reported in [131] (extrapolated from a different FPGA family, Altera Stratix II), tend to show that this bivariate interpolation technique has longer latency and consumes even more resources, mainly due to the high memory requirements.

The complexity assumption made in the introduction can be re-evaluated in the light of the current findings. In principle, classical (i.e. non-redundant) CORDIC is quadratic both in area and delay. The CORDIC algorithm computing to precision w consists of about w iterations, each consisting of three add/sub operations of about w bits, hence the quadratic area. Besides, there is a carry propagation in each iteration, whose least significant bit input depends on the most significant bit of the previous iteration: there is a critical path of size w^2.

And Table 10.1 confirms this quadratic area. However, the constant is very small. Specifically, it can be observed that the area of CORDIC is roughly $3w^2$. This means that each CORDIC iteration is indeed implemented in 3 LUT per bit. In other words, a multiplexed addition and subtraction is

Precision		LZC	Shift	Recip	Mult	Atan
12 bits	area	12	2x36	149	159	220
	delay	1.63	1.72	2.23	4.22	2.54
16 bits	area	16	2x47		294	
	delay	2.5	1.76		5.9	

Table 10.2: Breakdown of area and latency for RecipMultAtan method

mapped to a row of LUTs, and therefore consumes no more than a simple addition.

On the other hand, the latency of CORDIC does not seem quadratic: it seems linear in w. This is explained by the fact that the carry propagation delay is about 30 times faster than the standard routing used between two iterations. It justifies *a posteriori* the choice of ignoring redundant versions of CORDIC.

A second observation is that the first degree bivariate approximation is never interesting. Its size explodes too fast, and this even impacts the latency.

It should also be remarked that the scaling-based argument reduction is small and fast, as illustrated by Table 10.2. In [51], the shifts are implemented as one-hot encoding then multiplication in a DSP block. This is probably overkill.

Pipelining, DSP- and table-based results

Table 10.3 shows some synthesis results for pipelined operators at various frequencies, exploiting the embedded multipliers and memories, when possible. The multiplier-based methods manage to improve logic count, but bring no clear advantage in terms of latency nor frequency compared to a pipelined unrolled CORDIC. However, this is not something which is intrinsic to the algorithms, but more probably reflects weaknesses of FloPoCo in managing the pipelining of such entities, at the time of performing the experiences.

Still, the excellent match between CORDIC and FPGA logic can again be observed: as the third line of Table 10.3 shows, it is possible to pack two 16-bit iterations in one pipeline stage operating at nearly 400MHz.

The most novel method, based on piecewise bivariate polynomials, still does not manage to reduce the latency as much as expected. There are many improvements to bring to this method, the first being to reduce k using degree-2 approximation techniques that are more accurate than Taylor. Unfortunately, for functions of two or more inputs, there does not seem to exist an equivalent to the Remez or minimax polynomial approximation algorithms. The problem is that the alternation property on which Remez is based [92] is difficult to transpose in two dimensions.

Yet another advantage of the CORDIC method, which should be noted, is that it may also com-

Method	LUT + Reg.	BRAM + DSP	Speed cycles@freq.
CORDIC	816 + 44		2@191
	799 + 202		5@274
	796 + 336		8@389
RecipMultAtan 1	320 + 51	2+1	2@112
	315 + 68	2+1	3@199
RecipMultAtan 2	425 + 199	0+5	10@130
	432 +250	0+5	14@253
Taylor 2	331 + 53	4+6	1@135
	327 + 103	4+6	3@144
	329 + 140	4+6	5@220

Table 10.3: Results for pipelined 16-bit architectures

pute the module $\sqrt{x^2+y^2}$ along with the angle [92]. This costs only one additional constant multiplication by $1/K$.

Among other possibilities to explore around atan2, [55] decomposes the computation into two successive rotations, a coarse one and a finer one. Both first approximate $z = \frac{y}{x}$ and then compute $\arctan(z)$. This is a middle ground between CORDIC and multiplicative methods.

A table- and multiplier- based combined range reduction and scaling method could enable the bivariate techniques to scale better, but again at the expense of latency.

11

Reciprocal, Square Root and Reciprocal Square Root

The work presented in this chapter was done in collaboration with Bogdan Pașca, and was supported by the Altera European Technology Centre (now part of Intel Programmable Solutions Group (PSG), since the acquisition of Altera in 2016).

This chapter analyzes, as well, some of the available techniques for computing single-input functions. These functions are the reciprocal (also commonly referred to as the multiplicative inverse, denoted in the following as *recip.*), the square root (*sqrt*) and the reciprocal square root (*rsqrt*). The three functions are studied as they share many characteristics in common, and similar techniques can be applied for their computation. These functions are critical in many domains, ranging from 3D rendering and computer vision [108], to convolutional networks [43], MIMO communication systems [27][66], scientific computing [101], bioinformatics [113] and many others.

Many applications provide ad-hoc implementations for these functions, which are rarely optimal in terms of efficiency and rarely focus on topics such as the accuracy of the computations. Therefore, this chapter discusses the creation of generators for the reciprocal/square root/reciprocal square root functions, targeting fixed-point implementations, in the context of modern FPGA devices. These feature large numbers of embedded multiplier blocks and memory blocks, alongside the look-up tables.

Contrary to most of the work presented in this thesis, the arithmetic core generator is implemented in the DSP Builder Advanced (DSPBA) description language, presented in [28]. They are also part of the library of operators proposed by the DSPBA, as a command-line tool inside the

Quartus II design suite proposed by Altera.

This chapter sets out to create a set of flexible and versatile generators for the three functions, targeting a wide range of modern Intel devices. All the functions are implemented using fixed-point notations, targeting user-defined custom formats. The target frequency is also user-specified, with automatic pipelining made possible by the tool, using the techniques of [28]. Where possible, the generator should also pick between the best possible method, according to a trade-off between device resources that is left up to the user.

Numerous techniques for implementing the three fixed-point functions are available in the literature. To start with, there are the direct tabulation and bi- and multipartite methods, which combine tabulation with some basic arithmetic operations, covered in Section 11.4. Other classical approaches are based on Taylor series approximation, minimax polynomial approximations, or piece-wise polynomial approximations. These techniques, or some of their derivations are discussed throughout Sections 11.5 and 11.6. Iterative methods are also considered, in Section 11.6, starting with quadratic convergence for the classic Newton-Raphson method, and exploring higher-order methods such as the Halley or Householder methods, the latter being novel in this context.

An interesting remark is that different methods are efficient for different contexts, such as the varying sizes of the embedded multiplier blocks, or of the embedded memories. The most efficient methods provide the best trade-off between look-up tables and these resources. The generators described in this chapter seek to automate this choice.

A second remark worth making is that a large portion of the literature deals with the floating-point implementation of the three functions. There are many similarities between the two instances, and in many cases the bulk of the problem are the computations involving the mantissa, which is a fixed-point number between $[1, 2)$, where the functions in question have a nice behavior. Therefore, this chapter also explores architectures that avoid the typical normalize/compute/reconstruct flow, assessing the possible gains or the compromises to be made.

A Common Context

The purpose being to provide a fair and relevant comparison between the methods considered in this chapter, they are tested on devices with a common set of features. The target devices are Intel Cyclone V FPGAs, although the methods can easily be tuned for other families with ease. The look-up tables can be configured as two independent 4-input LUTs, or one 6-input LUT, or other combinations that share common inputs, and are grouped in Adaptive Logic Modules (ALMs). The resource usage for Intel devices are measured as number of required ALMs.

The selected target devices also have multiplier blocks available, that can be used in various modes: three 9×9 bit independent multiplications, two 18×19 bit multiplications, one 27×27 bit mul-

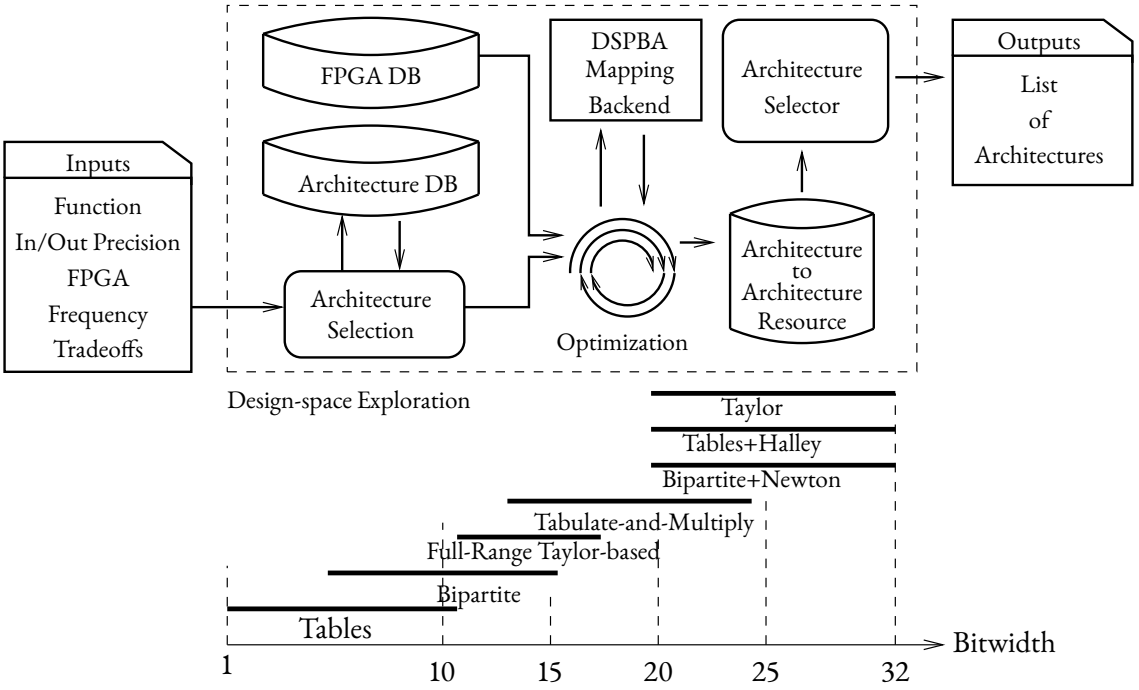


Figure 11.1: A high-level view of the generator and of the various methods and their bitwidth ranges.

tiplication and a few other modes. The embedded memory blocks are of a capacity of 10Kb each (and are therefore referred to as M10K), that can be used in various modes, some of the relevant ones being $2^{11} \times 5$ bits, $2^{10} \times 10$ bits, $2^9 \times 20$ bits or $2^8 \times 40$ bits.

Differentiating criteria must be chosen in order to classify the methods. While more criteria results in a more complicated selection process, it also has the advantage of providing the user with the most appropriate method for the given context. In the following, three main aspects are used for classification of the methods. They are by no means exclusive, but bring out some of the challenges with which the user is faced, as well as the constant quest for a better use of *all* the available resource on the target device.

Bitwidth: choosing the implementation method which is the most relevant for a certain bitwidth can help achieve this goal, as shown in Figure 11.1.

Input range: choosing an implementation while benefiting from extra knowledge about the input range allows for more specialized solutions. Trading in some of the generality of the method can have its benefits.

Approach: methods based on tabulation, polynomial approximation and iterative methods are some of the most popular choices for implementing the three functions. Hybrids between these classes are also possible. Some are more flexible than the others, and trade-offs are most likely to be made when choosing between the different classes.

Figure 11.1 gives an overview of the typical flow within the generator. The inputs are the target function (recip., sqrt or rsqrt), the fixed-point input format and possible constraints on the output. In terms of the performance specification, the inputs contain the target FPGA device and the target frequency. Where applicable, the user can also input a trade-off between the resources available on the target device. The output is the RTL specification of the implementation.

A Classical Range Reduction

For methods designed for floating-point to be relevant in the context of this chapter, the input to the fixed-point computational kernel has to be bounded to the interval $[1,2)$. In order to apply such an algorithm to a fixed-point input, a few steps are typically followed. First, the input is normalized by using a pre-scaling (gets the argument to $[1,2)$). A computation on the reduced argument, which is now in a reduced interval, is then performed. The last step is the reconstruction, in order to retrieve the original function (a post-scaling of sorts, whose complexity is dependent on the function).

In the case of the recip., sqrt and rsqrt functions the classical range reduction is based on the substitution

$$x = 2^e \cdot x'$$

where x is the input, e is the amount by which x needs to be shifted for normalization and x' is the normalized input. With $x' \in [1,2)$, algorithms designed for floating-point can be applied. The reconstruction phase for the three functions can be performed according to Equation 11.1:

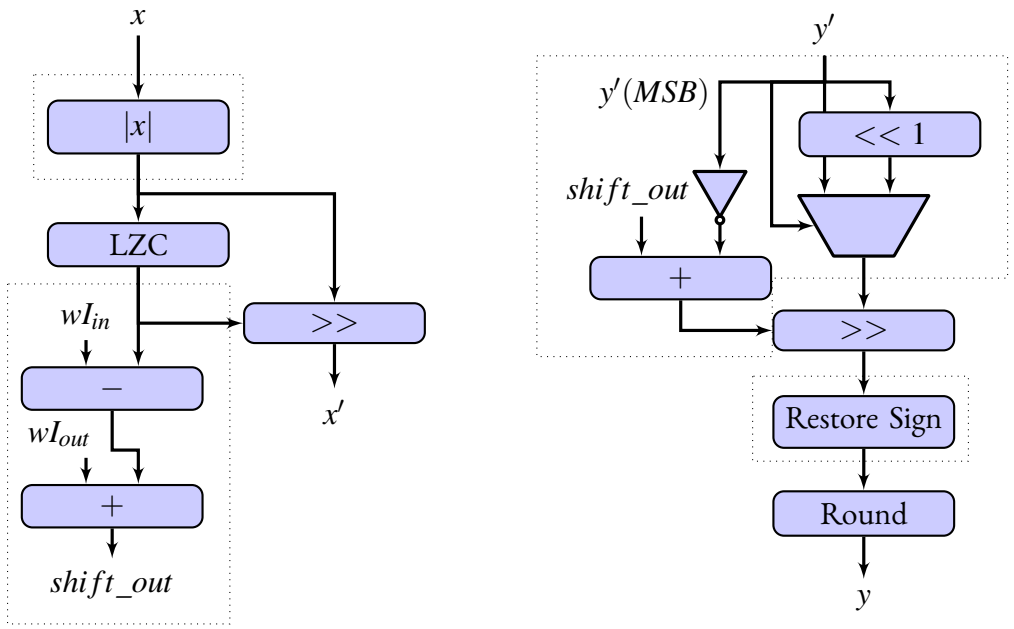
$$y_{recip} = 2^{-e} \cdot \frac{1}{x'} \tag{11.1a}$$

$$y_{sqrt} = \begin{cases} 2^{\frac{e}{2}} \cdot \sqrt{x'} & \text{if } e \text{ is even} \\ 2^{\frac{e-1}{2}} \cdot \sqrt{2} \cdot \sqrt{x'} & \text{if } e \text{ is odd} \end{cases} \tag{11.1b}$$

$$y_{rsqrt} = \begin{cases} 2^{-\frac{e}{2}} \cdot \frac{1}{\sqrt{x'}} & \text{if } e \text{ is even} \\ 2^{-\frac{e-1}{2}} \cdot \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{x'}} & \text{if } e \text{ is odd} \end{cases} \tag{11.1c}$$

In order to compute the functions on the reduced argument x', a choice of methods can be used.

Figure 11.2 presents the operations required for the pre- and post-processing phases. The computation of the absolute value of x in Figure 11.2a is only needed for the recip. function. The circuitry in the bottom left dotted rectangle is required for the post-processing phase, computing the amount by which to shift the result of the computation kernel. The post-processing phase is illustrated in Figure 11.2b, where the dotted polygon at the top of the figure contains the logic required for properly aligning y' and adjusting the shift amount. Restoring the sign is only required for the recip. function,



(a) Pre-scaling

(b) Post-scaling

Figure 11.2: Scaling range reduction

and can be integrated into the rounding operation, if needed.

Background

As modern FPGA devices are mostly made-up of a large number of storage elements (mainly look-up tables, but also embedded memories), *tabulation-based methods* are a popular solution. Direct tabulation is straight-forward and simple to apply for bitwidths up to the maximum addressing size of the storage elements, which is 10-12 bits. The correctly rounded results can be stored directly in the tables, having been previously computed using multiple precision software. Another positive aspect of this approach is that underflow and overflow logic can be integrated at no extra cost. One main drawback of tabulation is the size of the table increases exponentially with the size of the input, once the 10-12 bit range is surpassed.

Several methods try to overcome this limitation of tabulation-based methods, with as little extra resources as possible. Among them are the ATA method [135], the iATA method [78], the bipartite method [31], the SBTM method [109], STAM method [115] and the generalized multipartite methods [91][39][54]. The extra logic is usually composed of additions and bitwise operations. More details on this category of approaches are provided in Section 11.4.

Relying solely on tabulation-based methods eventually becomes impractical, as the input preci-

sion increases. *Polynomial approximation-based methods* prove to scale better. The polynomial itself can be obtained in several ways, amongst which variations on the *Taylor series* or *generic polynomial approximation* (minimax polynomials, for example). For smaller bitwidths, some relevant methods using the former approach include [119][59]. These are treated in more detail in Section 11.5. For larger bitwidths, a relevant example method is the one presented in [41]. This and other methods targeting larger bitwidths are treated in Section 11.6.

As an alternative to polynomial approximation, *iterative methods* are an option to consider for targeting large precisions. The *Newton-Raphson method* is probably the most popular in this class, a zero finding method that doubles the precision of the initial approximation at each iteration. Obtaining larger gains per iteration is possible using higher order methods. *Halley's method* is the next in the series, after Newton's method, which has a two-fold increase in precision per iteration. More generally, *n*-fold methods exist, known as the *Householder methods* [19]. Iterative methods are discussed in Section 11.6.

Figure 11.1 on page 169 also presents a rough estimation on which method to apply, according to the bitwidth.

Tabulation and the Multipartite Methods

Multipartite Methods

The multipartite methods are multiplierless techniques for the evaluation of functions. The basic principle is to create and approximation of a function f using a set of affine segments. Each of the segments represents the approximation of f on a corresponding segment of the input domain. The original *bipartite method* is based on a degree 1 polynomial using the first two terms of the Taylor series. evaluated at a given point A this results in:

$$f(x) \approxeq f(A) + f'(A) \cdot (x - A) \tag{11.2}$$

In order to simplify the addressing of the tables, the input is split into a power of 2 number of segments/subintervals. The point A is selected as an easy to compute point inside each of the intervals. The simplest is to choose A as the top k bits of the input x. However, a better balanced approximation can be obtained by choosing A as the middle of each of the subintervals.

In order to avoid the need for a multiplication in the second term in Equation 11.2, a coarser approximation is used for the derivative. The remaining bits of x are denoted as $B = x - A$. Instead of storing the slope for each subinterval, one slope $f'(A) \approxeq f'(C)$ is shared for a set group of subintervals. C is chosen as some of the most significant bits of A. Therefore, the second term can be computed without requiring multiplications, by tabulation using B and C. An illustration of the method

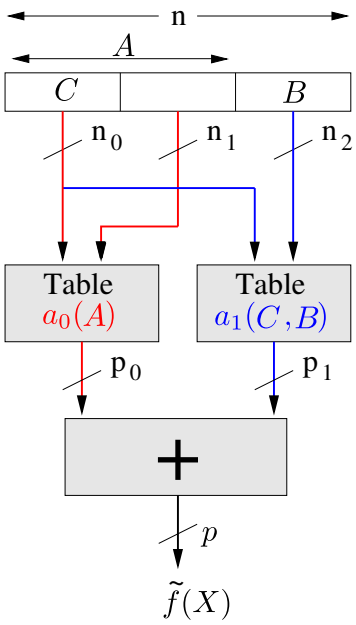


Figure 11.3: The bipartite method

is presented in Figure 11.3.

The size of the table storing the second term in Equation 11.2 can be cut in half by taking advantage of symmetry. The method can be further improved, as described in [115] and [39], where better polynomial approximations are achieved. The multipartite method is a generalization of the bipartite method [39]. The idea is to overcome the exponential growth of the two tables by splitting the input into several parts, not just the three as in the bipartite method.

A Method for Evaluation on the Full Range Format

The multipartite methods have convergence domains that are close to the $[1,2)$ range, for the three functions considered in Chapter 11. In general, the quality of the approximation provided by the multipartite methods decreases as the derivative increases. Applying these methods outside this range requires the classical range reduction and restoration phases of Section 11.2. However, this comes at a cost, as leading zero/one counters and barrel shifters are required for the normalization, and barrel shifters and constant multipliers (for sqrt and rsqrt) are required for the reconstruction phase.

Therefore, it is interesting to consider alternative methods that do not require that the input be bounded to the $[1,2)$, and as such do not require range-reduction and reconstruction stages. This section proposes such a bipartite-inspired method and analyzes the compromises that it imposes, as well as the potential gains it brings on a range of input fixed-point formats. The method has two stages: a first refinement stage, which improves the accuracy of the slope for input ranges that are

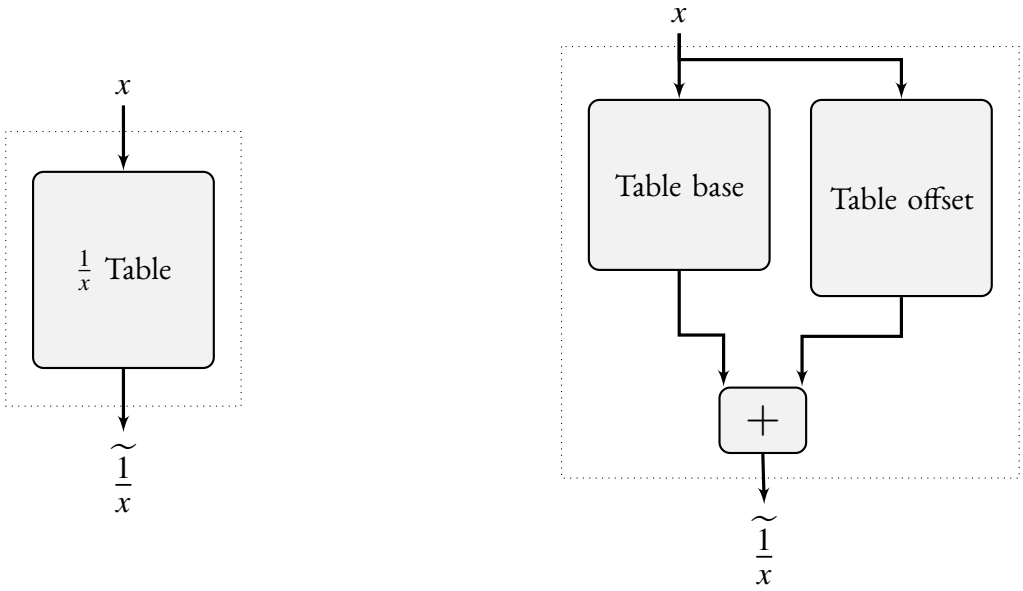


Figure 11.4: The two architectures for the tabulation of outputs (for cases when the method error exceeds the admitted value)

close to the divergence domain (for example, for the recip. function, the subintervals close to zero). In the second stage, for the parts of the domain for which the method cannot satisfy the accuracy constraint, dedicated circuitry is built to handle these cases separately. The architecture parameters are dependent on a variety of factors, therefore these decisions are automated inside the core generator.

The proposed architecture is also based around a degree 1 polynomial, for most of the input range. The coefficient are obtained using techniques and optimizations similar to those described in [39]. For input range where the method error exceeds the maximum allowable one, a tabulation technique is used. The accuracy of the used polynomials can be improved by performing a two dimensional space exploration, looking for the segment that minimizes the overall error. This could be visualized as small adjustments that are made to $f(A)$ and $f'(A)$.

A better approximation results in a smaller error and therefore less points for which the function needs to be tabulated. The embedded memory blocks on FPGAs are monolithic. Thus, lowering the number of points for which tabulation is needed can reduce the total memory blocks used by half.

For the recip. and rsqrt functions, for many of the input values the function output is larger than the maximum representable value on the output format. In these cases, the output is saturated. There are also cases where the value computed by the architecture produces an error larger than the maximum allowed error. The chosen approach for these situations is to use tabulation. An illustration of the overflow architecture is presented in Figure 11.4. It can be composed by a single table, or

comprised out of a base and an offset, in a similar way to the solution proposed in [54], which allows for the compression of the table.

A small optimization can be performed in cases where the total number of points in the saturation and the insufficiently precise ranges is lower than the number of cells in the memory block used for implementation. In such situations, tabulation of the rest of the inputs, inside the same memory block, comes for free.

As size of the input to the table increases, the number of memory blocks required for implementing the table also increases, and can increase considerably. An optimization can be made based on a simple observation, as suggested in [54]. As the table stores function values for consecutive input values, the differences between the elements stored in the table can be small. Therefore, a more efficient way of storing it is to store sampled values in a base table, and store the corresponding increments/decrements in an offset table. The original value can be retrieved without loss through a simple addition between the outputs of the two tables. The gain consists in having a lower combined bitwidth at the output of the two tables, compared to that of a unique table. Determining if the base and offset architecture is worth it can be done automatically, through a design space exploration done by the generator. This depends on the fixed-point format and the size of the embedded memory blocks on the target device.

The proposed architecture also handles underflow situations. For example, for the recip. function, depending on the fixed-point format used, many of the inputs result in underflow at the output. Consider the format with 16 bits in total, out of which 4 are for the fractional part: this is a case where a dedicated underflow architecture is more efficient.

The architecture is again based on tabulation. There are two main cases: for the first, tabulation is used for all values for which underflow does not occur (the output is different from zero on the given fixed-point format). In the second case, tabulation is used for all values for which the output is different from zero, or from one unit in the last place (the smallest number representable on the format, in absolute value). The indices from which the function should return one unit in the last place or zero can be determined using numerical methods. Based on these determined values, the generator can analyze the two possible architectures and the cost they incur. The best option can be selected automatically. When the same number of memory blocks are required for the two architectures, the method chosen is the first one, as it requires less logic elements for implementation. The underflow architectures are depicted in Figure 11.5.

Some Results for the Lower Precision Methods

Some results for the first-order polynomial approximation-based method of Section 11.4.2 are presented in Table 11.1, for varying fixed-point input formats on 16 bits. The left part of the table shows results using the coefficients obtained from the Taylor series expansion, while the right part corre-

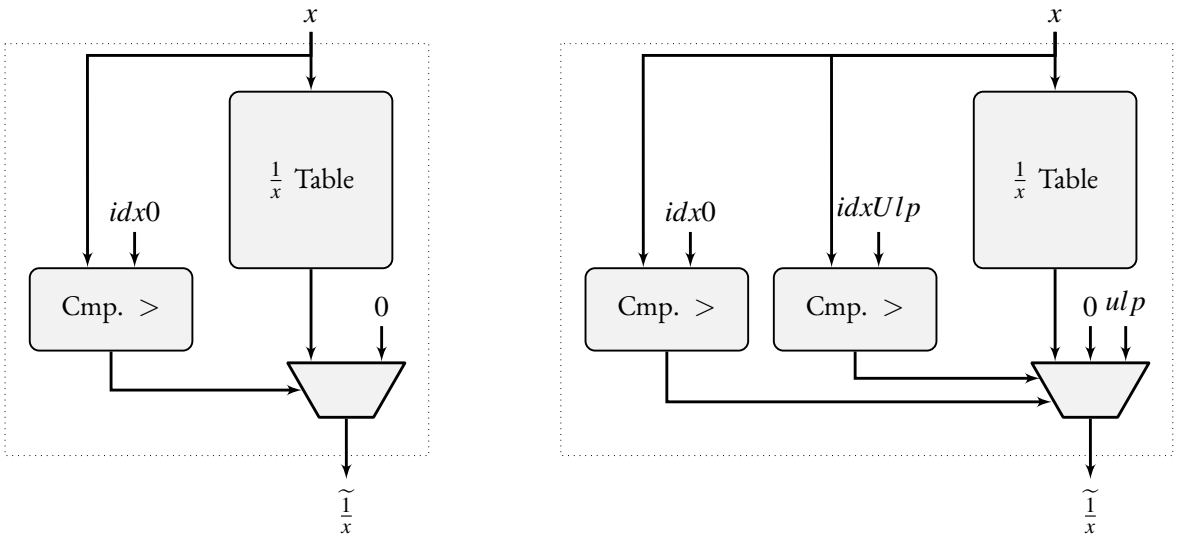


Figure 11.5: The two cases of the underflow architecture

sponds to results obtained by using the refined coefficients. The presented data correspond to the best architecture (the one which approaches the function closest), obtained through a design space exploration, done automatically by the generator in a matter of seconds. There are several possible architectures:

- dedicated underflow architecture: this architecture is used when a large proportion of the input range results in an underflow at the output. There are two versions, as shown in Figure 11.5. For example, the one on the left is triggered by the (11, -4) input format, while the one on the right is triggered by the (10, -5) format.
- an architecture that is sufficiently accurate in the input range (except for overflow). For example, this is triggered by the (0, -15) format.
- an architecture that uses extra table for compensating where the accuracy of the approximation is insufficient. The extra table is implemented so as to minimize the architecture. If table does not fill the size of a memory block, inputs that cause overflow can also be stored in the same memory element. For example, for the (3, -12) format, one memory block can be saved using the base+offset architecture. For the (1, -14) format one memory block can be saved as well.

A look over both sides of Table 11.1 shows that for some formats the achievable savings by using the optimizations are better than for the others. This is mainly due to the threshold effect that the hard memory blocks have. In those cases, it was not possible to reduce sufficiently enough the number of stored elements (usually below a power of 2), so as to trigger a smaller block memory utilization.

I/O Format	Resource utilization Default				Resource utilization Optimized			
(m, l)	ALM	DSP	M10K	Lat. @Freq.	ALM	DSP	M10K	Lat. @ Freq.
0,-15	61	1	3	7 @ 310MHz	54	1	3	7 @ 310MHz
1,-14	101	1	12	9 @ 242MHz	96	1	8	9 @ 235MHz
2,-13	107	1	15	9 @ 236MHz	101	1	10	9 @ 240MHz
3,-12	103	1	11	9 @ 233MHz	105	1	11	9 @ 232MHz
4,-11	88	1	12	7 @ 240MHz	86	1	8	6 @ 235MHz
5,-10	95	1	8	7 @ 241MHz	95	1	8	7 @ 236MHz
6,-9	85	1	6	7 @ 228MHz	86	1	6	7 @ 235MHz
7,-8	61	1	5	6 @ 241MHz	61	1	5	6 @ 241MHz
8,-7	59	1	5	6 @ 237MHz	61	1	5	6 @ 231MHz
9,-6	52	1	4	6 @ 303MHz	52	1	4	7 @ 303MHz
10,-5	58	1	2	6 @ 308MHz	58	1	2	7 @ 308MHz
11,-4	28	0	1	2 @ 315MHz	28	0	1	2 @ 315MHz
7,-8	126	0	5	12 @ 240MHz	← Generic Range Reduction			

Table 11.1: Resource utilization and estimated performance of the first degree Taylor based method working on the full input range , for 16- bit precision inputs , varying input / output formats from 15 fractional bits down to 4 fractional bits. The target function is the reciprocal. Results are given for a Cyclone V C6 device. The final row shows the utilization and performance of bipartite operator using the range-reduction to the interval $[1, 2)$. This implementation is agnostic to the input format.

The last line of Table 11.1 presents results for the bipartite method. It also contains the normalization and denormalization stages (common for fixed-point methods applied to a generic range). The actual format is not important (due to the need for the pre- and post-processing), and therefore only one line is shown. It can be observed that is has a longer latency as compared to the optimized implementations on the right of the table. Also, the method requires only memory block, and no multiplier blocks, but also consumes more logic resources than all the other presented methods. In terms of memory blocks, the method on the last line performs better than the proposed method for the (1, -14) to (6, -9) range of formats, but is on par or worse for the rest of the formats.

Tabulate and Multiply Methods

First Order Method

While the multipartite methods have the advantage of relying solely on tables, this turns into a shortcoming with the increase of the input bitwidth, when the table sizes grow considerably. The method of [119], also based on the Taylor series, tries to eliminate this problem, trading-off storage requirements for just one multiplication. The method is designed to compute specific power functions x^P,

where P is of the form $P = \pm 2^k$, with k integer. Taking $P = -1$ (with $k = 0$), $P = \frac{1}{2}$ or $P = -\frac{1}{2}$ (with $k = -1$) the recip., sqrt and rsqrt functions can be obtained.

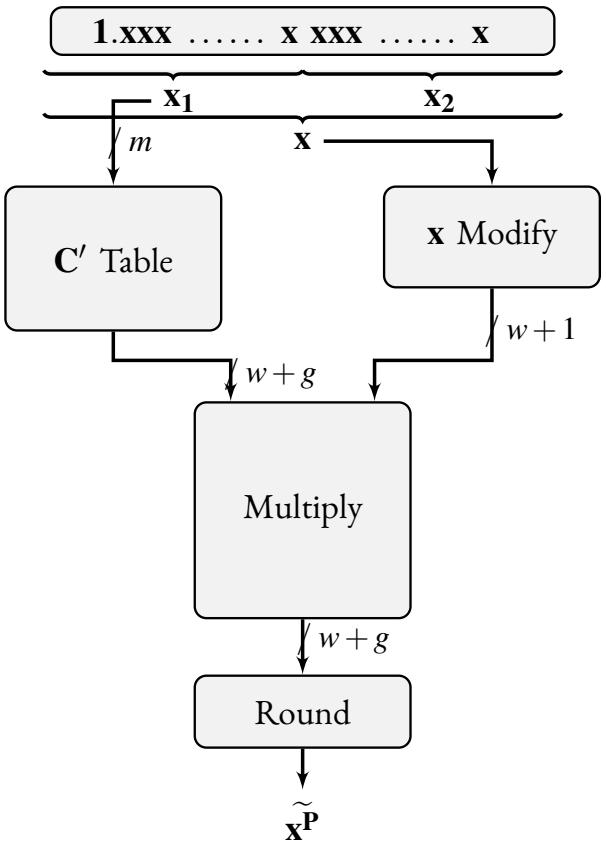


Figure 11.6: Tabulate-and-multiply method architecture

The first two terms of the Taylor series expansion are used. The method starts by splitting the input x into two parts $x = x_1 + x_2$, which satisfy $1 \leq x_1 < 2$ (the m most significant bits of x) and $0 \leq x_2 < 2^{-m}$ (the remaining least significant bits). The Taylor series around the middle of the interval defined by x_1 yields:

$$x^P \approxeq (x_1 + 2^{-m-1})^P + (x_1 + 2^{-m-1})^{P-1} \cdot P \cdot (x_2 - 2^{-m-1}) \tag{11.3}$$

Factoring out the term $(x_1 + 2^{-m-1})^{P-1}$ results in:

$$x^P \approxeq (x_1 + 2^{-m-1})^{P-1} \cdot [(x_1 + 2^{-m-1}) + P \cdot (x_2 - 2^{-m-1})] \tag{11.4}$$

The term $(x_1 + 2^{-m-1})^{P-1}$ is denoted as C, in keeping with the notations in [119], while the

second factor in Equation 11.4 is denoted $x' = (x_1 + 2^{-m-1}) + P \cdot (x_2 - 2^{-m-1})$.

$$\begin{cases} C &= (x_1 + 2^{-m-1})^P \\ x' &= x_1 + 2^{-m-1} + P \cdot (x_2 - 2^{-m-1}) \end{cases} \tag{11.5}$$

C can be obtained through tabulation, using only the m bits of x_1 for addressing. For the values of P that correspond to the three functions, x' can be obtained through bitwise manipulations of x_1 and x_2. Equation 11.4 can be evaluated as:

$$x^P \approxeq C \cdot x' \tag{11.6}$$

at the cost of one addition, one tabulation and a few logic operations, whose complexity depend on the target function. Figure 11.6 presents the architecture of the corresponding arithmetic operator's implementation.

In order to determine the widths of the signals on the datapaths in Figure 11.6, the errors performed first have to be analyzed. The method error of the technique in Equation 11.6 is due to the terms that have been left out in the Taylor series expansion and can be roughly expressed as:

$$\varepsilon_{method} = P \cdot (P-1) \cdot (x_1 + 2^{-m-1})^{P-2} \cdot \frac{1}{2} \cdot (x_2 - 2^{-m-1})^2 \tag{11.7}$$

This results in a bound for the method error of approximately

$$\varepsilon_{method} \approxeq 2^{-2m-3+\log_2(|P\cdot(P-1)|)} \tag{11.8}$$

The method error can be reduced by storing in the C table a correction term. Therefore, in Equation 11.6 C is replaced by:

$$C' = C + P \cdot (P-1) \cdot x_1^{P-3} \cdot 2^{-2m-4}$$

In the architecture of Figure 11.6, the block labeled *'x Modify'* mainly consists of inverting the bits of x_2, and concatenating them together with those of x_1, in the correct order (second line of Equation 11.5). This is true for all the three functions targeted by the generator, the expressions of x' for the recip., sqrt and rsqrt being:

$$\begin{cases} x'_{recip} &= x_1 + 2^{-m} - x_2 \\ x'_{sqrt} &= x_1 + 2^{-m-2} + 2^{-1} \cdot x_2 \\ x'_{rqsrt} &= x_1 + 2^{-m-1} + 2^{-m-2} - 2^{-1} \cdot x_2 \end{cases} \tag{11.9}$$

The computations are performed with g extra guard bits. In terms of implementation, the C' table is of size $2^m \times (w+g)$-bits, where w is the input/output width. The multiplication is of size $(w+g) \times (w+g)$, and a truncated multiplier can be used to obtain the result on $w+g$ bits. Com-

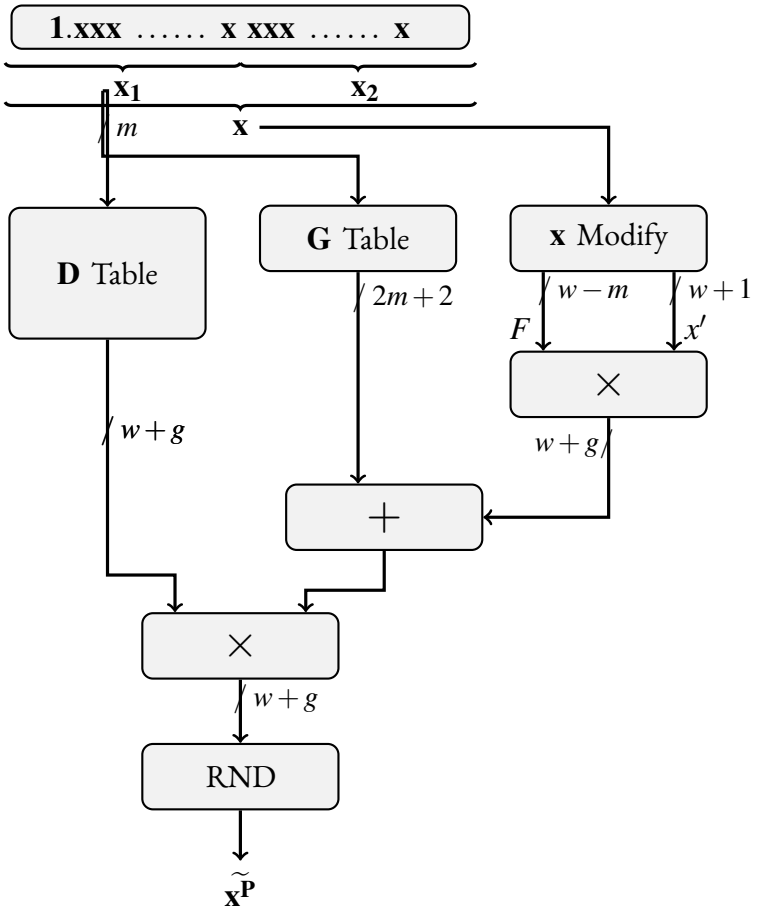


Figure 11.7: Second order tabulate-and-multiply method architecture for the reciprocal function.

pared to what is described in [119], the error analysis reveals that there is a trade-off when choosing the number of guard bits g. Some resource savings can be made when implementing x', by performing $x_2 - 2^{-m-1}$ as a one's complement. This, however requires one more guard bit for g and one more bit for m. The net gain is not obvious, therefore this approach is not chosen and the g can be set to a value of 2, while m can be chosen as $m = \lceil \frac{w}{2} \rceil$.

Second Order Method

The use of only two terms of the Taylor series expansion in the first order tabulate-and-multiply method limits the accuracy that it can achieve. Taking into consideration the constraint $\varepsilon_{method} \approx 2^{-2m-3} < 2^{-w}$, implied that $m \approx \frac{w}{2}$. For the larger precisions, the size of the C' table becomes a limiting factor. The *second order tabulate-and-multiply method* is an extension of the method in Section 11.5.1, and tries to overcome some of its shortcomings.

The method makes use of the first three terms of the Taylor series. As in the case of the first order method, the input x is split into two parts x_1 (the top m MSBs) and x_2 (the remaining LSBs), with $1 \leq x_1 < 2$ and $0 \leq x_2 < 2^{-m}$. Therefore, an approximation around the point $x = x_1 + 2^{-m-1}$ results in:

$$\begin{aligned} x^P \approxeq & (x_1 + 2^{-m-1})^P + (x_2 - 2^{-m-1}) \cdot (x_1 + 2^{-m-1})^{P-1} \cdot P \\ & + \frac{1}{2} \cdot (x_2 - 2^{-m-1})^2 \cdot (x_1 + 2^{-m-1})^{P-2} \cdot P \cdot (P-1) \end{aligned} \tag{11.10}$$

If $(x_1 + 2^{-m-1})^{P-2}$ is factored out:

$$\begin{aligned} x^P \approxeq & (x_1 + 2^{-m-1})^{P-2} \cdot [(x_1 + 2^{-m-1})^2 \\ & + (x_2 - 2^{-m-1}) \cdot (x_1 + 2^{-m-1}) \cdot P + \frac{1}{2} \cdot (x_2 - 2^{-m-1})^2 \cdot P \cdot (P-1)] \end{aligned} \tag{11.11}$$

The terms of Equation 11.11 can be grouped in a more convenient way, resulting in:

$$\begin{aligned} x^P \approxeq & (x_1 + 2^{-m-1})^{P-2} \cdot [(x_1 + 2^{-m-1})^2 \\ & + P \cdot (x_2 - 2^{-m-1}) \cdot (x_1 + 2^{-m-1} + \frac{1}{2} \cdot (P-1) \cdot (x_2 - 2^{-m-1}))] \end{aligned} \tag{11.12}$$

The same kind of notations can be used for the second order method, as in Equation 11.6 for the first order method:

$$\begin{aligned} D &= (x_1 + 2^{-m-1})^{P-2} & x' &= x_1 + 2^{-m-1} + \frac{1}{2} \cdot (P-1)(x_2 - 2^{-m-1}) \\ G &= (x_1 + 2^{-m-1})^2 & F &= x_2 - 2^{-m-1} \end{aligned} \tag{11.13}$$

Therefore, Equation 11.12 can be evaluated as:

$$x^P \approxeq D \cdot [G + P \cdot F \cdot x'] \tag{11.14}$$

In terms of the implementation, the terms D and G can be read simultaneously from a table, indexed by x_1. The term F can be obtained through bit manipulations, by flipping the MSB of x_2 and storing the new sign of F, for later use. The term x' can also be obtained through bit manipulations (and an addition, for rsqrt):

$$\begin{cases} x'_{recip} &= x_1 + 2^{-m} + x_2 \\ x'_{sqrt} &= x_1 + 2^{-m-1} + 2^{-m-3} - 2^{-2} \cdot x_2 \\ x'_{rqsrt} &= x_1 + 2^{-m} - 2^{-m-3} - 3 \cdot x_2 \cdot 2^{-2} \end{cases} \tag{11.15}$$

In addition, a rectangular multiplication is also required, between F and x'. Both of the multiplications can be truncated to a smaller intermediary precision. An illustration of a possible implementation is presented in Figure 11.7, for the recip. function (where $P = -1$).

The method error ε_{method} inherent to the second order tabulate and multiply method is of the order of magnitude of the terms which were left out of the Taylor series expansion, and can be expressed as:

$$\varepsilon_{method} \approx P \cdot (P-1) \cdot (P-2) \cdot (x_1 + 2^{-m-1})^{P-3} \cdot \frac{1}{6} \cdot (x_2 - 2^{-m-1})^3 \tag{11.16}$$

Therefore, the method error can be bounded as $\varepsilon \approx 2^{-3m-5+\log_2|P(P-1)(P-2)|}$ (with P taking the values -1, $\frac{1}{2}$ and $-\frac{1}{2}$ for the recip., sqrt and rsqrt functions, respectively). Compared to the first order method which produces a result that is correct within approximately $2m$ bits, the second order method has an accuracy of approximately $3m$ bits.

The multiplications $F \cdot x'$ and between D and the term between square brackets can be truncated so as to save resources. This entails rounding errors, which means that a larger intermediary precision, using g extra guard bits, is required for the datapaths.

The error budget $\varepsilon_{total} < 2^{-w}$ (where w is the output precision) is classically divided as $\varepsilon_{total} = \varepsilon_{finalRound} + \varepsilon_{method} + \varepsilon_{round}$. The final rounding incurs an error bounded by 2^{-w-1}, which means that the method and the rounding errors must satisfy $\varepsilon_{method} + \varepsilon_{round} < 2^{-w-1}$. Finding a bound on ε_{method} determines the value of m, while finding a bound on ε_{round} gives the number of guard bits g.

Equation 11.14 can be rewritten so as to reflect the sources for the errors: $x^P \approxeq \widetilde{D} \cdot [G - P \cdot \widetilde{F \cdot x'}]$, where the tilded operations are approximations of the true mathematical functions. The multiplication by the factor P does not introduce an error, as the constant is a power of two for the three functions considered. The term G is also without error, as the square can be stored in the table without any approximations. Introducing notations for the error terms, the equation can be further expressed as

$$x^P \approxeq (D + \varepsilon_{tab}) \cdot [G - P \cdot (F \cdot x' + \varepsilon_{mult})] + \varepsilon_{mult}$$

Expanding this expression and grouping the terms gives:

$$x^P \approxeq D \cdot [G - P \cdot F \cdot x'] - D \cdot P \cdot \varepsilon_{mult} + \varepsilon_{tab} \cdot [G - P \cdot (F \cdot x' + \varepsilon_{mult})] + \varepsilon_{mult}$$

from which the rounding error can be identified as:

$$\varepsilon_{round} = -D \cdot P \cdot \varepsilon_{mult} + \varepsilon_{tab} \cdot [G - P \cdot (F \cdot x' + \varepsilon_{mult})] + \varepsilon_{mult}$$

Taking into account the fact that the for the three functions studied the corresponding values of P

are -1, $\frac{1}{2}$ and $-\frac{1}{2}$, and using the upper bounds for the terms D, G, F and x', a bound can be found for ε_{round}:

$$\varepsilon_{round} \leq \overline{\varepsilon_{round}} = \overline{\varepsilon_{mult}} \cdot (1 - P) + \overline{\varepsilon_{tab}} \cdot (1 - P \cdot \overline{\varepsilon_{mult}})$$

Considering that the same internal precision is required for the tabulations and for the truncated multiplications ($\overline{\varepsilon_{tab}} = \overline{\varepsilon_{mult}} = 2^{-w-g}$), and that , then the number of extra guard bits must satisfy:

$$g \geq 2 + \lceil \log_2(2 - P - P \cdot 2^{-w-g}) \rceil$$

This remains, however, a pessimistic bound on g.

Large Precision Methods

Taylor Series-based Method

As the bitwidth of the input increases, methods based on polynomial approximations require an increasing number of terms for satisfying their accuracy requirements, and therefore an increasing amount of operations and resources. The method presented in [41] tries to overcome this issue using an approach made out of three steps:

1. range reduction of the input
2. evaluation of the function on the reduced argument (using a polynomial approximation)
3. post-processing, which reconstructs the final result

The main reason for the argument reduction is to render the second step, the evaluation, less costly. The weight of the higher order terms gets shifted towards the target precision, and therefore removes the need to evaluate some of them.

The range reduction takes the input x, with $1 \leq x < 2$, and returns $x' \in (-2^{-k}, 2^{-k})$, obtained as $x' = x \cdot \hat{x} - 1$, where $\hat{x}$ can be read from a table and is a $k+1$ bit approximation of $\frac{1}{x}$. The $\hat{x}$ is addressed using x's top k most significant bits.

The evaluation step uses a polynomial approximation in order to evaluate the function f (the recip., sqrt or rsqrt functions) at the point x':

$$f(x') = C_0 + C_1 \cdot x' + C_2 \cdot (x')^2 + C_3 \cdot (x')^3 + \ldots$$

The coefficients C_0, C_1, C_2, C_3, ...can be obtained from the Taylor series development. For the evaluation itself, the reduced input x' is split four-wise into $x'_1 \cdot 2^{-k}$ ...$x'_4 \cdot 2^{-4k}$, each of size k. Replacing x'

with its chunks in the expression of $f(x')$ and removing the terms with weights less than 2^{-4k} results in:

$$f(x') = C_0 + C_1 \cdot x' + C_2 \cdot (x'_2)^2 \cdot 2^{-4k} + 2C_2 \cdot x'_2 \cdot x'_3 \cdot 2^{-5k} + C_3 \cdot (x'_2)^3 \cdot 2^{-6k}$$

The last step, the post-processing requires the multiplication by a factor which can be obtained in a similar way as $\hat{x}$. For the recip. function, this term is $\hat{x}$, for the sqrt it is $\frac{1}{\sqrt{\hat{x}}}$, and finally for the rsqrt it is $\sqrt{\hat{x}}$.

There is one main advantage to this method. For the range of input bitwidths it is assigned to in Figure 11.1, on page 169, the small multiplications and table fit well with the resources available on the target FPGA devices. Another advantage is represented by the fact that the pre- and post-computation phases are fairly simple.

Newton-Raphson Iterative Method

The Newton-Raphson method is probably the best known iterative scheme. It is a root-finding method and has quadratic convergence. The method itself can be deduced starting, once again, from the Taylor series expansion around a point x_n, and using only two terms:

$$f(x) = f(x_n) + f'(x_n) \cdot (x - x_n)$$

A root of $f(x)$ satisfies $f(x) = 0$, therefore:

$$f(x_n) + f'(x_n) \cdot (x - x_n) \approx 0$$

This equation can be solved for x, resulting in:

$$x \approx x_n - \frac{f(x_n)}{f'(x_n)}$$

This can be used for the recurrence relation, which equates to:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{11.17}$$

where each x_n is twice as close to the root as the one in the previous iteration.

The error due to the method can be expressed using a recurrence relation:

$$\varepsilon_{n+1} = \varepsilon_n^2 + \frac{f''(x_n)}{2 \cdot f'(x_n)} \tag{11.18}$$

which points to the quadratic convergence of the method.

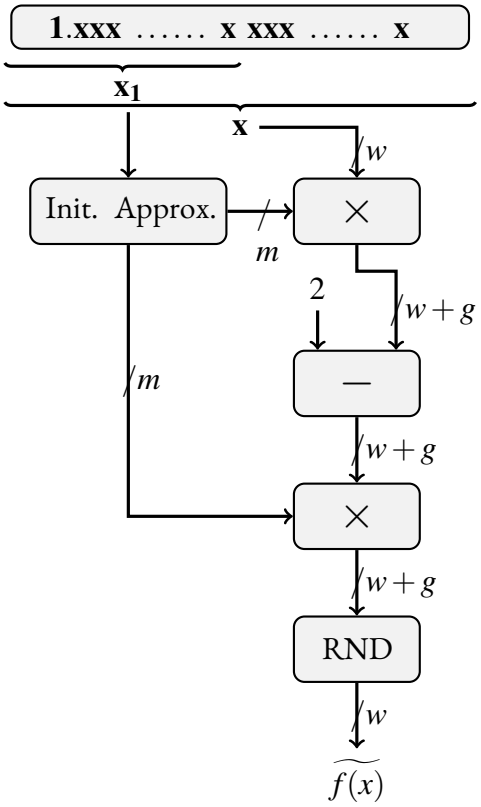


Figure 11.8: Newton-Raphson iterative method architecture for the reciprocal function.

For the sake of clarity, the discussions in the rest of the section are exemplified using the reciprocal function. It is, however, quite straight-forward to adapt the same procedures to the other two functions. The first step in the evaluation is to find the recurrence relation. In order to compute the reciprocal of a number a, a function f which satisfies $f(\frac{1}{a}) = 0$ is required. A good candidate is $f(x) = \frac{1}{x} - a$. Applying the recurrence relation in Equation 11.17 to the candidate function f gives the following iteration:

$$x_{n+1} = x_n \cdot (2 - a \cdot x_n) \tag{11.19}$$

Equation 11.19 can be used for implementing the architecture that evaluates the reciprocal of a number. For the bitwidth ranges on which the method is to be applied (according to Figure 11.1) one iteration is sufficient, as long as the initial approximation is accurate enough. Two truncated multiplications and a subtraction are required, which makes for a fairly simple architecture, as shown in Figure 11.8. In order to ensure that the target accuracy at the output is ensured, an analysis of the errors is performed in the following.

The total error is $\varepsilon_{total} = \varepsilon_{finalRound} + \varepsilon_{method} + \varepsilon_{rounding}$, with the constraint that $\varepsilon_{total} < 2^{-w}$.

The final rounding has an error $\varepsilon_{finalRound} < 2^{-w-1}$. The initial approximation is chosen in such a way so as to be correctly rounded to $m = \lceil \frac{w+1}{2} \rceil$ (which means that the output is accurate to $w+1$ bits). Therefore, the accumulation of rounding errors must satisfy $\varepsilon_{round} < 2^{-w-2}$. Equation 11.19 can be rewritten so as to show the rounding errors:

$$x_{n+1} \approxeq \widetilde{x_n} \cdot (2 - \widetilde{a \cdot \widetilde{x_n}}) \tag{11.20}$$

This can further be expanded as:

$$x_{n+1} = (x_n + \varepsilon_{tab}) \cdot [2 - a \cdot (x_n + \varepsilon_{tab}) + \varepsilon_{mult}] + \varepsilon_{mult}$$

and if the terms are factored:

$$x_{n+1} = x_n \cdot (2 - a \cdot x_n) + \varepsilon_{tab} \cdot (2 - a \cdot (x_n + \varepsilon_{tab})) + \varepsilon_{mult} \cdot (1 + \varepsilon_{tab} + x_n) - \varepsilon_{tab} \cdot a \cdot x_n$$

from which ε_{round} can be identified as:

$$\varepsilon_{round} = \varepsilon_{tab} \cdot (2 - a \cdot (2x_n + \varepsilon_{tab})) + \varepsilon_{mult} \cdot (1 + \varepsilon_{tab} + x_n)$$

If the bounds for the tabulation and multiplication errors are used, $\overline{\varepsilon_{tab}} = 2^{-m-1}$ and $\overline{\varepsilon_{mult}} = 2^{-w-g}$, a bound can be obtained for ε_{round}:

$$\varepsilon_{round} \leq \overline{\varepsilon_{round}} = 2^{-m-1} \cdot (2 - a \cdot (2x_n + 2^{-m-1})) + 2^{-w-g} \cdot (1 + 2^{-m-1} + x_n) \tag{11.21}$$

The first term in Equation 11.21 is of the order of magnitude of 2^{-2m-1} (take into consideration the cancellation between the terms a and x_n). Therefore, a number of $g = 3$ extra guard bits will suffice for the computations on the datapath.

A option to be taken into consideration for the initial approximation is the bipartite method. For up to approximately 32 bits, it falls into the range of bitwidths that are suitable an efficient implementation using the bipartite method. A possible compromise is to add another iteration of the Newton-Raphson method. This results in smaller multiplications and tabulations being required, but increases the latency and double the overall number of operations.

Halley Iterative Method

The Newton-Raphson method of Section 11.6.2 is the first in a class of methods, generally known as the Householder methods. If the Newton-Raphson method has quadratic convergence, bringing at each iteration a one-fold improvement in the accuracy of the result, the generalized method has a

n-fold increase hence a faster convergence rate [19]. The second method in the class is known as the Halley method*, which has cubic convergence.

In order to deduce the method, as with Newton's method, the starting point is Taylor's series, evaluated around the point x_n and using the first three terms:

$$f(x) \approxeq f(x_n) + f'(x_n) \cdot (x - x_n) + \frac{1}{2} \cdot f''(x_n) \cdot (x - x_n)^2$$

Again, a root has to satisfy $f(x) = 0$, therefore:

$$f(x_n) + f'(x_n) \cdot (x - x_n) + \frac{1}{2} \cdot f''(x_n) \cdot (x - x_n)^2 \approxeq 0$$

Regrouping the terms results in:

$$x \approxeq x_n - \frac{f(x_n)}{f'(x_n) + \frac{f''(x_n) \cdot (x - x_n)}{2}}$$

Using the Newton-Raphson iteration, from which $x - x_n \approxeq -\frac{f(x_n)}{f'(x_n)}$, results in the iteration formula:

$$x_{n+1} = x_n - \frac{2 \cdot f(x_n) \cdot f'(x_n)}{2 \cdot (f'(x_n))^2 - f(x_n) \cdot f''(x_n)} \tag{11.22}$$

which is known as the Halley iterative method. The error inherent to the method can be deduced similar to Section 11.6.2 and its recurrence formula is:

$$\varepsilon_{n+1} = \varepsilon_n^3 \cdot \frac{f'''(x_n)}{6 \cdot f'(x_n)} \tag{11.23}$$

which points to the cubic convergence of the method.

In general, the $n+1$-st method in the Householder class of methods has an iteration of the following form:

$$x_{n+1} = x_n + (n+1) \cdot \frac{\left(\frac{1}{f(x_n)}\right)^{(n)}}{\left(\frac{1}{f(x_n)}\right)^{(n+1)}} \tag{11.24}$$

where the superscript $^{(n)}$ signifies the n-th order derivative. Setting n to 0 results in the Newton-Raphson method, while setting it to 1 gives Halley's method, which can be easily verified. In terms of the errors, the Householder methods provide a $n+1$-fold increase in the accuracy of the result at each iteration.

*named after the Edmond Halley, an English astronomer, geophysicist, mathematician, meteorologist, and physicist, best known for computing the orbit of Halley's Comet

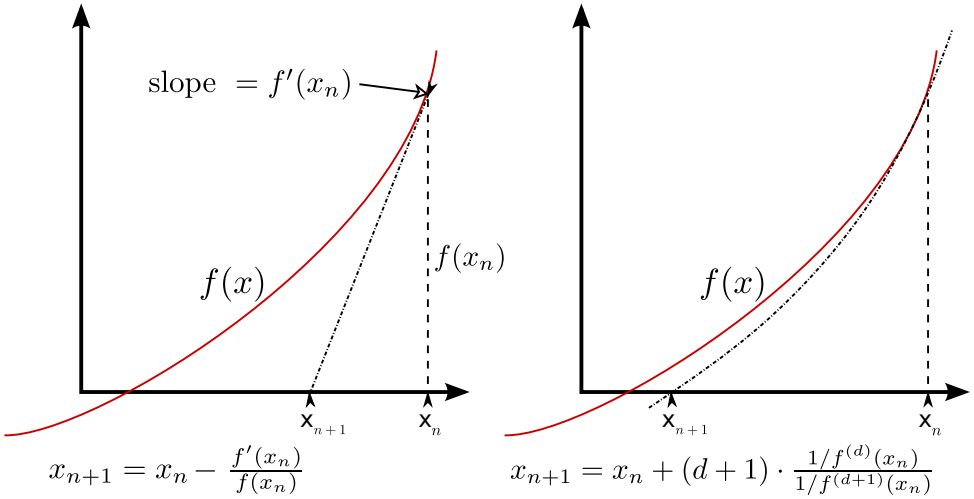


Figure 11.9: A graphical illustration of the Newton-Raphson and Householder methods

A graphical manner of visualizing these methods is presented in Figure 11.9. The Householder method (and therefore the Halley method, as well) replaces the tangent to the plot of the function used by Newton's method with a higher order curve. This curve has a higher number of derivatives in common with the plot of the function at the point where the approximation is made. This means that it fits better the plot, and therefore gives a better approximation.

However, as was also remarked in [45], for the reciprocal the formulation of 11.22 is not particularly useful, due to the cancellations that appear. Using the same function $f(x) = \frac{1}{x} - a$, as with the Newton-Raphson method, results in the need to compute the reciprocal during the evaluation. A slightly different approach can be taken in order to solve this issue and to obtain a second order curve, as remarked in [103] and even further back in [133]. The starting point is again the Taylor series, evaluated around the point x_n. The value of $x - x_n$ can be written out as a power series of $f(x_n)$:

$$(x - x_n) = a \cdot f(x_n) + b \cdot (f(x_n))^2 + c \cdot (f(x_n))^3 + \dots \tag{11.25}$$

Inserting Equation 11.25 in the Taylor series expansion and using the fact that $f(x) = 0$ results in:

$$\begin{aligned} 0 \approxeq & f(x_n) \\ & + f'(x_n) \cdot (a \cdot f(x_n) + b \cdot (f(x_n))^2 + c \cdot (f(x_n))^3 + \dots) \\ & + \frac{f''(x_n)}{2} \cdot (a \cdot f(x_n) + b \cdot (f(x_n))^2 + c \cdot (f(x_n))^3 + \dots)^2 \\ & + \dots \end{aligned} \tag{11.26}$$

Equation 11.26 can be seen as a equation, with $f(x_n)$ as a variable. Therefore, the coefficients of

the terms with the same power on both sides of the equation can be identified, resulting in the following expressions for the a, b, c ...coefficients in Equation 11.25:

$$\begin{aligned} a &= -\frac{1}{f'(x_n)} \\ b &= -\frac{f''(x_n)}{2 \cdot (f'(x_n))^3} \\ c &= -\frac{(f''(x_n))^2}{2 \cdot (f'(x_n))^5} \\ &\dots \end{aligned} \tag{11.27}$$

These values can be replaced in Equation 11.25, which give the following iterative formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{(f(x_n))^2 \cdot f''(x_n)}{2 \cdot (f'(x_n))^3} - \dots \tag{11.28}$$

In order to obtain the iteration for the reciprocal, the function $f(x)$ is chosen as $f(x) = \frac{1}{x} - a$, and therefore:

$$\begin{aligned} x_{n+1} &= x_n \cdot (1 + h_n \cdot (1 + h_n)) \\ h_n &= 1 - a \cdot x_n \end{aligned} \tag{11.29}$$

For the implementation of the architecture, illustrated in Figure 11.10, Equation 11.29 is partially expanded (with h_n keeping the same meaning):

$$x_{n+1} = x_n + x_n \cdot (h_n + h_n^2) \tag{11.30}$$

It can be seen that the $h_n < 2^{-m}$, where m is the size of the initial approximation. The argumentation for this follows the same reasoning as in [41]. A squarer can be used for the computation of h_n^2, and $h_n^2 < 2^{-2m}$. These arguments allow for the use of a smaller squarer and multiplier, when taking into consideration the weight alignment of the terms involved in the operations.

The use of truncated operations implies the use of a larger intermediary precision for the datapath. The total rounding errors must satisfy $\varepsilon_{round} < 2^{-w-2}$, where the rest of the total error budget is accounted for by the error due to the final rounding and the method error. Making the errors explicit in Equation 11.30 gives:

$$x_{n+1} = (x_n + \varepsilon_{tab}) + (x_n + \varepsilon_{tab}) \cdot (h_n - a \cdot \varepsilon_{tab} + \varepsilon_{mult} + (h_n - a \cdot \varepsilon_{tab} + \varepsilon_{mult})^2) + \varepsilon_{mult}$$

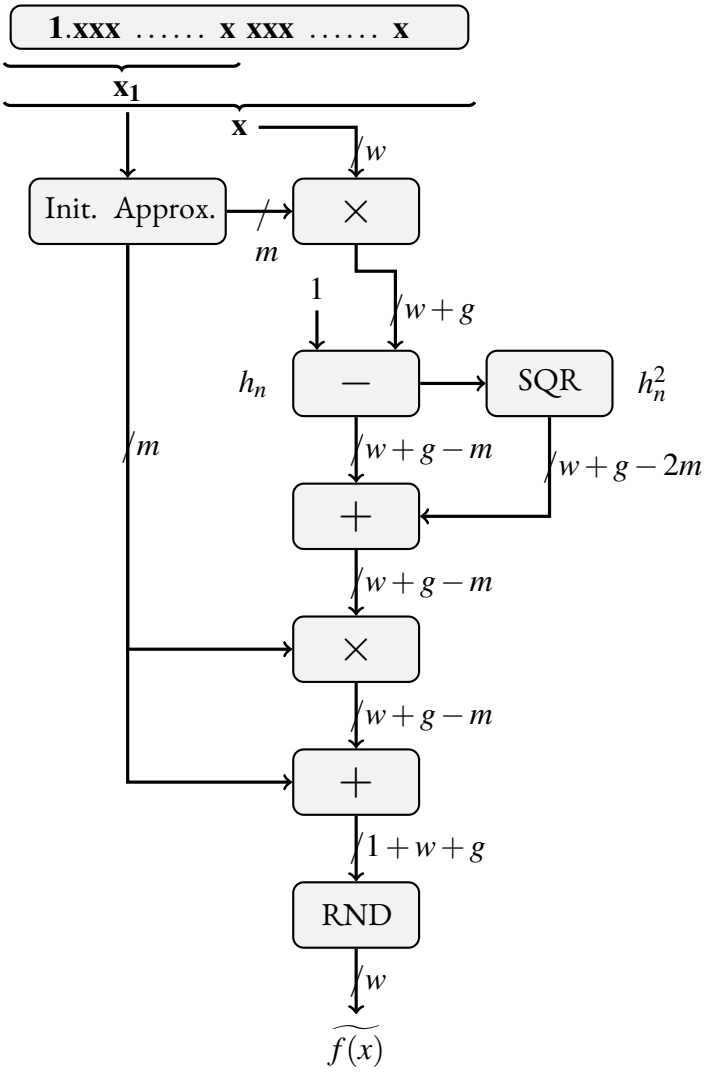


Figure 11.10: Halley iterative method architecture for the reciprocal function.

from which ε_{round} can be identified as:

$$\begin{aligned}\varepsilon_{round} =& \varepsilon_{tab} \cdot (1 + h_n + h_n^2) \\ &+ (x_n + \varepsilon_{tab}) \cdot (a \cdot \varepsilon_{tab} + \varepsilon_{mult}) \cdot (a \cdot \varepsilon_{tab} + \varepsilon_{mult} - 1 - 2 \cdot h_n) \\ &+ \varepsilon_{mult}\end{aligned} \tag{11.31}$$

I/O Format (m,l)	Implementation	Resource utilization and Performance			
		ALMs	DSPs	M10Ks	Lat. @ Freq.
0,-10	Tabulation	1	0	3	2 @ 315MHz
0,-15	Bipartite	27	0	3	4 @ 315MHz
0,-17	Tab. Mult.	54	1	1	5 @ 200MHz
0,-19	Tab. Mult.	48	1	1	6 @ 310MHz
0,-22	Tab. Mult.	58	1	3	6 @ 310MHz
0,-23	Tab. Mult. 1^{st}	64	1	5	6 @ 310MHz
0,-23	Tab. Mult. 2^{nd}	68	2	1	7 @ 250MHz
0,-31	Taylor	282	3	1	23 @ 267MHz
	Bipart.+ NR	188	2	5	15 @ 285MHz
	Halley	155	3	3	15 @ 235MHz
	Bipart.+ Halley	159	3	1	15 @ 228MHz

Table 11.2: Resource utilization and estimated performance of the architectures implemented using the methods studied. The function is the reciprocal. The target device is Cyclone V C8 speedgrade. Input is unsigned, and in the interval $[1,2)$

By substituting ε_{round} and ε_{mult} with their bounds 2^{-m-1} and 2^{-w-g} results in:

$$\begin{aligned}\varepsilon_{round} \leq \overline{\varepsilon_{round}} =& 2^{-m-1}\cdot(1+h_n+h_n^2)\\ &+(x_n+2^{-m-1})\cdot[((-a)2^{-m-1}+a^2 2^{-2m-2}-a2^{-m}h_n)\\ &+2^{-w-g}(1-2^{-w-g}-2h_n+a2^{-m})]\\ &+2^{-w-g}\end{aligned} \tag{11.32}$$

The terms that do not contain 2^{-w-g} result in a term of the order of 2^{-3m-3}. Therefore, imposing a constraint of the form $g > 3$ ensures that a faithful result can be obtained. These findings have also been confirmed through numerical testing.

Results

Table 11.2 shows an overview of all the methods studied. The input is normalized, $x \in [1,2)$, and the bitwidths range from 10 to 32 bits, as per the ranges specified in Figure 11.1, on page 169. The tabulation-based method maps well up to 10 bits, the bipartite method performs well up to 16 bits, the tabulate and multiply performs well up to 24 bits. For large precisions the methods used are: the Taylor-based method, the Newton-Raphson method, using the bipartite method for the initial approximation, and two variations of the Halley method, using either tabulation or the bipartite

method for the initial approximation.

It is worth mentioning that the second order tabulate and multiply method becomes more efficient on the top part of its intended range, as compared to the first order method. This is due to the exponential increase in the size of the tables required. It is also interesting to see the compromises available for the large precision methods. The Newton-Raphson+bipartite method has the highest number of used memory blocks, but also the lowest multiplier block consumption. It also has an average-low logic consumption. The Taylor method has a high multiplier consumption, but a low memory block consumption. It also has the highest logic consumption and by far the longest latency. The Halley method seems worthwhile if used in conjunction with the bipartite method (even if it requires more logic resources), as it reduces the number of memory blocks required. It also has good performance in terms of speed and latency and the lowest logic usage.

Part IV

Digital Filters

Part IV switches to a new application domain: digital signal processing. The filters studied here share many common features and resemble the structure of arithmetic functions, that have been studied up to this point.

The main drive throughout Part IV is the investigation of a methodology for creating digital filters, from specification to hardware implementation, with as little intervention from the designer as possible. All the while, this is set in the same context of FPGA devices, with a focus on correctness at the output and efficiency in the implementation.

One of the main building block for the development of digital filters, the sum of products by constants, is introduced in Chapter 12. Most of the explored filter architectures can be based on it. Chapter 13 discusses finite impulse response (FIR) filters, with an interest in automating the design process, and facilitating the work of the designer. Finally, in Chapter 14 the infinite impulse response filters are studied, with a focus on correct implementations that avoid some of their inherent problems when implemented in hardware.

Money can buy bandwidth, but latency is forever.

John Mashey

12

Sums of Products by Constants

Many digital filters and other signal-processing transforms that can be expressed as a sum of products with constants (SOPC). The implementation of SOPCs is the focus of Chapter 12. Specifically, a SOPC is any computation of the form:

$$y = \sum_{i=0}^{N-1} a_i x_i \tag{12.1}$$

where the a_i are real constants and the x_i are inputs in some machine-representable format.

Some notable examples of this type of computations include the finite impulse response (FIR) and infinite impulse response (IIR) filters, and many other operators used in signal processing, like, for example, the discrete cosine transform (DCT).

The *specification* of the problem is represented by:

- Equation 12.1
- a mathematical definition of each of the real constants a_i

In order to create an *implementation* for an SOPC, finite-precision input and output formats need to be defined. What is also required is a specification of the accuracy of the computation.

Classical design flows often separate these two concerns. However, it must be remarked that there is an obvious connection between them. Consider, as an example, that in the implementation process the real-valued a_i must be rounded to some machine format. But this format naturally depends on the input and output formats, and therefore influences the computation accuracy.

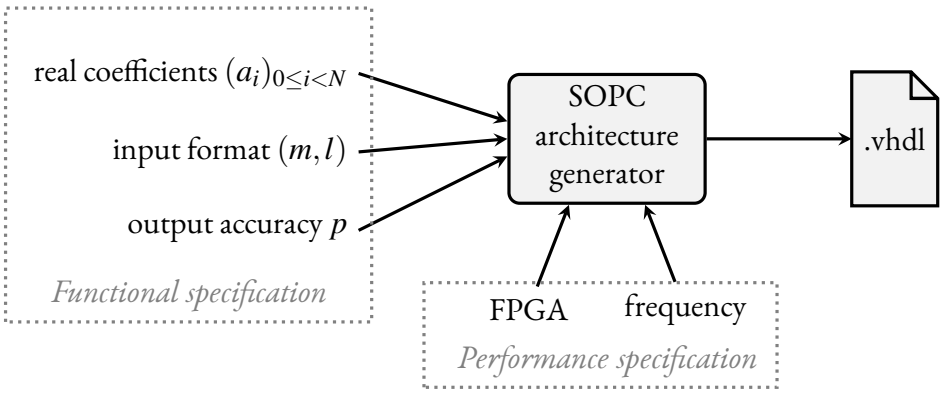


Figure 12.1: Interface to the proposed tool. The integers m, l and p are bit weights: m and l denote the most significant and least significant bits of the input; p denotes the least significant bit of the result.

An explicit link between I/O precision and accuracy can be made. This is motivated, as with the rest of FloPoCo's operators, by a desire to create architectures which are last-bit accurate, i.e. return only meaningful bits. This simple specification, formalized in the following, enables:

- *a simple interface* for the generator. This is shown in Fig. 12.1. It enables the designer to focus on the design parameters which are relevant. Take, as an example, the format to which the a_i constants are rounded. It is not a relevant design parameter: it can be deduced optimally from the mathematical specification and the input/output formats.
- *a fully automated implementation process* that builds provably-minimal architectures with accuracy guarantees.

The resulting operator generator is integrated in FloPoCo's library, under the name `FixSOPC`. Some technical choices lead to logic-only architectures which suit even low-end FPGAs. This choice is motivated by work on implementing the ZigBee protocol standard [57]. The examples used are also from this standard. Other architecture generators can be built is a very similar manner, like for instance those based on the embedded multiplier blocks.

Last-bit Accuracy: Definitions and Motivation

Fixed-point Formats, Rounding Errors, and Accuracy

The error performed during the evaluation of a SOPC architecture in fixed-point is defined as the difference between the computed value $\widetilde{y}$ and its mathematical specification y:

$$\varepsilon \;=\; \widetilde{y} - y \;=\; \widetilde{y} - \sum_{i=0}^{N-1} a_i \cdot x_i \tag{12.2}$$

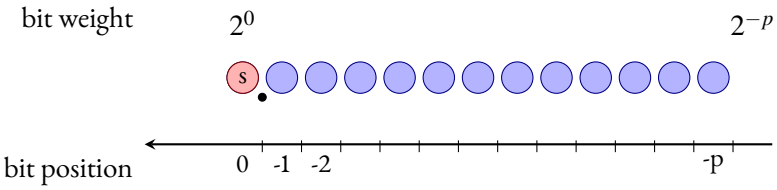


Figure 12.2: The bits of the fixed-point format used for the inputs of the SOPC

where the tilded letters represent approximate or rounded terms (for example $\widetilde{y}$).

An observation that has to be made is that from the point of view of the SOPC, the fixed-point inputs x_i are considered exact. This is even if the x_is are the potential result of some approximate measurement or computation. This is merely a convention.

In the following, the precision of the output format is denoted as p, with the least significant bit taking the value 2^{-p}, as shown in Figure 12.2. The best accuracy that can be achieved for a hardware implementation of Equation 12.1 with a precision-p output, is a perfectly rounded computation with an error bound $\overline{\varepsilon} = 2^{-p-1}$.

However, the constants a_i are arbitrarily accurate, and therefore perfect rounding to a target accuracy may require an arbitrary intermediate precision. This is not feasible for a hardware implementation. Therefore, a slightly relaxed constraint is imposed instead: $\overline{\varepsilon} < 2^{-p}$, and last bit accuracy is used.

The main reason for choosing last-bit accuracy over perfect rounding is that in the context of SOPCs it can be reached with very limited hardware overhead. Therefore, an architecture that is last-bit-accurate to p bits makes more sense than a perfectly rounded architecture to $p-1$ bits, for the same accuracy 2^{-p}. The second reason for using last-bit accuracy is that it ensure that only meaningful bits are computed and implemented, resulting in a fast and efficient implementation.

High-level Specification Using Real Constants

It is important to emphasize that the a_i constants are considered here as *infinitely accurate real numbers*.

In the case of many of the classical signal-processing filters, like the DCT, for example, the coefficients are defined by explicit formulas. This is also true for the ones that are used as examples here: the half-sine FIR and the root-raised cosine FIR [57]. In such cases, last-bit accuracy can be defined with respect to the mathematical specification. Thus the implementation will be as faithful to the true mathematical result as the I/O formats allow.

In other cases, there are no explicit formulas, but the coefficients are provided as double- or even multi-precision numbers (for example those obtained with the use of MATLAB). In such situations,

last-bit accuracy is defined with respect to an infinitely accurate evaluation of Equation 12.1 using the values provided for the coefficients. The resulting architectures behave numerically as if Equation 12.1 was evaluated by MATLAB in double-, or multi-, precision, with a single rounding of the final result to the output fixed-point format.

When converting the specification to finite-precision hardware, last-bit accuracy ensures that the hardware implementation is as close to the specification as possible. In a classical design flow, such a conversion involves *coefficient quantization* and *datapath dimensioning*, which have to be decided by the designer. This puts a considerable amount of effort in the hands of the designer. It is a process which is typically done by trial-and-error, until an implementation with a satisfactory compromise between the signal-to-noise ratio and the implementation cost is achieved. However, optimal choices for these steps can be made automatically from the specification.

Removing the burden of these time-consuming and error-prone tasks from the designer is one of the important contributions of the work presented in Chapter 12. It is definitely a progress with respect to most existing DSP tools.

Tool interface

The ideal interface for an SOPC operator generator, as illustrated in Figure 12.1 should, therefore:

- allow the designer to input the constants $(a_i)_{0\leq i<N}$ to the generator either as mathematical formulas, or as arbitrary-precision floating-point numbers.
- offer a unique integer parameter p, which dictates the accuracy and serves as the definition of the LSB of the output format.

The designer should not have to convert the a_i coefficients to some fixed-point format, nor have to select the format of intermediate computations. These can be deduced from the minimal specification just outlined, and depicted in Figure 12.1, together with the weight of the most significant bit of the output format. This is a direct consequence of the specification of the operator as last-bit accurate.

Ensuring last-bit accuracy

The summation of the products $a_i \cdot x_i$ is illustrated in Figure 12.3. The example used is a 4-tap FIR filter with arbitrary coefficients. Mathematically, the exact product of a real a_i by an input x_i can have an infinite number of bits.

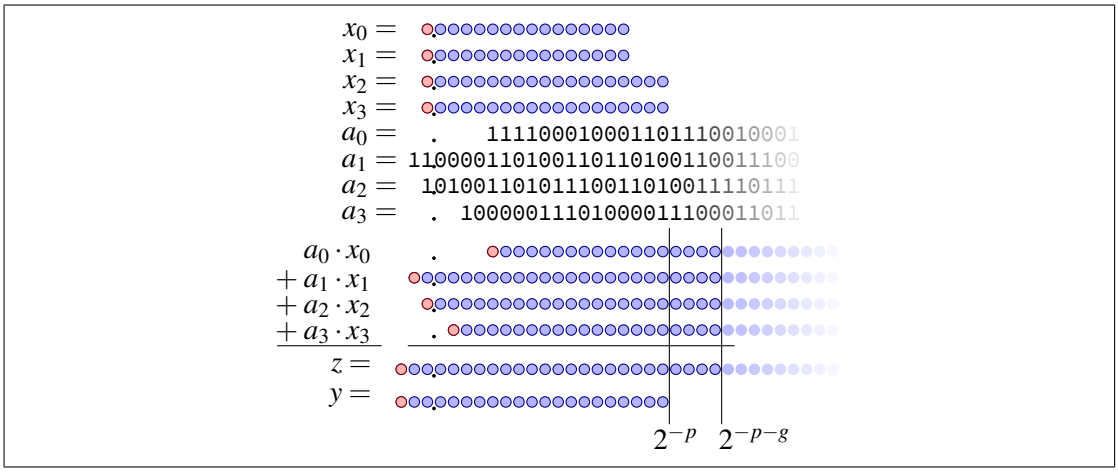


Figure 12.3: The alignment of the $a_i \cdot x_i$ follows that of the a_i

Determining the Most Significant Bit of $a_i \cdot x_i$

It should be remarked that the MSB of the products $a_i \cdot x_i$ is completely determined by the a_i and the fixed-point format of the input. The use of a fixed-point format, means that the inputs can be bounded. Thus, if the domain of x_i is $x_i \in (-1, 1)$, the MSB of $a_i \cdot x_i$ is the MSB of $|a_i|$. This is illustrated by Figure 12.3. In general, if $|x_i| < M$, the MSB of $a_i \cdot x_i$ is $\lceil \log_2(|a_i|M) \rceil$.

When $a_i \cdot x_i$ is positive, this is obvious. When $a_i \cdot x_i$ is negative, this is no longer true, since sign extension is needed in two's complement, up to the MSB of the result. The sign extension can be done using the technique of Section 2.2.2, which is also used in Chapter 6 for the bitheaps. After the sign extension, the variable part of the product has the same MSB as $|a_i|$, just as in the case of positive $a_i \cdot x_i$.

The extra cost of the transformation is the need to add the constant. However, in the context of a SOPC all of these constants can be added together in advance. Thus the overhead cost of supporting signed numbers for SOPCs is limited to the addition of one single constant. A small optimization is the inclusion of this addition in the computations of the $a_i x_i$.

Determining the Most Significant Bit of the Result

A second remark concerns the MSB of the result, which can also be completely determined by the $(a_i)_{0 \leq i < N}$. Starting again with the bound $|x_i| < M$, the MSB of the result can be deduced as:

$$\text{MSB}_{\tilde{y}} = \left\lceil \log_2 \left(\sum_{i=0}^{N-1} |a_i| M \right) \right\rceil \tag{12.3}$$

The upper bound of Equation 12.3 ensures that there will be no overflow at the output. It is also the reason for which the user need not specify the output MSB in the specification of Figure 12.1: providing the a_i suffices.

There are situations when a tighter bound on the result is dictated, due to some additional relationship between the x_i, for instance a consequence of their being successive samples of the same signal. In order to handle this kind of situations, the generator offers the possibility of specifying completely the output format, as additional input parameters. This remains, however, only an option. In a typical context, the generator uses Equation 12.3 to compute the minimal value of the output MSB that guarantees the absence of overflow.

Determining the Least Significant Bit: Error Analysis

Performing all the internal computations to the p precision is not accurate enough, in the great majority of cases. Suppose, for a moment, perfect hardware constant multipliers can be build. They would return the perfect rounding $\widetilde{p_i} = \circ_p(a_i \cdot x_i)$ of the mathematical product $a_i \cdot x_i$ to the target output precision p. However, even such a perfect multiplier has an inherent error, $\varepsilon_i = \widetilde{p_i} - a_i \cdot x_i$, bounded by $\varepsilon_i < \overline{\varepsilon_{\text{mult}}} = 2^{-p-1}$. This is due to the limited-precision output format. The problem is that in a SOPC, rounding errors produced by the constant multipliers add up, and last-bit accuracy can no longer be ensured.

The output value $\widetilde{y}$ is computed as the sum of the $\widetilde{p_i}$. As long as it is performed using adders of a sufficient size, it entails no error. In general, the fixed-point addition of numbers of the same format may entail overflows, but no rounding error. This means that:

$$\widetilde{y} \quad = \quad \sum_{i=0}^{N-1} \widetilde{p_i} \tag{12.4}$$

therefore the total error can be expressed:

$$\varepsilon \quad = \quad \sum_{i=0}^{N-1} \widetilde{p_i} - \sum_{i=0}^{N-1} a_i x_i \quad = \quad \sum_{i=0}^{N-1} \varepsilon_i \tag{12.5}$$

Unfortunately, the accumulation of the ε_i errors reaches $N\overline{\varepsilon_{\text{mult}}}$, which, as soon as $N > 2$, is larger than 2^{-p}. Therefore the naive approach is not last-bit accurate.

The solution consists, again, of using a larger intermediate precision, with g extra guard bits. As a consequence, the error of each multiplier is now bounded by $\overline{\varepsilon_{\text{mult}}} = 2^{-p-1-g}$, which can be controlled through the value of g. Therefore, a value for g has to be determined such that $N\overline{\varepsilon_{\text{mult}}} < 2^{-p-1}$. It is also not necessary to use correctly rounded multipliers, as long as a bound $\overline{\varepsilon_{\text{mult}}}$ of their accuracy can be computed, and this bound is proportional to 2^{-g}.

Seeing how the intermediary computations are performed using $p+g$ bits, a rounding is needed to get to the target output format. In the worst case, the final rounding entails an error $\varepsilon_{\text{final\_round}}$ which is bounded by 2^{-p-1}.

With all these considerations, the total error of a faithful architecture SOPC is therefore:

$$\varepsilon \;=\; \varepsilon_{\text{final\_rounding}} + \sum_{i=0}^{N-1} \varepsilon_i \;<\; 2^{-p-1} + N\overline{\varepsilon_{\text{mult}}} \tag{12.6}$$

ε can be made smaller than 2^{-p}, as soon as the multipliers satisfy:

$$N\overline{\varepsilon_{\text{mult}}} < 2^{-p-1} \tag{12.7}$$

The previous error analysis was fairly independent of the target technology: it could apply to ASIC synthesis as well as FPGAs. Conversely, the following is focused on the LUT-based SOPC architectures for FPGAs context.

An Example Architecture for FPGAs

The context for the following are look-up table-based FPGAs, where the basic logic elements are addressed by α bits. Most of the current generation FPGAs have $\alpha = 6$.

Perfectly rounded constant multipliers

The first option for building a perfectly rounded multiplier is tabulating all the possible products, as there are a finite number of possible values for x_i. This make for a good match, for small input precisions: as long as $x_i \leq \alpha$, each output bit of the perfectly rounded product $a_i \cdot x_i$ consumes exactly one α-input LUT. The advantages, as well as the disadvantages of using tabulation are the same as those exposed in Chapter 8, on page 109, in the discussions introducing the techniques for constant multiplication. The main advantage of using tabulation is that they offer perfect rounding to $p+g$ bits, meaning an error bound $\overline{\varepsilon_{\text{mult}}} = 2^{-p-g-1}$. For real-valued a_i, this is more accurate than using a perfectly rounded multiplier that inputs $\circ_p(a_i)$, which accumulates two successive rounding errors.

Table-Based Constant Multipliers for FPGAs

For larger precisions, the constant multipliers of Chapter 8 using the KCM technique can be used.

The tabulated sub-products $a_i \cdot x_{ik}$ each consume exactly one α-bit LUT per output bit. Seeing as the $a_i \cdot x_{ik}$ products can have an infinite number of bits, in the general case, they are rounded to precision 2^{-p-g}. This rounded value is denoted $\widetilde{t_{ik}} = \circ_p(a_i \cdot x_{ik})$, and illustrated in Figure 12.4.

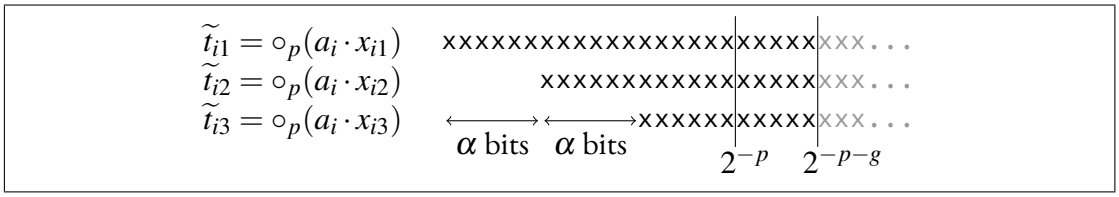


Figure 12.4: Alignment of the terms in the KCM method

In terms of the implementation, not all the tables consume the same amount of resources. The table output $\widetilde{t_{ik}}$ is shifted by a factor of $2^{-k\alpha}$, as illustrated by Figure 12.4, therefore less bits can be stored.

The fixed-point addition is errorless, in this case as well. The error of a constant multiplier implemented in this way is the sum of the errors of the n tables:

$$\varepsilon_{\text{mult}} < n \times 2^{-p-g-1} \tag{12.8}$$

This error is proportional to 2^{-g}, and therefore can be made as small as needed by increasing the value of g.

Computing the Sum

If implemented using classical adders, the LUT cost of the summation for a KCM architecture is roughly proportional to that of the tables, in the context of contemporary FPGAs. Better results can be obtained, however, by stepping back and considering the summation at the SOPC level. The faithful SOPC result is obtained by computing a double sum:

$$\widetilde{y} = \circ_p \left(\sum_{i=0}^{N-1} \sum_{k=1}^{n} 2^{-k\alpha} \cdot \widetilde{t_{ik}} \right) \tag{12.9}$$

Taking into consideration Equation 12.9, it becomes clear that SOPCs are a perfect match for bitheaps. Each of the tables corresponding to the constant multipliers sends its output $\widetilde{t_{ik}}$ to the bitheap, which compresses them. Bitheaps are naturally suited to adding terms with various MSBs, as is the case with SOPCs. The fact that the terms are all aligned to the same LSB makes the implementation that much simpler.

The implementation ca, however still be improved, based on the following observation. In Equation 12.9, the errors of each $\widetilde{t_{ik}}$ add up into an overall SOPC error, out of which the number of guard bits required for each of the KCM multipliers can be computed (denoted in the following as g').

It is often possible, however, to use a finer bound than that of Equation 12.8. There are some constant multipliers which entail no error. Take, for example, the case of the multiplications by 0 and by

1. Such trivial cases can happen if the proposed SOPC generator is used as a back-end for a larger architecture generator. Besides, such trivial cases deserve specific treatment since their implementation is much simpler than the generic case.

Therefore, the implementation first invokes, for each constant, a method that returns the maximum error that will be entailed by a multiplier by its corresponding constant. This error is expressed in units in the last place (ulp), whatever the value of g' will be. The implementation then sums these errors, and uses this sum to compute the value of g' that will enable faithful rounding. Once g' has been determined, the actual construction of the multipliers can be carried on.

The following special cases are managed by the implementation:

- if $a_i = 0$, then $\varepsilon_{\text{mult}} = 0$.
- if $|a_i| = 1$, or more generally if $|a_i| = 2^k$, then $\varepsilon_{\text{mult}} = 0$ if $-p+k \geq -p-g$ (shift of x_i such that all the bits will be kept), otherwise $\varepsilon_{\text{mult}} = 1$ (shift to the right, losing some bits due to truncation). This case can result in an overestimation of the error, as the test should compare using g', which is not yet known.
- in the general case, when the generic KCM architecture is used, $\varepsilon_{\text{mult}} = \frac{n}{2}$ (there are n tables, each entailing one half-ulp of error).

There is one final consideration: it has been assumed so far that the number of tables n is computed out of the input size. However, for small constants, it may happen that the contribution of the lower tables can be neglected. Therefore, the implementation should not generate a table if its MSB is smaller than $2^{p-g'-1}$. The error analysis remains valid in this case, although the source of the error is no longer the rounding of the table, but the table being neglected altogether. If more than one table is fully neglected, this error analysis is slightly pessimistic, but it remains safe.

Figure 12.5 bitheaps for two classical filters from [57], the half-sine and the root-raised cosine. The figure are generated by FloPoCo, and present the bitheaps before their compression.

It can be observed that the shape of the bitheap reflects the various MSBs of the a_i constants. One bit heap is higher than the other, although they add the same number of products, each decomposed into the same number of KCM tables. This is due to special coefficient values that lead to specific optimizations. For instance $a_i = 1$ or $a_i = 0.5$ lead to a single addition of x_i to the bit heap; $a_i = 0$ leads to nothing.

The bitheaps of Figure 12.5 also contain the rounding bit 2^{-p-1}, required for the final rounding, and the sum of the constant bits corresponding to the sign-extensions.

A minor optimization can still be done: these constant bits can be added in advance to all the values of one of the tables, for instance T_{00}. Then the rounding bit comes for free. The sign extension constants may add a few MSB bits to the table output, hence increase its cost by a few LUTs.

The architecture of the generator is depicted in Figure 12.6.

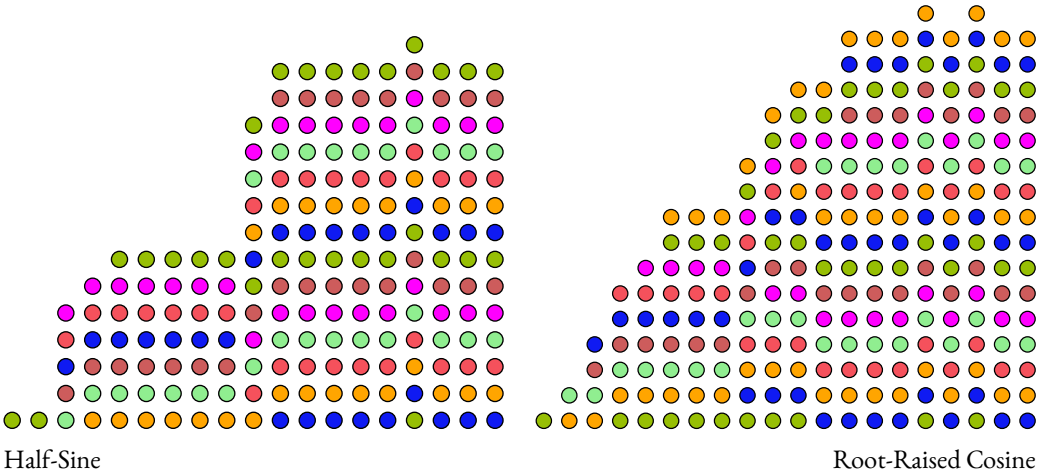


Figure 12.5: Bit heaps for two 8-tap, 12-bit FIR filters generated for Virtex-6

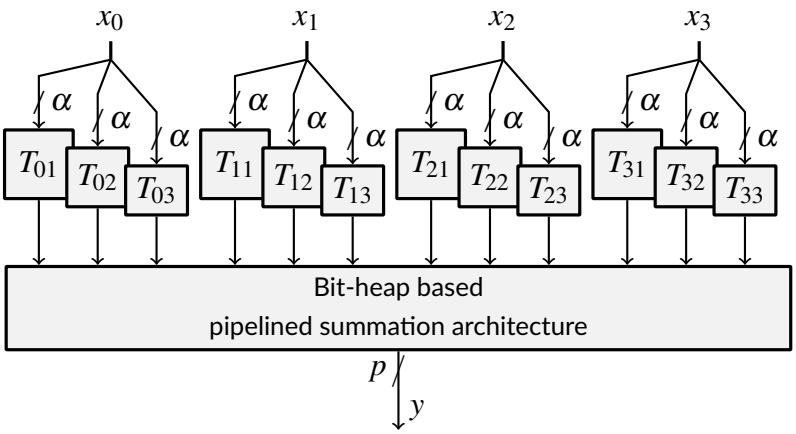


Figure 12.6: KCM-based SOPC architecture for $N = 4$, each input being split into $n = 3$ chunks

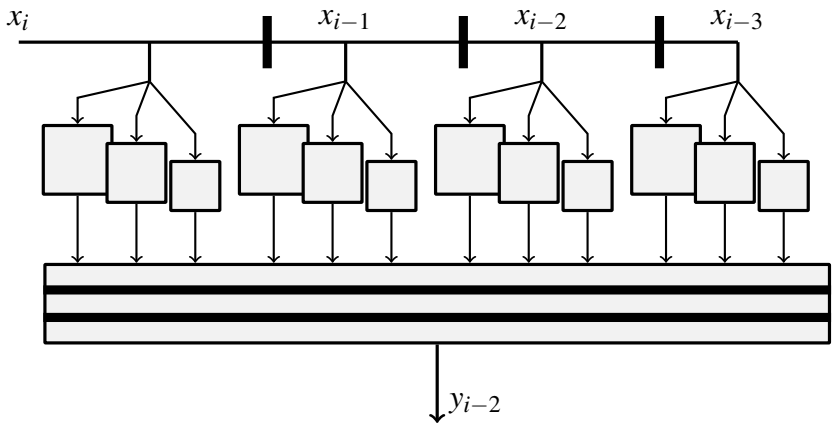


Figure 12.7: A pipelined FIR – thick lines denote pipeline levels

Pipelining

Reading the table values takes one LUT delay, which is the best possible scenario on FPGAs. The summation introduces a more significant delay, which is at most that of $N - 1$ additions. In practice it is much less than that, due to the efficiency of the compression. Pipelining is handled by the bitheap.

Using a SOPC for implementing a FIR results in an architecture like the one illustrated in Figure 12.7. Pipelining the summation come at the cost of an increased latency of the operator.

Implementation and Results

The method described in Chapter 12 is implemented as the `FixSOPC` operator of FloPoCo. The interface of the operator generator is that of Fiure 12.1, and inputs the a_i as arbitrary-precision numbers. Support for mathematical formulas is still somewhat limited and relies on the Sollya library. The operators `FixHS` and `FixRRCF` use the SOPC as a component, and evaluate the a_i coefficients, using their mathematical formulas, for Half-Sine and Root-Raised Cosine respectively. Many other operators of this type are possible.

All the operators have confirmed their last-bit accuracy through extensive simulation, using FloPoCo's testbench framework.

Table 12.1 shows synthesis results for architectures generated by `FixRRCF`, as of FloPoCo SVN revision 2666. All the results use the Xilinx ISE 14.7 suite, and target Virtex-6 (6vhx380tff1923-3) devices. Concerning the results, it should be noted that the register count does not include the input shift register, as this is out of the scope of these comparisons.

Taps	I/O size	Speed	Area
8	12 bits	4.4 ns (227 Mhz) 1 cycle @ 344 Mhz	564 LUT 594 LUT 32 Reg.
8	18 bits	5.43ns (184 MHz) 2 cycles @ 318 MHz	1325 LUT 1342 LUT 92Reg.
16	12 bits	5.8 ns (172 Mhz) 1 cycle @ 289 Mhz	1261 LUT 1257 LUT 41 Reg.
16	18 bits	7.3 ns (137 MHz) 2 cycles @ 265 MHz	2863 LUT 2810 LUT 120 Reg.

Table 12.1: Synthesis results of a Root-Raised Cosine FIR.

The behavior in terms of logic resource consumption is predictable, the dependency of the area to the number of taps being almost linear. An increase in the number of taps brings an increase of the number of needed guard bits. However, more taps means a larger bitheap, with more opportunities for optimization. Still, there is also the dependency of the coefficients a_i themselves to the number of taps.

Conversely, the dependency of the logic resource consumption on precision is more than linear. It is actually expected to be almost quadratic, as can be observed by evaluating the number of bits tabulated for each multiplier (see Figure 12.4).

Interestingly, the dependency of the delay to the number of taps is clearly sub-linear. This is obvious from Figure 12.6. All the tables are accessed in parallel, and the delay is dominated by the compression, which is logarithmic in the bitheap height.

Comparison to a Naive Approach

For evaluating the cost and benefits of a last-bit accurate architecture, a test scenario is devised. It consists in building a variant of the `FixFIR` operator following a more classical approach. It follows many common practices used by designers in the signal processing field. The methodology consists of first quantizing all the coefficients to precision p, such that their LSB has weight 2^{-p}. Fixed-point constants are obtained, and then an architecture is built employing the classical integer KCM algorithm.

Analyzing the errors, it is revealed that the mere quantization to precision p is already responsible for the loss of two bits of accuracy on $\widetilde{y}$. The exact error due to coefficient quantization depends on the coefficients themselves. Then, the multiplications are exact. Figure 12.8 illustrates that their exact results extend beyond precision 2^p, and therefore rounding is required. A choice is made to simply

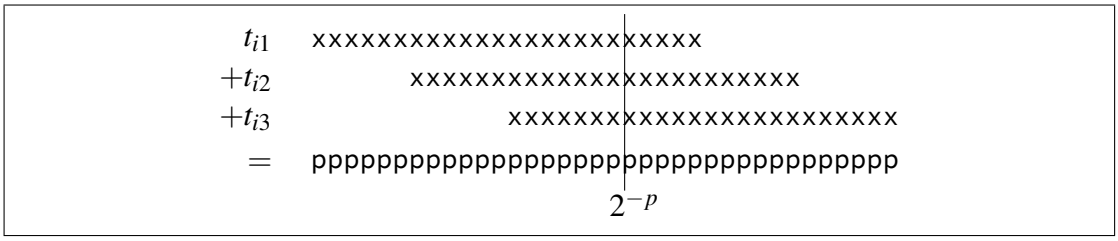


Figure 12.8: Using integer KCM: $t_{ik} = \circ_p(a_i) \times x_{ik}$. This multiplier is both wasteful, and not accurate enough.

truncate the output of each KCM operator to 2^{-p}. However, this more than doubles the overall error. It also allows the synthesis tools to optimize out the rightmost 6 bits of Fig. 12.8. The products are summed using a sequence of $N-1$ adders of precision p.

The results are shown in Table 12.2. The naive approach leads to a much smaller, but slower design. The area difference does not come from the tables, as can be noticed when comparing Figures 12.4 and 12.8. In the case of the last-bit accurate approach, the summation is performed on g extra bits, which explains the difference. Also, the last-bit accurate approach uses a bitheap, whose compression heuristics were at the time mostly optimized for speed, than for area.

However, a meaningful comparison should be made between architectures outputting results with the same numerical quality. The proposed approach is accurate to 12 bits, while the naive approach loses more than 3 bits to quantization and truncation. Therefore, a more meaningful comparison is with the proposed approach, but accurate to $p = 9$, which is also shown in Table 12.2. This version is better than the naive one in both area and speed.

The point to be made here is that it only makes sense to compare designs of equivalent accuracies. And the only accuracy that makes sense is last-bit accuracy, for the reasons exposed in Section 12.1.1 and in Section 2.3.1. This is confirmed by this comparison.

Most current FPGA devices contain dedicated DSP blocks (their contents and sizing is even influenced by the signal processing market). Therefore, an architecture using DSP blocks is also implemented, presented on the last line of Table 12.2. Considering the large internal precision of the DSP blocks, very little effort is required to design a last-bit accurate architecture for output precisions of up to $p = 18$ (using the 24-bit input of the DSP block for the constant, stored with enough guard bits).

An alternative tool is Xilinx CoreGen FIR compiler, which generates only DSP-based architectures. However, it requires the designer to quantize the coefficients and to take all the decisions about the intermediate accuracies and rounding modes, which might prove too much for the average user.

Method	Speed	Area	accuracy
$p = 12$, proposed	4.4 ns (227 Mhz)	564 LUT	$\overline{\varepsilon} < 2^{-12}$
$p = 12$, naive	5.9 ns (170 Mhz)	444 LUT	$\overline{\varepsilon} > 2^{-9}$
$p = 9$, proposed	4.12 ns (243 Mhz)	380 LUT	$\overline{\varepsilon} < 2^{-9}$
$p = 12$, DSP-based	9.1 ns (110 Mhz)	153 LUT, 7 DSP	$\overline{\varepsilon} < 2^{-9}$

Table 12.2: The accuracy/performance trade-off on 8-tap, 12 bit Root-Raised Cosine FIR filters.

Conclusions

Creating last bit accurate sum-of-product with constants architectures two main advantages. First, it gives a clear view on the trade-off between accuracy and performance, freeing the designer from several difficult choices. Second, it actually results in better solutions. The `FixSOPC` operator in FloPoCo's library stands as proof.

Only a small fraction of the vast literature about filter architecture design has been explored, leaving room for many other optimizations. Many other successful approaches exist, in particular those based on multiple constant multiplication (MCM) using the transpose form (where the registers are on the output path) [12, 20, 83, 102, 128]. A technique called Distributed Arithmetic, which predates FPGAs [132], can be considered a generalization of the KCM technique to the MCM problem. From the abstract of [69] that, among other things, compares these two approaches, "if the input word size is greater than approximately half the number of coefficients, the LUT based multiplication scheme needs less resources than the DA architecture and vice versa". Such a rule of thumb should be reassessed with architectures computing just right on each side. Most of this vast literature treats accuracy after the fact, as an issue orthogonal to architecture design.

A repository of FIR benchmarks exists, precisely for the purpose of comparing FIR implementations [4]. Unfortunately, the coefficients there are already quantized, which prevents a meaningful comparison with our approach. Few of the publications they mention report accuracy results.

13

Finite Impulse Response Filters

Up to this point, only the implementation of a filter with the coefficients precomputed has been considered. Approximation algorithms, such as Parks-McClellan, can compute these coefficients, essentially working in the real domain. The question they answer is "what is the best filter with real coefficients that matches this specification".

Chapter 13 presents a new, integrated synthesis framework for finite impulse response (FIR) filters on FPGAs. Written in C++, the main goal is to link state-of-the-art open-source tools to offer guaranteed-quality VHDL filters with best-of-class performance. This is no trivial task, however, as some of these tools work in the frequency domain, while the others work in the time domain.

The work presented in Chapter 13 was done in collaboration with Nicolas Brisebarre and Silviu-Ioan Filip, from the ENS of Lyon, LIP.

Digital Filters: From Specification to Implementation

A digital filter is usually specified in the *frequency domain* (FD) by its **frequency response** . An example of such a specification is shown in Figure 13.1. The implementation of a FIR filter, however, computes sums of products in the *time domain* (TD). An example hardware implementation is shown in Figure 13.2.

The traditional approach for obtaining such an implementation of a FIR filter out of its FD specification typically follows the following three-step process:

1. first, the filter length N and the real coefficients $\{a_i\}_{i=0}^{N-1}$ of an appropriate frequency response

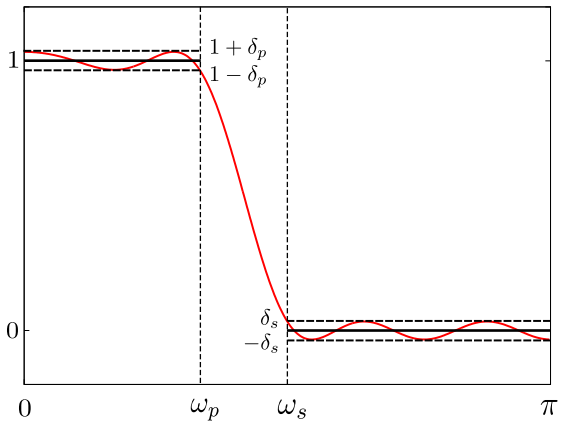


Figure 13.1: Prototype lowpass minimax FIR filter

are determined:

$$H(\omega) = \sum_{i=0}^{N-1} h_i \cdot e^{-ji\omega}, \quad \omega \in [0, 2\pi) \tag{13.1}$$

2. then, the obtained coefficients $\{a_i\}_{i=0}^{N-1}$ need to be quantized to some machine-representable formats (the values $\{\widetilde{a_i}\}_{i=0}^{N-1}$), while trying to minimize the degradation in quality of the frequency response from Equation 13.1;

3. lastly, the computation is implemented in the time-domain:

$$y_k = \sum_{i=0}^{N-1} \widetilde{a_i} x_{k-i} \tag{13.2}$$

where x_i are the input signal values, given in some machine-representable format.

In the following, several previous works that independently address these steps are fused: equiripple designed filters using the Parks-McClellan algorithm [44, 82] for step 1, Euclidean lattice-based coefficient quantization [23] for step 2, and FloPoCo to generate sums of products with constants (SOPC) operators, like in Chapter 12, for step 3.

Implementation Parameter Space

Each of these three steps can be controlled by some parameters, which can be roughly grouped into:

- functional, frequency-domain parameters and constraints (e.g. bands of frequency response);
- architectural, time-domain parameters and constraints: the fixed-point format of x_k (or input format); the format of y_k (output format); the format of the intermediate products and sums, represented by a ? in Figure 13.2;

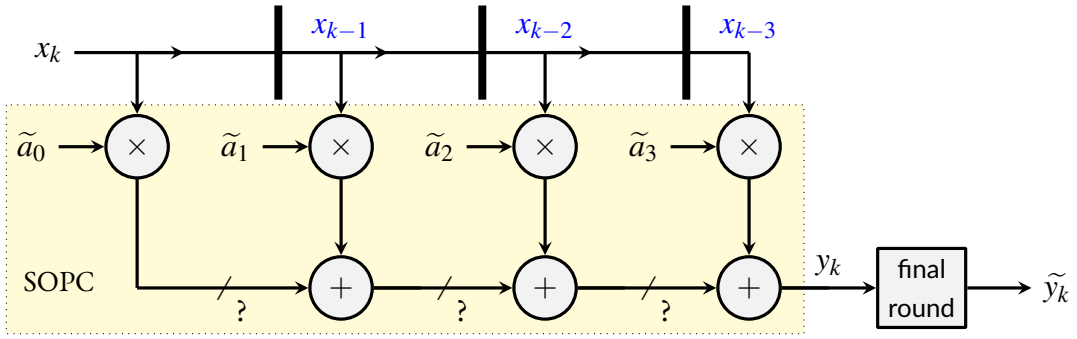


Figure 13.2: A FIR filter architecture

- performance and resource consumption constraints.

Ideally, the quality of a filter is specified in the frequency domain. However, the architectural choices made in the time domain obviously also constrain this quality. For instance, however clever the filter design may be, the quantization noise added by the I/O formats will impose a limit to the filter quality. It is difficult to assess accurately the impact of such time-domain parameter choices in the frequency domain. The opposite is true, as well: however accurate the inputs and outputs may be, the choice of N will limit the quality of the filter, for instance.

To overcome this problem, current mainstream design flows tend to expose all these parameters. To illustrate this, the versatile and widely used MATLAB Signal Processing Toolbox, which goes from the specification to a synthesizable (VHDL or Verilog) hardware description (figure 13.3) is considered in the following.

Alternatives like GNURadio* or the Scipy `signal` package† do not support quantization or fixed-point synthesis. For this reason, they are not considered further. .

From Specification to Architecture in MATLAB

MATLAB offers two similar graphical user interfaces (GUIs), `filterbuilder` and `fdatool`.

For the first step, a precomputed filter can be used, or specifying the frequency constraints for the target filter and letting the tool determine an adequate frequency response. MATLAB uses double-precision arithmetic for this step as a good enough approximation of real numbers (experiments in [44] show that for large filters, it is not sufficient).

With the first step completed, the implementation step can be started. The fixed-point formats for the inputs x_i must obviously be specified, as it defines the interface of the hardware component

*`http://gnuradio.org`
†`http://docs.scipy.org/doc/scipy/reference/signal.html`

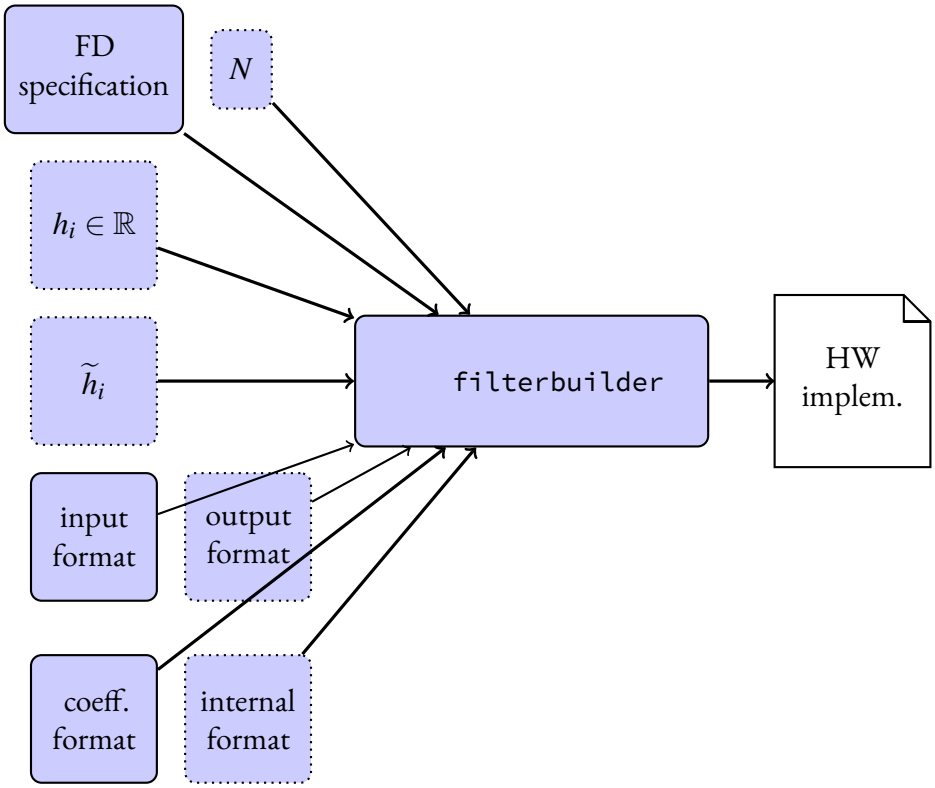


Figure 13.3: MATLAB design flow with all the parameters. The dashed ones can be computed by the tool.

to be generated. The MATLAB flow also demands that the format of the coefficients $\widetilde{a}_i$ be specified. This is a non-trivial task if it is to be ensured that the filter with quantized coefficients still satisfies the frequency-domain constraint. However, it is left up to the user. The `minimizecoeffwl` routine can help, but it does not seem to be integrated directly with either GUI tool, and as such must be used in an external manner.

For the third step, by default the design is carried out in such a way that the output precision of y_k is *maximal*, i.e. there is no rounding in the computation of Equation 13.2. This is usually overkill, and necessarily leads to an output format much larger than the input format, which is not a desirable goal in a signal processing filter. Therefore the output format can also be given, but then it must be specified together with the formats used to store intermediate addition and multiplication results in Equation 13.2.

As Section 13.9 will illustrate, this freedom in terms of parametrization makes the task of controlling *both* output precision and quantization quality difficult, especially when also trying to optimize for hardware resources. If an implementation does not match the specification, it is easy to detect. However, as soon as an implementation matches the specification, it is very difficult to decide if it is

the best one, i.e. the one that minimizes the cost in terms of resource consumption.

Objectives and Outline

The main contribution is to show how near-optimal values of most implementation parameters can be deduced, in an automatic way, from:

1. the FD filter specification
2. the desired input/output formats of the implementation.

A nice side effect is a huge simplification of the user interface, since the user only requires to express these parameters.

The method is automated and based on error analysis performed both in the FD and TD. It builds upon recent open-source efforts that address each of the three steps with guarantees of numerical quality: equiripple designed filters using the Parks-McClellan algorithm [44] for step 1, Euclidean lattice-based coefficient quantization [23] for step 2, and the SOPC operator for step 3. The robustness results from [44] make it possible to address a broader class of FIR filters than MATLAB is able to do. The quantification process presented in [23] directly gives near-optimal quantized coefficients, whereas MATLAB uses a random process whose quality of the output is variable. Finally, the SOPC architecture offers a guarantee on the accuracy of the evaluation of the signal in the time domain.

This integration is non trivial due to the mutual impact of FD and TD parameters on the filter quality. This is detailed in Section 13.8. Section 13.9 shows that this tool leads to highly efficient architectures compared to MATLAB. Before that, Section 13.5 introduces some required concepts, while Section 13.6 surveys relevant previous works on the subject.

The resulting technique is implemented in an open-source C++ tool that is available for download from `http://perso.ens-lyon.fr/silviuioan.filip/icassp.html`.

Background

Many filtering tasks require a frequency response $H(\omega)$ which has linear phase. For FIR filters used in practice, this means that $\{a_i\}_{i=0}^{N-1}$ are chosen to be symmetric or antisymmetric with respect to the middle coefficient(s) [97, Sec. 5.7]. For explanation purposes, only type I linear phase filters are considered in the sequel (*i.e.*, symmetric coefficients and odd N), but the method works in the general case also.

Frequency-Domain Versus Time-Domain Specification

Frequency domain specification of H is generally carried out in terms of a L^∞ formulation, a practice which will also be follow here. Given Ω, a compact subset of $[0,\pi]$, an ideal frequency response D, continuous on Ω, and a weight function W positive and continuous over Ω, H has to be determined such that the weighted error function $E(\omega) = W(\omega)(D(\omega) - H(\omega))$ has minimal uniform norm:

$$\delta = \|E(\omega)\|_{\infty,\Omega} = \sup_{\omega\in\Omega} |E(\omega)| \tag{13.3}$$

In the time domain, the error can similarly be defined for an architecture computing $\widetilde{y_k}$, when supposed to compute y_k, as $\varepsilon(x_k) = |\widetilde{y_k} - y_k|$. The specification will be given as a bound $\overline{\varepsilon}$ on $\varepsilon(x_k)$.

There is a deep relationship between this bound and the output format. How to relate it to the frequency-domain accuracy specification is presented in Section 13.8.

Minimax Filter Design via the Exchange Method

Finding the optimal set of real-valued taps $\{a_i\}_{i=0}^{N-1}$ which minimize δ in Equation 13.3 is a classic Approximation Theory problem. The usual approach to solve it is to use the Remez exchange algorithm [105], which in Signal Processing is more commonly known as the Parks-McClellan algorithm [99]. It is a very robust approach. In the following, the implementation from [44] is used, which is capable of outputting extremely accurate filters.

Depending on the design requirements, the tool can produce:

1. filters of minimal length N satisfying some given error constraints on Ω;
2. for a specified N, filters which ensure a certain target error on some part of Ω.

As an example, consider the design of a prototypical lowpass system, as shown in Figure 13.1, with passband $[0,\omega_p]$ and stopband $[\omega_s,\pi]$. It is defined on $\Omega = [0,\omega_p] \cup [\omega_s,\pi]$ with

$$D(\omega) = \begin{cases} 1, & \omega \in [0,\omega_p], \\ 0, & \omega \in [\omega_s,\pi]. \end{cases} \tag{13.4}$$

For a specification of the first type, the target is to minimize N, while the approximation error $|D-H|$ is upper bounded by δ_p on the passband and by δ_s on the stopband. Taking

$$W(\omega) = \begin{cases} \frac{\delta_s}{\delta_p}, & \omega \in [0,\omega_p], \\ 1, & \omega \in [\omega_s,\pi], \end{cases} \tag{13.5}$$

reduces the problem to finding the smallest N for which the minimax error $\delta \leqslant \delta_s$. The resulting H is shown in red in Figure 13.1. A simple way of determining it requires an initial guess for N. Kaiser [64] gave the formula

$$N \approx \frac{-10\log_{10}(\delta_p\delta_s) - 13}{2.324(\omega_s - \omega_p)},$$

which was later refined in [111]. This degree can be modified iteratively, and the exchange algorithm is applied until $\delta \leqslant \delta_s$. More information can be found for instance in [14, Sec. 15.7–15.9].

When N is fixed, the goal is to bound δ_s by δ (the process for δ_p is analogous). By setting

$$W(\omega) = \begin{cases} w_p, & \omega \in [0, \omega_p], \\ 1, & \omega \in [\omega_s, \pi], \end{cases} \tag{13.6}$$

w_p can iteratively be adjusted until, again, $\delta \leqslant \delta_s$.

A Robust Quantization Scheme

The coefficient quantization problem, although not explicitly controlled by the user, is still present in the proposed method. To address the coefficient quantization issue, the approach described in [23] is used, based on the LLL algorithm [73]. It inputs the value of the quantum 2^p, $p \in \mathbb{Z}$, and considers filters with coefficients of the form $\frac{m_i}{2^p}, m_i \in \mathbb{Z}$. Among these, it quickly finds a filter with minimal (or close to minimal) FD error norm $\sup_{\omega\in\Omega} \left|W(\omega)(D(\omega) - \widetilde{H}(\omega))\right|$.

This technique does not directly provide the optimal coefficient format, but it is fast enough to be iterated over with increasing values of p until the target FD error is matched.

For a type I filter, the constraints are:

$$\widetilde{a}_i = \widetilde{a}_{2n-i} = \frac{m_k}{s}, m_i \in \mathbb{Z}, k = i, \ldots, n,$$

with the m_i's as the unknowns. It remains to determine a filter:

$$\widetilde{H}(\omega) = \sum_{i=0}^{N-1} \widetilde{a}_i e^{-ji\omega},$$

The detailed, in depth, presentation can be found in [23].

The main advantages of this quantization scheme are its scalability to large degrees and the excellent quality of the results it produces, when compared to other quantization methods [23, Sec. 5].

The literature on FIR filters is extremely vast, encompassing decades of advancements in techniques for generating FIR filters. However, references of solutions that generate FIR filters starting from their FD specification are far sparser. Thus, this section focuses on those solutions.

Most major FPGA vendors (like Xilinx, Altera or Actel) provide IP cores for a variety of DSP-oriented tasks, which also include FIR filtering. However, most of the available solutions rely on Matlab/Simulink for the coefficient generation and for testing. Most of the details can, therefore, be deferred to Sec. 13.3. In [56], for example, the authors describe System Generator, a tool which makes the connection between Matlab/Simulink and the hardware generation. System generator provides an abstraction for the hardware, while also integrating with Matlab (e.g. certain design parameters in the design are Matlab functions/variables), on which it relies for generating the filter coefficients.

Open-source alternatives like GNURadio or the Scipy `signal` package essentially address step 1. It is hoped that the present work could establish a missing link to FPGA back-end work [77, 79].

Some of the most relevant references come from the design of VLSI integrated circuits and ASICs, with solutions that go from FD specification all the way to the floorplanning of the circuit. ASCAD [24] is a CAD tool with the goals of reducing the design effort on the designer's side, producing a FIR filter that is correct by construction and as efficient as possible. Their approach combines testing and circuit generation: the designer iterates on the filter design (coefficients, wordlengths) and once it is satisfied, it can pass on to the hardware generation. The designer still needs to input some detailed informations, such as coefficient values and most data formats.

Other later works such as [136], [58], [61], [81] push forward the same general ideas, but manage to also integrate the generation of filter coefficients, or the optimization of the filter order by iterating over the filter design when the FD specification is not satisfied.

The authors of [87] consider an error analysis, but only take into account the errors in the FD domain. In [25] the authors also describe an automatic method for FIR filter generation. In addition, they are also concerned with ensuring the accuracy of their outputs. They start from the FD specification (though it is not clear what they require from the user) and generate the filter coefficients. They then choose fixed-point formats for the coefficients so as to still ensure the output accuracy. The formats are chosen using a random process, and are gradually increased in case the filter specification is not met. More recently, the authors of [112] present as well an automatic generator for FIR filter architectures, also dedicated to FPGAS. Their back-end is based on the RNS, though the details concerning the error analysis in the TD are scarce. Unfortunately, in what coefficient generation is concerned, they refer the user to a reference which is not in English and which is not readily available.

Sum of Product Generation

The third step of the proposed method is based on the SOPCs introduced in Chapter 12 A schematic view is presented in Figure 13.2. The use of such SOPC operators allows for computing the optimal values of all the architectural parameters, and ensure last-bit accuracy of the architecture. The automated error analysis performed for the SOPCs is adapted to the proposed method. A feature of the SOPCs that is exploited here is that constants could be input to the tool as arbitrary real numbers: quantization was part of the architecture generation.

Integrated Hardware FIR Filter Design

This section presents the integrated tool flow which has been implemented. One main objective is to simplify the user interface, however it is important that the designer stays in control of the implementation quality. A discussion of this issue entails in 13.8.1, which identifies two use-cases, each with its distinct interface. Section 13.8.2 presents one of the algorithms, that implements its corresponding minimal interface.

Output Format and TD Accuracy Specification

As stated in Section 13.5.1, the overall TD error of an architecture is $\varepsilon_k = |\widetilde{y}_k - y_k|$, and the error is given as a bound $\overline{\varepsilon}$ on ε_k. There is a connection between $\overline{\varepsilon}$ and the output format, as, on the one hand, it is not possible to output more accuracy than the format can hold. On the other hand, it makes little sense to output less accuracy than the output format allows, especially in a hardware context where every bit has a cost. Therefore, for the proposed method, the accuracy of the architecture is specified by the output format: if the LSB of the output has weight 2^{-p}, then the architecture must be such that $\forall k, |\widetilde{y}_k - y_k| < \overline{\varepsilon} = 2^{-p}$. Conversely, if an architecture is accurate to $\overline{\varepsilon} = 2^{-p}$, then its output format will have LSB 2^{-p}.

This leads to two use cases:

1. A value of N for which the FD specification holds is chosen. A small output format is computed such that the implemented filter satisfies the target FD specification.
2. The user provides the output format. The tool computes a small N such that the implemented filter satisfies the target FD specification.

The first case is easier to implement, but the second is probably more intuitive in a hardware context, when I/O formats are part of the specification. In the following, the focus is only on the second case.

Computing the Optimal Filter Out of the I/O Format Specification

Figure 13.4 shows the proposed automated design flow. The input and output formats and FD quality bounds are given. It might happen that there is no solution with these constraints, in which case the user is alerted. The first step determines an acceptable real-coefficient minimax filter with minimal N. This is to limit the complexity of the implemented filter.

In the next box, the SOPC architecture determines the internal formats it will use to satisfy the TD constraints inherited from the I/O format. This format is also used to quantize the coefficients. This choice can be reevaluated later in the algorithm.

Based on this coefficient format, the coefficients are quantized, still subject to the FD constraints. Since the TD constraints hold for the quantized coefficients (the coefficient format was chosen to guarantee this), the next step in the design flow is to ensure that the FD specifications still hold as well. An example where this is not the case has not yet been encountered, but if needed, one of the following (or a combination of the two) can be done:

1. increase the coefficient and internal formats until the FD constraints are satisfied;
2. increment N and restart the design process from Step 1.

One of these choices leads to the smallest architecture. Eventually, a cost model of the architecture that helps in taking the best decision, will be created. Currently, the first choice is on a loop as long as it improves the FD response. When this is no longer true, the second choice is used and increases N.

Examples and Results

The goal of this section is to highlight, in detail, the design flow and robustness of the proposed approach, when compared to other mainstream design suites.

Working Example and Experimental Setup

As a working example, the default high-pass filter specification of the `filterbuilder` command in MATLAB R2014B illustrates well the scope of our tool. It is a high-pass FD specification with $\Omega = [0, 0.45\pi] \cup [0.55\pi, \pi]$ and

$$D(\omega) = \begin{cases} 0, & \omega \in [0, 0.45\pi] \quad \text{(stop band)} \\ 1, & \omega \in [0.55\pi, \pi] \quad \text{(pass band)} \end{cases} \tag{13.7}$$

The error bounds are specified as a passband ripple below 1dB ($\delta_p \leqslant 0.0575$) and a stopband attenuation of at least 60dB ($\delta_s \leqslant 10^{-3}$). The input format is set to $(w_i, f_i) = (16, 15)$ in MATLAB's fixed-point format notation: total width of 16 bits, out of which 15 fractional bits.

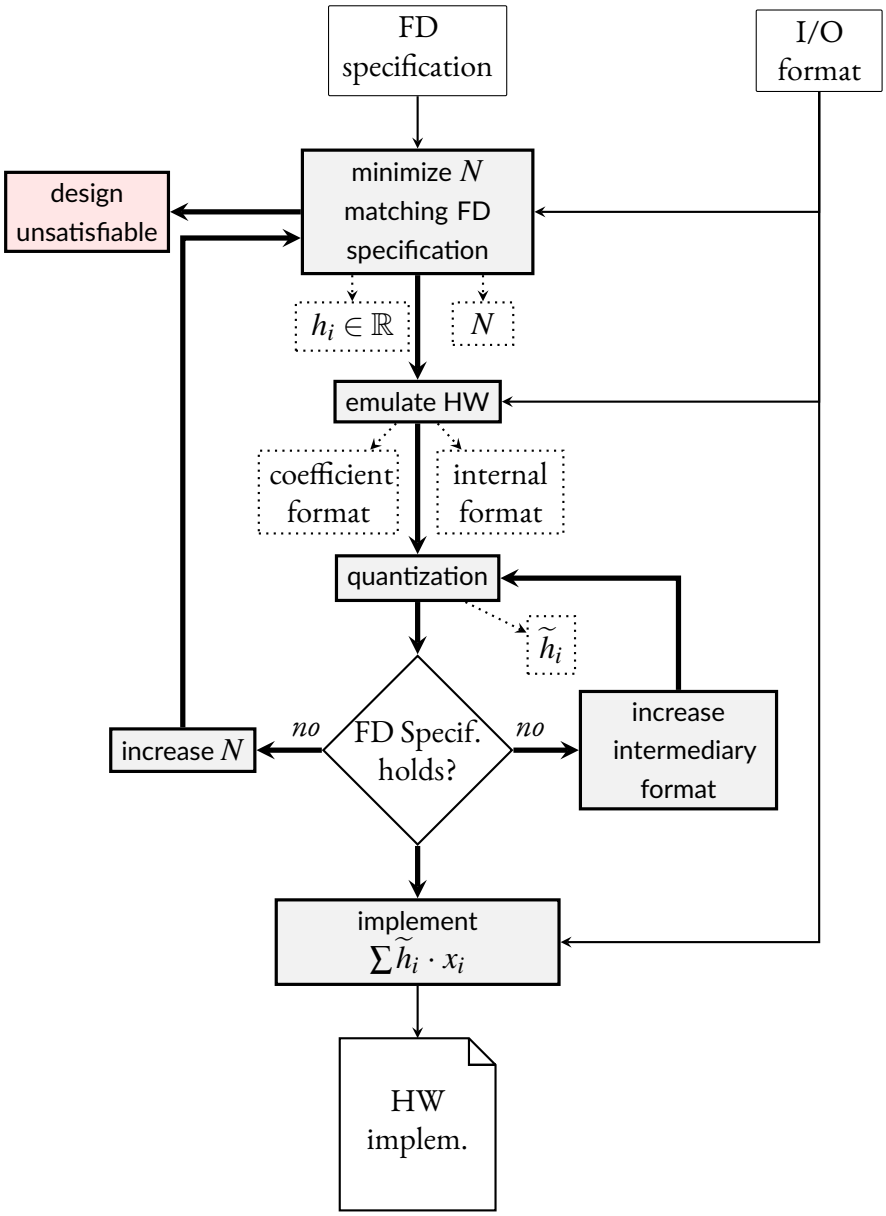


Figure 13.4: Proposed design flow

In order to compare the performance of the method with respect to MATLAB's `filterbuilder` from the hardware point of view, several variants of this filter are synthesized.

Syntheses were performed using the Xilinx ISE 14.7 suite, with the Virtex6 xc6vcx75t-2ff484 FPGA as a target device, and design goals set to balanced. The performance of the designs is classically measured as FPGA resource consumption and maximum operating frequency at a given latency.

In order to evaluate the accuracy of the generated architectures, they are both emulated (the one generated by `filterbuilder` and the proposed one) using the MPFR multiple-precision library, on 10^6 randomly-generated inputs. The maximum absolute difference between the result computed by the architecture (in its internal format, just before the final rounding to the $(16, 15)$ format) and a very large precision simulation of the real filter (before any quantization) was measured. This is referred to this maximum error as "implementation accuracy" in the sequel.

Steps 1 and 2

The context is that of the second scenario of Section 13.8, where the design flow is given by Figure 13.4. The starting point is the determination of a minimal N for which the FD specification remains satisfied when the filter has real coefficients. The `filterbuilder` result is a type I filter with $N = 45$ coefficients. The proposed method finds a type I filter with $N = 43$, which already gives a slightly more efficient design. With coefficients are quantized to 2^{-21}, its passband ripple is smaller than 0.9 dB and a stopband attenuation larger than 61 dB, so it matches the FD specification.

MATLAB's overwhelming choice

Table 13.1 shows how the performance and the resource utilization depend on the three fixed-point formats that appear in Figures 13.3and 13.4. It is a good illustration of the complexity incurred by the vast design parameter space.

The output format and the input format are assumed to be identical, (16,15). It remains to jointly optimize the coefficient format and the internal computation format, for an efficient hardware implementation. The upper part of Table 13.1 is obtained using `filterbuilder`. The table reports results where these formats are set to the input format, then to larger ones. The phrase "full prec" denotes a format on which all the multiplications and additions are be computed exactly (no rounding). The phrase "smart prec" denotes that we reported in `filterbuilder` the format found by the proposed method. It is shown in the sequel how this format can be achieved by trial and error in MATLAB.

The first 8 lines of Table 13.1 keep the coefficient format fixed at $(16, 15)$, while gradually augmenting the precision of the internal computations. It can be seen that the implementation accuracy

reaches a limit at $(22, 21)$, after which incrementing the internal computation precision only increases the resource consumption. The implementation accuracy is also much lower than what is required, the designs providing only 9 meaningful bits, as opposed to the 15 requested in the specifications.

The following two groups of results provide a different scenario. The internal computation format is fixed, at $(16, 15)$ and then at $(23, 21)$, while the coefficient quantization format is varied. As in the previous scenario, the output accuracy reaches a limit. The format $(23, 21)$ is chosen (by error analysis) so as to avoid the accumulation of errors due to the internal computations. The $(23, 21)$ internal format the scenario also shows that once there is enough information in the quantized coefficients, the implementation accuracy requirement can be satisfied.

The last scenario in Table 13.1 first shows the results of leaving the decisions concerning the used formats (internal computations and coefficient quantization) up to `filterbuilder`. The line "full prec" denotes that the tool determines the used formats so as to not have any roundings. The first line provides best implementation accuracy (more than 24 bits), but at a very high cost. This is wasteful, because 8 of these these 24 bits can not be expressed in the output format.

To alleviate this problem, `filterbuilder` is then manipulated so as to use a smaller coefficient quantization format $(22, 21)$, denoted *smart precision* in the table. This precision could be determined by error analysis so that there are no errors propagated in the final results at the given output precision. This choice does increase the efficiency of the design, as can be seen in the table. The next line of results tries to improve the design even further, by reducing the format of the internal computations as well. Strangely, due to implementation choices made in `filterbuilder`, the resource consumption is *higher* than compared to using full precision computations and the design is *less* efficient.

Such an exploration of the parameter space takes time. It is non trivial to obtain from MATLAB the implementation accuracy data, or an equivalent signal/noise measure. The proposed method replaces this search with a-priori error analyses, that are both safer and faster.

Automating the Choice Leads to Better Performance

The last two lines of Table 13.1 present synthesis results of the filter generated using the proposed method. The execution time is less than one minute. It is better than MATLAB-generated filters in resource and performance, while achieving the last-bit accuracy target.

The last line of the MATLAB table and the line obtained by the proposed method use the same parameter set. They are therefore particularly interesting to compare. The accuracy is indeed comparable. The slightly better accuracy of the proposed approach probably comes from the a-priori error analysis that integrates the effect of all rounding errors, including the final rounding to the target format.

I/O Format (w, f)	Coeff. Format (w, f)	Internal Format (w, f)	Resources and Performance			Accuracy (max. abs. error)
			LUTs	Reg.	Lat. @ Freq. cycles @ MHz	
Using MATLAB ($N = 45$)						
(16,15)	(16,15)	(16, 15)	5504 5336	766 1415	2 @ 88.39 8 @ 138.38	1.9397*e*-3 ($2^{-9.009}$)
		(22, 21)	5658 5551	738 1691	2 @ 80.62 8 @ 138.79	1.1393*e*-3 ($2^{-9.777}$)
		(26, 25)	5693 5619	741 1889	2 @ 80.53 8 @ 138.27	1.1451*e*-3 ($2^{-9.770}$)
		full prec. (33, 31)	5180 4713	779 2048	2 @ 95.32 8 @ 156.80	1.1231*e*-3 ($2^{-9.798}$)
	(16, 15)	(16, 15)	5504 5336	766 1415	2 @ 88.39 8 @ 138.38	1.9397*e*-3 ($2^{-9.009}$)
	(22, 21)		8420 8295	760 1402	2 @ 78.87 8 @ 116.13	9.6687*e*-4 ($2^{-10.014}$)
	(26, 25)		10210 10065	751 1408	2 @ 68.81 8 @ 100.39	9.6988*e*-4 ($2^{-10.009}$)
	full prec. (31, 31)		12251 12203	759 1471	2 @ 64.65 8 @ 92.74	9.5673*e*-4 ($2^{-10.029}$)
	(16, 15)	(23, 21)	5727 5551	758 1691	2 @ 77.82 8 @ 138.79	1.1618*e*-3 ($2^{-9.749}$)
	(22, 21)		8712 8503	775 1701	2 @ 72.24 8 @ 118.52	2.6128*e*-5 ($2^{-15.224}$)
	(26, 25)		10419 10458	743 1696	2 @ 64.59 8 @ 104.04	1.5934*e*-5 ($2^{-15.937}$)
	full prec. (31, 31)		12493 12440	794 1756	2 @ 60.29 8 @ 92.80	1.5229*e*-5 ($2^{-16.002}$)
	full prec. (31, 31)	full prec. (33, 31)	13911 13826	775 2198	2 @ 66.06 8 @ 91.93	4.2366*e*-8 ($2^{-24.492}$)
	smart prec. (22, 21)	full prec. (38, 36)	7870 7341	783 2351	2 @ 83.27 8 @ 123.39	1.3245*e*-5 ($2^{-16.204}$)
	smart prec. (22, 21)	smart prec. (23, 21)	8712 8503	775 1701	2 @ 72.24 8 @ 118.52	2.6128*e*-5 ($2^{-15.224}$)
Using the proposed method ($N = 43$)						
(16,15)	*(22,21)*	*(23,21)*	3437 3312	630 710	1 @ 75.78 2 @ 209.05	1.5152*e*-5 ($2^{-16.010}$)

Table 13.1: Syntheses and accuracy measurements of a FIR filter generated using Matlab and the proposed method.

Comparing the resource consumption and the performance justifies the architectural choices made for the FloPoCo operator. MATLAB uses integer shift-and-add multipliers that compute 38 bits, half of which are then rounded out. `FixRealKCM`, on the other hand, computes just the right number of bits, sufficient to achieve the desired accuracy. Besides, the operations are performed on a format (w, f) with $f = 21$, but the total width w depends on the coefficient. Finally, the additions are performed using a bit heap in the SOPC, and using rows of additions in the MATLAB-generated code. Which of these optimizations contributes more to the satisfactory results is difficult to evaluate.

Conclusions

Chapter 13 introduced two new heuristic approaches (one of which detailed in 13.8.2) for automating the fixed-point FIR filter design process for FPGAs. A real-size filter implementation of guaranteed quality is obtained within seconds. Apart from requiring minimal user intervention, the tool produces results which are more accurate, almost twice as fast, and with twice the resource-efficiency than what can be generated with similar mainstream tools.

This result is obtained thanks to a pervasive concern to minimize the size of computations in the resulting architecture, coupled to strict error evaluation techniques to ensure the numerical quality.

The main issue that is raised is the relationship between quality specification in the frequency domain and quality specification in the time domain. This is well understood and well implemented in both domains, with error analysis procedures. And efficient heuristics can be designed, which ensure that the resulting filters meet the specification in both domains. However, the error analysis computed in one domain is not really transfered to the other. Improving on this could allow for a better balance of both error contributions to the overall filter quality, hence even better efficiency.

14

Infinite Impulse Response Filters

The work presented in the following was done in collaboration with Thibault Hilaire, Anastasia Volkova and Benoit Lopez from UPMC Paris, LIP6.

IIR filters are often specified and simulated using high-precision software, before being implemented in low-precision hardware. One of the reasons for this are the quantization and rounding errors. Therefore, a technique is investigated for the construction of IIR, and more generally Linear Time Invariant (LTI), filter implementations which behave as if the computation was performed with infinite accuracy, then rounded to the output format. As with the FIR filters, only a minimalistic specification is required, as many of the design parameters can be deduced optimally in an automatic manner. A detailed error analysis is required for this purposes, in order to capture the rounding errors, but also their infinite accumulation in IIR filters. This analysis then allows the creation of hardware implementations that are able to satisfy their accuracy specification, all at the minimal hardware cost.

Specifying a Linear Time Invariant Filters

LTI filters are very common in signal processing and control, and can be defined as a transfer function in the frequency domain:

$$\mathscr{H}(z) = \frac{\sum_{i=0}^{n_b} b_i z^{-i}}{1 + \sum_{i=1}^{n_a} a_i z^{-i}}, \quad \forall z \in \mathbb{C} \tag{14.1}$$

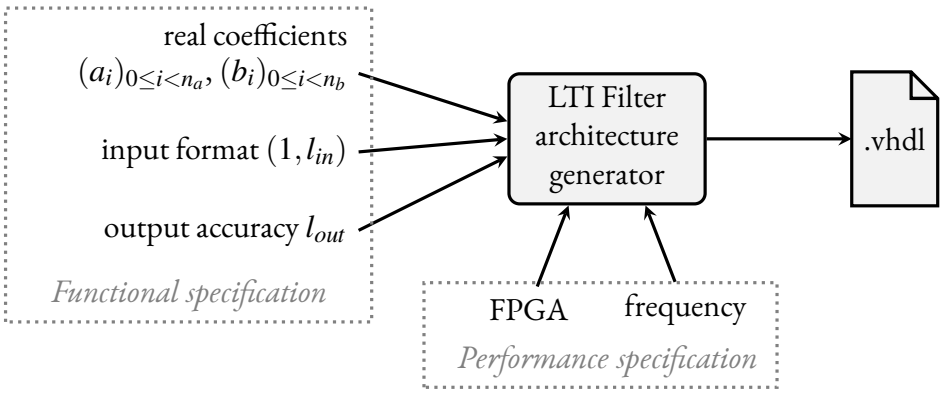


Figure 14.1: Interface to the proposed tool. The coefficients a_i and b_i are considered as real numbers: they may be provided as high-precision numbers from e.g. Matlab, or even as mathematical formulas such as `sin(3*pi/8)`. The integers ℓ_{in} and ℓ_{out} respectively denote the bit position of the least significant bits of the input and of the result. In the proposed approach, ℓ_{out} specifies output precision, but also output accuracy.

The output signal $y(k)$ and the input signal $u(k)$ can in turn be related by the following time domain equation:

$$y(k) = \sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i y(k-i) \tag{14.2}$$

Equation 14.1 or 14.2, together with the mathematical definition of each coefficient a_i and b_i, constitute the *mathematical specification* of the filter. The coefficients a_i and b_i are considered as real numbers, as was the case for the FIR filter, and can be specified in the same manner, as mathematical formulas or multi-precision numbers.

In the following, *implementations* of such kind of specifications are considered, targeting hardware platforms (more specifically FPGAs) and operating on low-precision data (typically 8 to 24 bits).

To create an LTI filter an implementation, the designer first needs to define several parameters in addition to the mathematical specification. Obviously, the finite-precision input and output formats need to be defined, as well as a format for rounding each real-valued coefficient. With the experience of Chapter 13, this is not a trivial task, seeing how for large filter orders (large values of n_a and n_b) or for sensitive filters, the result can become very inaccurate. The other extreme, choosing an extended internal precision, is not acceptable either, as there is risk of obtaining a wasteful architecture in terms of area, time and power, by computing more accuracy than can be output.

Therefore, LTI architectures should be last bit accurate. As with the FIR filters, many of the design decisions can be automated, making the designer's work easier and helping ensure the accuracy claims at the output.

Section 14.4 shows that this accuracy claim can be formalized, and results into an error analysis.

This enables a very simple interface to a LTI filter generator, like the one proposed in Figure 14.1. The designer can therefore once again focus on design aspects which are relevant: the (real) coefficients, and the input/output formats.

The context are implementations targeting FPGAs. The resulting generator is implemented as a new FloPoCo operator. As a side-effect, it also benefits form the infrastructure that FloPoCo has to offer, from automated testing to pipelining, and access to the rest of the operator library.

Definitions and Notations

In the following, fixed-points formats are used for the manipulated data. These are as those described in Section 2.2.1, and can be described by the pair (m,ℓ), which are integers, either positive, or negative.

Due to the finite precision implementation, the exact filter $\mathscr{H}$, with exact output y, cannot be synthesized. Only a filter producing the finite precision output $\widetilde{y}_{\text{out}}$ can be, as shown in Figure 14.3. The overall error, noted ε_{out}, of an architecture that outputs a fixed-point result $\widetilde{y}_{\text{out}}$ is classically defined as the difference between the computed value and its mathematical specification:

$$\varepsilon_{\text{out}}(k) \quad = \quad \widetilde{y}_{\text{out}}(k) - y(k) \tag{14.3}$$

Also, $\overline{\varepsilon}$ denotes a bound on $\varepsilon(k)$, *i.e.* the maximum value of $|\varepsilon(k)|$ over time.

As with SOPCs and FIR filters in the previous chapters, the best that can be done when implementing Equation 14.2 with a precision-ℓ output, is a *perfectly rounded* computation with an error bound $\overline{\varepsilon}_{\text{out}} = 2^{\ell-1}$.

Obtaining a correctly rounded result can require arbitrary intermediate precision, which is unacceptable in an architecture. Therefore, last-bit accuracy is used: $\overline{\varepsilon}_{\text{out}} < 2^{\ell}$.

Specifying the output precision (ℓ_{out} in Figure 14.1 is enough to also specify the accuracy of the implementation.

Worst-Case Peak Gain of an LTI Filter

In order to determine the MSB position of the output (m_{out}) and to be able to perform the error analysis, the amplification of a signal by an LTI filter needs to be computed. This measure is called the *Worst-Case Peak-Gain* (WCPG) [15, 21], and is defined as follows.

Consider a LTI filter $\mathscr{H}$ with input u (bounded by $\overline{u}$) and output y. Then the WCPG of $\mathscr{H}$, denoted $\langle\!\langle \mathscr{H} \rangle\!\rangle$ is defined as the largest peak value of the output y over all possible input u with unitary peak value:

$$\langle\!\langle \mathscr{H} \rangle\!\rangle = \min_{||u||_\infty = 1} ||y||_\infty \tag{14.4}$$

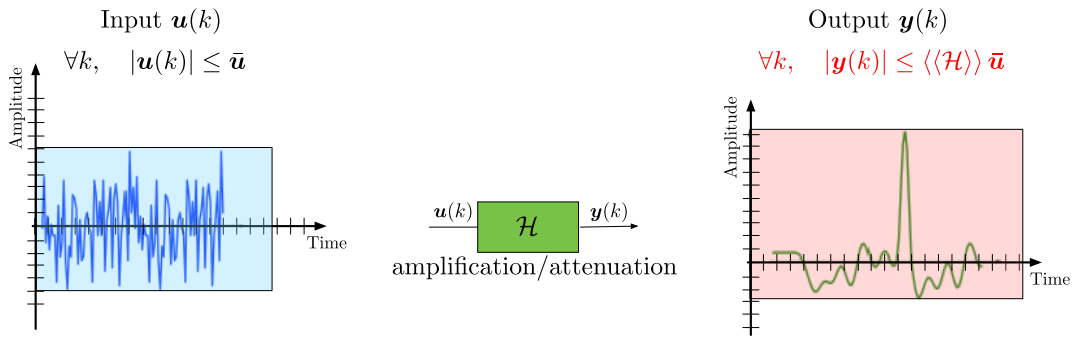


Figure 14.2: Illustration of the *Worst-Case Peak-Gain* (WCPG) Theorem

where $||u||_\infty$ is defined as $||u||_\infty = \max_k |u(k)|$.

Due to the linearity of the filter $\mathscr{H}$ and the property of the impulse response, the output $y(k)$ is a convolution between the impulse response and the input:

$$y(k) = \sum_{l=0}^{k} h(l)u(k-l) \tag{14.5}$$

So,

$$|y(k)| \leq \sum_{l=0}^{k} |h(l)|\bar{u} \tag{14.6}$$

with equality if $\forall 0 \leq l \leq k \quad u(k-l) = \text{sign}(h(l))\bar{u}$. Finally, the WCPG can be computed as can be computed as the ℓ_1-norm of the impulse response h of $\mathscr{H}$, *i.e.*

$$\langle\langle \mathscr{H} \rangle\rangle = \sum_{k=0}^{\infty} |h(k)|. \tag{14.7}$$

The WCPG cannot be computed if the filter $\mathscr{H}$ is not Bounded Input Bounded Output (BIBO) stable [63], *i.e.* if the moduli of the poles of its transfer function are not all strictly smaller than 1 (otherwise, that makes its impulse response not absolutely summable).

The bound $\langle\langle \mathscr{H} \rangle\rangle \bar{u}$ on the output is quite conservative in practice, but it is always possible to exhibit an input $u(k)$ bounded by $\bar{u}$ such that the corresponding output is arbitrary close to its bound $\langle\langle \mathscr{H} \rangle\rangle \bar{u}$.

In the following, the WCPG is computed using the reliable algorithm presented in [126].

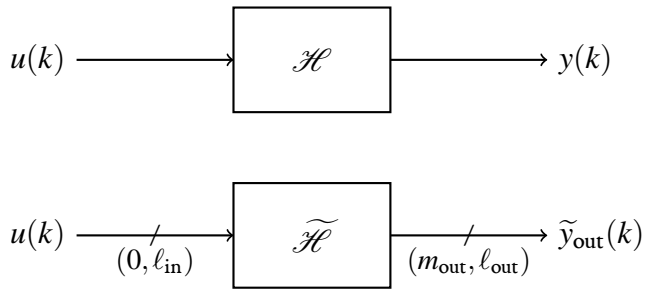


Figure 14.3: The ideal filter (top) and its implementation (bottom)

Error Analysis of Direct-Form LTI Filter Implementations

In the following it is shown how to obtain an implementation of the mathematical definition of an LTI filter given in Equation 14.2 in fixed-point, with last-bit accuracy on the computed result. The ideal and the implemented filters are shown in Figure 14.3.

The considered filters are linear, therefore it can be assumed without loss of generality that the MSB of the input is equal to 0. The MSB of the output m_{out} is therefore defined by:

$$m_{\text{out}} = \lceil \log_2 \langle\!\langle \mathscr{H} \rangle\!\rangle \rceil \tag{14.8}$$

In the worst case, it may happen that rounding errors propagate all the way to the MSB. These errors are bounded by $2^{\ell_{\text{out}}-1}$, therefore the formula to be used is actually:

$$m_{\text{out}} = \left\lceil \log_2 \left(\langle\!\langle \mathscr{H} \rangle\!\rangle + 2^{\ell_{\text{out}}-1} \right) \right\rceil \tag{14.9}$$

In addition, the implementation computes for Equation 14.8 a slight but safe overestimation of $\langle\!\langle \mathscr{H} \rangle\!\rangle$, as shown in more detail in [127].

A larger internal format is used, and instead of computing the ideal output $y(k)$, given in Equation 14.2, an approximation $\widetilde{y}(k)$ of the corresponding SOPC is actually computed:

$$\widetilde{y}(k) \approx \sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \tag{14.10}$$

The final output $\widetilde{y}_{\text{out}}(k)$ is the rounding of the intermediate value $\widetilde{y}(k)$ to the output format.

An illustration of this approach is presented in the abstract architecture of Figure 14.4.

The definition of the overall evaluation error is therefore:

$$\varepsilon_{\text{out}}(k) \quad = \quad \widetilde{y}_{\text{out}}(k) - y(k) \tag{14.11}$$

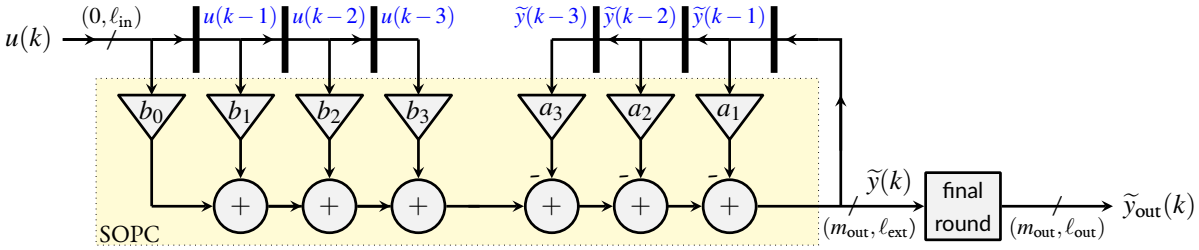


Figure 14.4: Abstract architecture for the direct form realization of an LTI filter

Final Rounding of the Internal Format

The extended format $(m_{\text{out}}, \ell_{\text{ext}})$ used for the internal computations offers additional guard bits in which the rounding errors accumulate. The intermediate result, in the extended format, is rounded to the output format, shown as the "final round" box in Figure 14.4. The entailed error ε_{f}, is defined as:

$$\varepsilon_{\text{f}}(k) = \widetilde{y}_{\text{out}}(k) - \widetilde{y}(k) \quad . \tag{14.12}$$

and can be bounded by $\overline{\varepsilon}_{\text{f}} = 2^{\ell_{\text{out}}-1}$.

It should be noted that the intermediate result $\widetilde{y}(k)$ is fed back on the extended format, not on the output format. This prevents an amplification of $\varepsilon_{\text{f}}(k)$ by the feedback loop, which could compromise the last-bit accuracy.

Rounding and Quantization Errors in the Sum of Products

The coefficients a_i and b_i are real numbers, and rounding them to a finite value before the multiplication can take place is unavoidable. The multiplications also add rounding errors. All the rounding errors can be summarized in a single term $\varepsilon_{\text{r}}(k)$, which is defined as:

$$\varepsilon_{\text{r}}(k) \quad = \quad \widetilde{y}(k) - \left(\sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \right) \tag{14.13}$$

Equation 14.13 should be read as follows. $\varepsilon_{\text{r}}(k)$ measures how much a result $\widetilde{y}(k)$ computed by the SOPC implementation diverges from that of an ideal SOPC (that would use the infinitely accurate coefficients a_i and b_i, and be free of rounding errors). The ideal SOPC would be applied to the same inputs $u(k-i)$ and $\widetilde{y}(k-i)$ as the one used for the implementation.

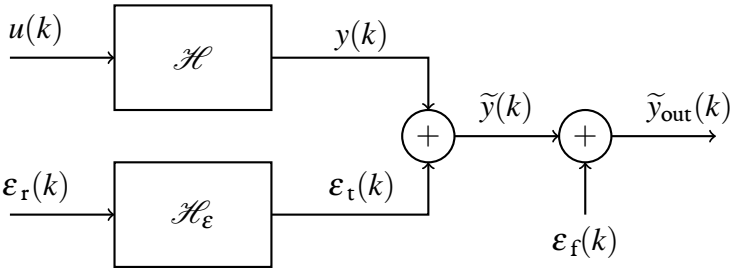


Figure 14.5: A signal view of the error propagation with respect to the ideal filter

Error Amplification in the Feedback Loop

The input signal $u(k)$ is considered exact. However, the feedback signal $\widetilde{y}(k)$ that is input back into the computation, as shown in the architecture of Figure 14.4, differs from the ideal $y(k)$. The entailing error is denoted as $\varepsilon_{\text{t}}(k)$:

$$\varepsilon_{\text{t}}(k) \quad = \quad \widetilde{y}(k) - y(k) \tag{14.14}$$

This error is potentially amplified by the architecture.

Using Equation 14.14, $\widetilde{y}(k-i)$ can be rewritten in the right-hand side of Equation 14.13:

$$\begin{aligned}
\varepsilon_{\text{r}}(k) &= \widetilde{y}(k) - \sum_{i=0}^{n_b} b_i u(k-i) + \sum_{i=1}^{n_a} a_i y(k-i) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \\
&= \widetilde{y}(k) - y(k) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \quad \text{(using Equation 14.2)} \\
&= \varepsilon_{\text{t}}(k) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \qquad \text{(using Equation 14.14)}
\end{aligned} \tag{14.15}$$

from which $\varepsilon_{\text{r}}(k)$ can be deduced as:

$$\varepsilon_{\text{t}}(k) \quad = \quad \varepsilon_{\text{r}}(k) - \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \tag{14.16}$$

This is the equation of an LTI filter inputting $\varepsilon_{\text{r}}(k)$ and outputting $\varepsilon_{\text{t}}(k)$, and with transfer function:

$$\mathscr{H}_{\varepsilon}(z) = \frac{1}{1 + \sum_{i=1}^{n_a} a_i z^{-i}} \tag{14.17}$$

Figure 14.5 illustrates the relationship between the ideal output y, the implemented output $\widetilde{y}_{\text{out}}$ and the different error terms.

The Worst-Case Peak-Gain can now be applied to $\mathscr{H}_\varepsilon$, with input ε_r, in order to bound ε_t by:

$$\overline{\varepsilon}_t = \langle\!\langle \mathscr{H}_\varepsilon \rangle\!\rangle \, \overline{\varepsilon}_r \tag{14.18}$$

Therefore, $\overline{\varepsilon}_t$ can now be made as small as needed by increasing the internal precision ℓ_{ext}, which, in turn, reduces $\overline{\varepsilon}_r$.

Putting It All Together

Equation 14.11 can be rewritten as:

$$\begin{aligned} \varepsilon_{out}(k) &= \widetilde{y}_{out}(k) - \widetilde{y}(k) + \widetilde{y}(k) - y(k) \\ &= \varepsilon_f(k) + \varepsilon_t(k) \end{aligned} \tag{14.19}$$

hence

$$\begin{aligned} \overline{\varepsilon}_{out} &= \overline{\varepsilon}_f + \overline{\varepsilon}_t \\ &= \overline{\varepsilon}_f + \langle\!\langle \mathscr{H}_\varepsilon \rangle\!\rangle \, \overline{\varepsilon}_r \end{aligned} \tag{14.20}$$

Last-bit accuracy imposes the constraint: $\overline{\varepsilon}_{out} < 2^{\ell_{out}}$. The final rounding adds an error bounded by $\overline{\varepsilon}_f = 2^{\ell_{out}-1}$. For faithful rounding, the error $\overline{\varepsilon}_t$ of the filter before final rounding has to be bounded by $2^{\ell_{out}-1}$. This leads to the following constraint on $\overline{\varepsilon}_r$:

$$\overline{\varepsilon}_r < \frac{2^{\ell_{out}-1}}{\langle\!\langle \mathscr{H}_\varepsilon \rangle\!\rangle} \tag{14.21}$$

Equation 14.21 can be used to compute the LSB ℓ_{ext} of the intermediate result. Assuming that an SOPC can be built faithful to any value of ℓ_{ext}, for it will satisfy $\overline{\varepsilon}_r < 2^{\ell_{ext}}$. Therefore, the constraint of Equation 14.21 holds if:

$$2^{\ell_{ext}} < \frac{2^{\ell_{out}-1}}{\langle\!\langle \mathscr{H}_\varepsilon \rangle\!\rangle} \tag{14.22}$$

The optimal value of ℓ_{ext} that ensures this constraint is:

$$\ell_{ext} = \ell_{out} - 1 - \lceil \log_2 \langle\!\langle \mathscr{H}_\varepsilon \rangle\!\rangle \rceil \tag{14.23}$$

The implementation of this error analysis actually uses a guaranteed overestimation of $\langle\!\langle \mathscr{H}_\varepsilon \rangle\!\rangle$ [126]. This ensures that rounding errors in the the computation of $\langle\!\langle \mathscr{H}_\varepsilon \rangle\!\rangle$ itself do not jeopardize the accuracy. Because of this, the computed value of ℓ_{ext} may very rarely be one less than the mathematical value defined in Equation 14.23. This has no impact in practice.

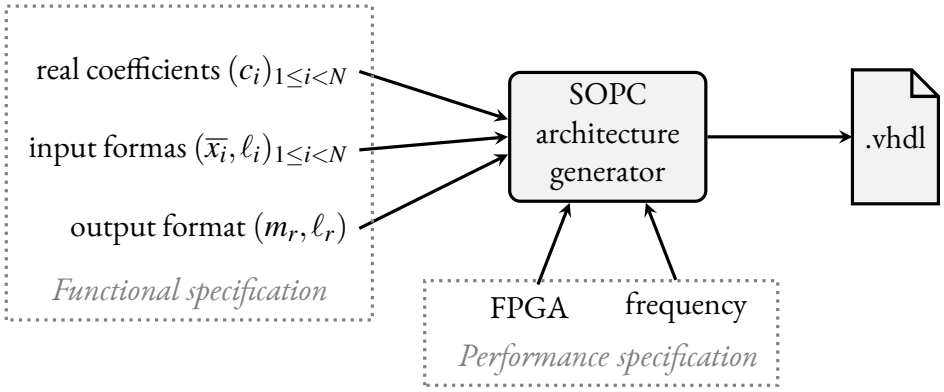


Figure 14.6: Interface to the SOPC generator in the context of LTI filters

Meanwhile, the MSB of the internal format is the same as that of the result, m_{out}. Some overflows may occur in the internal computation, but since the computation is performed modulo $2^{m_{\text{out}}}$, the final result will be correct.

Sum of Products Computations for LTI Filters

In the following, the problem of building a SOPC architecture for the LTI filters is addressed, *i.e.* an architecture computing:

$$r = \sum_{i=1}^{N} c_i x_i \tag{14.24}$$

for a set of real constants c_i, and a set of fixed-point inputs x_i.

In Chapters 12 and 13, all the inputs x_i shared the same format. In the context of LTI filters, this is no longer true. Figure 14.4 shows a single SOPC, where the c_i may be a_i or b_i, and the x_i may be either some delayed u_i, or some delayed y_i. The format of y_i, as previously determined, is in general different from that of the u_i.

Therefore, a more generic interface to the SOPC generator is needed, where the format of each input can be specified independently. This interface is shown on Figure 14.6. Specifically, the LSBs of the inputs are provided as ℓ_i. For the MSBs, instead of specifying m_i, the interface uses the maximum absolute value $\overline{x_i}$ of each x_i, which provides more information, which is used in the sequel (indeed some information is lost, about the sign, but this is not important for building the SOPC).

Another difference with Chapter 12 is that the output MSB m_r is input to the generator. An overestimation of m_r can be computed out of the c_i and the input formats, as in Chapter 12. However, the worst case peak gain of an IIR filter provides a more precise value of m_r. This value is therefore provided to the SOPC generator.

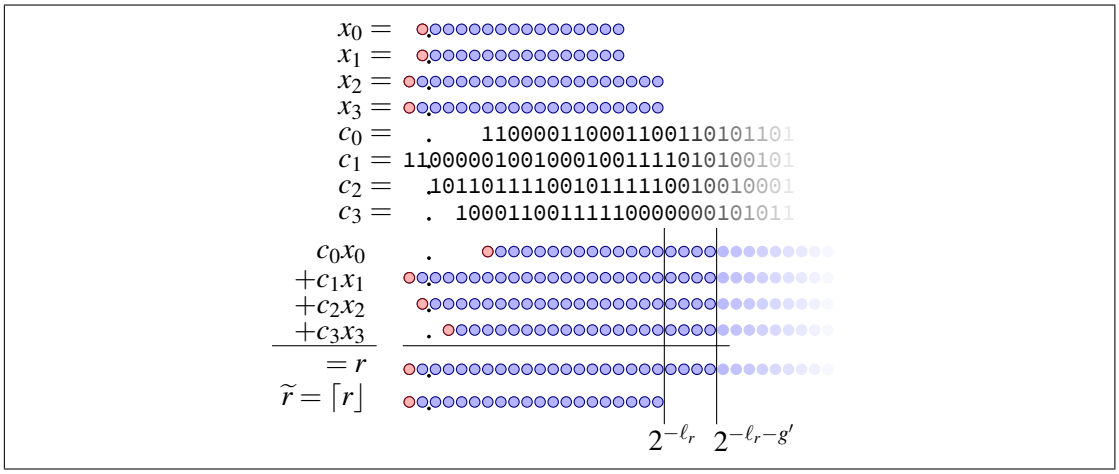


Figure 14.7: Alignment of the $c_i x_i$ for fixed-point x_i and real c_i

Here again, ℓ_r also specifies the accuracy of the SOPC. In the following it is shown how to build an SOPC accurate to 2^{ℓ_r}, which was assumed to be known in the previous section.

Error Analysis for a Last-Bit Accurate SOPC

The fixed-point summation of the terms $c_i x_i$ is illustrated in Figure 14.7, with the presented interface to the SOPC. A 4-input SOPC of an IIR filter of order 2, with arbitrary coefficients, is used as an example. It is a smaller version of the one depicted in Figure 14.4, where x_0 and x_1 are $u(k)$ and $u(k-1)$, respectively, while x_2 and x_3 are $\widetilde{y}(k-1)$ and $\widetilde{y}(k-2)$. The output r is $\widetilde{y}(k)$.

The error analysis is similar to the one done originally for the, SOPC, with only a few changes.

The MSB of the products $c_i x_i$ can be easily determined out of the value of c_i and $\overline{x_i}$, as $|x_i| \leq \overline{x_i}$. Therefore $|c_i x_i| \leq c_i \overline{x_i}$, so the MSB of $c_i x_i$ can be computed as $\lceil \log_2(|c_i \overline{x_i}|) \rceil$. This is where using $\overline{x_i}$ instead of an MSB specification for x_i can save one bit. To anticipate possible overflows due to rounding, the implementation must add, before taking the $\log_2$, an upper bound of its rounding error.

Negative $c_i x_i$ need to be sign extended, and the same technique of Section 2.2.2 can be used.

In general, performing the internal computations on the output precision ℓ_r would not be accurate enough, due to the accumulation of rounding errors. As in the case of Chapter 12, an extended precision for the internal computation needs to be used. Determining the number of extra guards to be used follows the same reasoning as in the case of the analysis originally done for the SOPC.

For the implementation of the multiplications, the KCM constant multipliers of Chapter 8 can be used, with the same optimizations and considerations as in Chapter 12.

The summation of the products, can again be performed using a bitheap.

f_c	guard bits	Speed	Area
0.6	7	175 MHz	170R + 1388L
0.7	8	168MHz	180R + 1621L
0.8	10	163MHz	190R + 1912L
0.9	15	169 MHz	215R + 2623L
0.95	19	159MHz	235R + 3141L

Table 14.1: Needed guard bits and synthesis results for some 12-bit, 5th-order Butterworth filters.

Implementation and results

The proposed method is implemented as the `FixIIR` operator of FloPoCo. `FixIIR` offers the interface shown on Figure 14.1, and inputs the a_i and the b_i as arbitrary-precision numbers.

To test `FixIIR`, a small Python script uses `numpy` and `scipy.signal` to generate the double-precision coefficients for Butterworth filters. Such wrappers can easily be written for other classical filter families.

Thanks to this script, several 5th-order Butterworth low-pass filters were generated for 12-bit signals, with increasingly values of the normalised cutoff frequency f_c. Table 14.1 shows how the number of guard bits computed by `FixIIR`, and the corresponding area and operating frequency.

`FixIIR`, like most FloPoCo operators, was designed with a testbench generator [37]. All these operators reported here have been checked for last-bit accuracy by extensive simulation.

These results were obtained for Virtex-6 (6vhx380tff1923-3) using ISE 14.7.

Part V

Conclusions and Future Works

15

Conclusions and Future Works

The work presented throughout this thesis has in common a unique vision of what the arithmetic for reconfigurable circuits should be. Computations should always be meaningful. This ensures that the end-user can fully take advantage of the computed information, at the minimal cost. The common context for this thesis are coarse arithmetic operators, explored in two different directions: arithmetic functions (algebraic and elementary, univariate and multivariate) and digital filters (finite and infinite impulse response). With the goal of fair comparisons, all methods were developed with the same level of dedication and optimizations.

Arithmetic Functions. The analysis of methods for function evaluation has brought innovative, efficient and high-performance solutions. They manage to provide results that offer an accuracy guarantee, while making the best possible use of the resources offered by the target device. Among the findings of the explorations of the solution space, there are valuable remarks to be made. It is interesting to observe how some classical techniques for the evaluation of functions, known and in use for some time now, manage to have an absolute cost that is similar to modern, state-of-the-art methods. As shown for the elementary and algebraic functions studied in this thesis, CORDIC and Taylor series approximations can fit in these category. At the same time, it can also be observed that, in spite of this positive finding, they do not adapt particularly well to the target architecture, which results in an underutilization of the hardware's potential. The newly developed techniques, on the other hand, perform much better from this point of view and are, therefore, also better adapted for the future evolution of the FPGAs.

The FloPoCo project provided the perfect setting for developing the elementary (sine, cosine and

arctangent) and the algebraic (reciprocal, square root and reciprocal square root) functions. This also means that they are contributions to the generator's library, which can help ensure that FloPoCo remains among the most important open-source alternatives for generating arithmetic operators. With the increasing amount of attention that FPGAs are receiving, it is even more important that they have good tools, easily accessible for the users. One can only hope that FloPoCo counts among them.

And in this spirit, the work presented throughout this thesis should open new doors. The trigonometric functions offer an easy starting place, as many of the techniques presented, and the design flow itself, can be applied to other trigonometric functions. Among these, the tangent/cotangent, or the hyperbolic functions spring to mind. In reality, what the work on the trigonometric functions can really provide is a set of techniques and best practices for arithmetic core development, which have been selected, or developed where necessary, so as to best suit their context.

The same is true for the algebraic functions. The methods presented can very easily be extended to powering functions. Here, in addition to just providing the methods, the corresponding chapter presented a methodology for choosing the one that fits best in a given context, out of the many available ones. Therefore, the same overall operator design methodology was used in two different tools.

While for univariate functions many state ot the art techniques have made their way into hardware design, the same cannot be said about the multivariate functions. This is partially be due to the lack of optimal approximation methods. Even worse, many of the available techniques from software are ill-suited for hardware, leaving a big question mark as to what exactly is the right approach. Future works in this direction are definitely worth their time.

Traditionally, FloPoCo has been oriented towards fully unrolled, pipelined architectures. Iterative solutions could therefore be a new direction to explore. Their main allure consists in the possibility of having a small computing kernel, that comes at the cost of increased latency, due to the need to iterate on the kernel. Iterative methods would allow the attack of large precision methods, a domain where unrolled methods start to be prohibitive, due to their increasing resource demands.

Digital Filters. In what signal processing concerned, the works presented in this thesis only considered the direct form for the implementation of the digital filters. There are, however, many possible forms, each with its advantage. An investigation between these possibilities would result in very interesting comparisons, and could possibly answer some fundamental questions as to which of the forms is the most appropriate, given a specific filter. On a finer-granularity level, the implementation that was matched to the direct form was based on the idea that the constant multiplications were independent. It would be interesting to consider them as a single operation, a problem known in the literature as the multiple constant multiplication (the MCM problem). Large operators offer many optimization opportunities, and there are plenty to be made for the MCMs.

Most works throughout the literature consider the problem of designing a filter as two independent ones: one in the frequency domain, and one in the temporal domain. It is, however, unfair to consider the two as independent problems. They are superseded by a more generic one, which is how to connect the error analyses in the frequency and the temporal domains, in a quest to create the ultimate filter generator, that follows the compute-just-right ideology. A considerable step in this direction is presented in this thesis. The efforts made for developing FloPoCo's filter infrastructure could push it forward as the missing link to bridge many excellent works done in software, for the generation of filters, to a hardware back-end.

Arithmetic Optimizations: the Bitheap Framework. Development of implementations for many of the discussed methods, as well as many that were omitted, was greatly eased by the bitheap framework. From of a developer's point of view, many of the tedious tasks such as alignments, extensions, signal manipulations and probably countless more, are almost completely eliminated, or rendered that much more trivial. A boost in ease of development and productivity is a great time saver and a major opportunity. However, porting the whole FloPoCo project to this new framework is a considerable task to consider and undergo. As might have also been remarked from the implementations described throughout this thesis, some arithmetic cores take advantage of the bitheap framework, and some do not. This is due to the fact that they predate it. The operators in FloPoCo were also one of the main motivations for the bitheap framework. The upside is that the bitheap framework is confirming itself as the universal tool that it promises to be, with a majority of the framework relying on it.

The evolution of the bitheap framework is far from over, either. Optimal strategies for compression are yet to be explored, and could provide gains to all operators that are based on bitheaps. New and ever more ingenious compressors can still be added, as well. One of the main outcomes of the bitheap framework is that it has restarted the research into optimal compression strategies, compressors and the arithmetic operators that can be based around it. An example are the works presented in [70, 71], which are implemented using FloPoCo.

It is also interesting to consider up to which level should the bitheap be connected to its underlying hardware platform. This defines, in a way, the level of abstraction that it provides. One of the facets of the bitheap that is yet to be explored is that, on a very basic level, the bitheap can be seen as a measure of the complexity of arithmetic operators. Comparisons between different implementations of the same operators are complicated, and trade-offs usually need to be made. Comparing different operators is even more complicated. The bitheap could be a solution to this problem.

The Generator. For fine-grain and coarse operators to reach their maximum potential in terms of performance, pipelining is essential, essentially in the context of unrolled operators. This process

is complicated, even at a small scale, and becomes a bottleneck to the performance very quickly. Automation is the answer. It is interesting to explore the possibilities left in this constrained scenario, where the timing information is used as a design-time parameter. Though not seemingly so, this is a difficult problem. But pipelining a design should not be limited to the currently chosen strategy. Once the circuit is built, a new strategy can be created, which only influences the timing, and not the structure, thus maintaining the compatibility with the circuit created initially. In a way, this acts as a retiming of the circuit. Retiming is also an open opportunity, though a complex and difficult one.

Future Directions. Not only the evolution of the FloPoCo project has to be taken into account, but also the evolution of the underlying hardware platforms. It has become relatively clear, at this stage, that FPGAs are evolving towards increasingly heterogeneous devices. A process that started with the introduction of the multiplier blocks, current FPGAs contain a large proportion of resources that are not logic elements. Among the game-changer decisions are the introduction of floating-point blocks, which begs the question of which should be the direction in which arithmetic core generators should evolve, and at what level of granularity should they exist. This is also influenced by the current trend of a switch towards higher-level languages for design specification. With it, the tools are also making the change. So the question arises as to what are the opportunities for arithmetic core generators, in the context of this switch?

Finally, FloPoCo was initially designed for FPGA devices. However, the generated operators are portable, and the used algorithms can be manipulated to fit other given constraints than those imposed by FPGAs. Therefore, other target platforms, such as ASIC circuits, can be foreseeable in the future development of the project. It is interesting therefore to consider the new challenges that these devices bring to an arithmetic core generator. ASICs have a different set of constraints, offering more degrees of freedom to the designer.

References

[1] (1985). *XC2000 Logic Cell Array Families*. Xilinx Corporation.

[2] (1994). *XCELL vol. 15*. Xilinx Corporation.

[3] (2009). *CORDIC v4.0 (DSD249)*. Xilinx Corporation.

[4] (2009). FIR suite. http://www.firsuite.net/.

[5] (2016a). *7 Series DSP48E1 Slice User Guide (UG479)*. Xilinx Corporation.

[6] (2016b). *7 Series FPGAs Configurable Logic Block User Guide (UG474)*. Xilinx Corporation.

[7] (2016c). *7 Series FPGAs Memory Resources User Guide (UG473)*. Xilinx Corporation.

[8] (2016a). *Stratix 10 Embedded Memory User Guide (ug-s10-memory)*. Intel.

[9] (2016b). *Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide (ug-s10-lab)*. Intel.

[10] (2016c). *Stratix 10 Variable Precision DSP Blocks User Guide (ug-s10-dsp)*. Intel.

[11] Ahmed, H. M. (1982). *Signal Processing Algorithms and Architectures*. PhD thesis, Stanford University. AAI8220419.

[12] Aksoy, L., Costa, E., Flores, P., & Monteiro, J. (2008). Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6), 1013–1026.

[13] Andraka, R. (1998). A survey of CORDIC algorithms for FPGA based computers. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, FPGA '98 (pp. 191–200). New York, NY, USA: ACM.

[14] Antoniou, A. (2005). *Digital Signal Processing: Signals, Systems, and Filters*. McGraw-Hill Education.

[15] Balakrishnan, V. & Boyd, S. (1992). On computing the worst-case peak gain of linear systems. *Systems & Control Letters*, 19, 265–269.

[16] Banescu, S., De Dinechin, F., Pasca, B., & Tudoran, R. (2011). Multipliers for floating-point double precision and beyond on FPGAs. *ACM SIGARCH Computer Architecture News*, 38(4), 73–79.

[17] Baugh, C. R. & Wooley, B. A. (1973). A two's complement parallel array multiplication algorithm. *IEEE Transactions on Computers*, 22(12), 1045–1047.

[18] Beaumont, O., Boudet, V., Rastello, F., Robert, Y., et al. (2002). Partitioning a square into rectangles: Np-completeness and approximation algorithms. *Algorithmica*, 34(3), 217–239.

[19] Borwein, J. M. & Borwein, P. B. (1987). *Pi and the AGM: A Study in the Analytic Number Theory and Computational Complexity*. New York, NY, USA: Wiley-Interscience.

[20] Boullis, N. & Tisserand, A. (2005). Some optimizations of hardware multiplication by constant matrices. *IEEE Transactions on Computers*, 54(10), 1271–1282.

[21] Boyd, S. & Doyle, J. (1987). Comparison of peak and RMS gains for discrete-time systems. *Systems Control Letters*, 9(1), 1–6.

[22] Brandolese, C., Fornaciari, W., & Salice, F. (2004). An area estimation methodology for FPGA based designs at systemc-level. (pp. 129–132).

[23] Brisebarre, N., Filip, S.-I., & Hanrot, G. (2016). A lattice basis reduction approach for the design of quantized FIR filters. submitted.

[24] Cappello, P. R. & Wu, C.-W. (1987). Computer-aided design of VLSI FIR filters. *Proceedings of the IEEE*, 75(9), 1260–1271.

[25] Chan, S. C., Tsui, K. M., & Zhao, S. H. (2006). A methodology for automatic hardware synthesis of multiplier-less digital filters with prescribed output accuracy. In *APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems* (pp. 61–64).

[26] Chapman, K. D. (1994). Fast integer multipliers fit in FPGAs. *Electronic Design News*, 5.

[27] Chen, D. & Sima, M. (2011). Fixed-point CORDIC-based qr decomposition by givens rotations on FPGA. In *2011 International Conference on Reconfigurable Computing and FPGAs* (pp. 327–332).

[28] Chung, A. J., Cobden, K., Jervis, M., Langhammer, M., & Pasca, B. (2014). Tools and techniques for efficient high-level system design on FPGAs. *CoRR*, abs/1408.4797.

[29] Chung, F., Gilbert, E., Graham, R., Shearer, J., & van Lint, J. (1982). Tiling rectangles with rectangles. *Mathematics Magazine*, 55(5), 286–291.

[30] Dadda, L. (1965). Some schemes for parallel multipliers. *Alta frequenza*, 34(5), 349–356.

[31] Das Sarma, D. & Matula, D. (1995). Faithful bipartite rom reciprocal tables. In *ARITH'12* (pp. 17–28).

[32] Dawson, R. & Paré, R. (1993). Characterizing tileorders. *Order*, 10(2), 111–128.

[33] de Dinechin, F., Detrey, J., Creţ, O., & Tudoran, R. (2007). *When FPGAs are better at floating-point than microprocessors*. Technical Report ensl-00174627, ÉNS Lyon. http://prunel.ccsd.cnrs.fr/ensl-00174627.

[34] de Dinechin, F. & Didier, L.-S. (2012). Table-based division by small integer constants. In *International Symposium on Applied Reconfigurable Computing* (pp. 53–63). Berlin, Heidelberg: Springer Berlin Heidelberg.

[35] de Dinechin, F., Joldes, M., & Pasca, B. (2010a). Automatic generation of polynomial-based hardware architectures for function evaluation. In *ASAP 21st IEEE International Conference on Application-specific Systems, Architectures and Processors* (pp. 216–222).

[36] de Dinechin, F. & Pasca, B. (2009). Large multipliers with fewer dsp blocks. In *2009 International Conference on Field Programmable Logic and Applications* (pp. 250–255).

[37] de Dinechin, F. & Pasca, B. (2011). Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4), 18–27.

[38] de Dinechin, F., Takeugming, H., & Tanguy, J.-M. (2010b). A 128-tap complex FIR filter processing 20 giga-samples/s in a single FPGA. In *44th Asilomar Conference on Signals, Systems & Computers*.

[39] de Dinechin, F. & Tisserand, A. (2001). Some improvements on multipartite table methods. In *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15* (pp. 128–135).

[40] Detrey, J. & de Dinechin, F. (2007). Floating-point trigonometric functions for FPGAs. In *International Conference on Field Programmable Logic and Applications* (pp. 29–34).

[41] Ercegovac, M., Lang, T., Muller, J.-M., & Tisserand, A. (2000). Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *Computers, IEEE Transactions on*, 49(7), 628–637.

[42] Ercegovac, M. D. & Lang, T. (2004). *Digital arithmetic*. Elsevier.

[43] Farabet, C., Poulet, C., Han, J. Y., & LeCun, Y. (2009). Cnp: An FPGA-based processor for convolutional networks. In *2009 International Conference on Field Programmable Logic and Applications* (pp. 32–37).

[44] Filip, S. I. (2016). A robust and scalable implementation of the Parks-McClellan algorithm for designing FIR filters. *ACM Transactions on Mathematical Software (TOMS)*, 43(1), 7.

[45] Flynn, M. (1970). On division by functional iteration. *Computers, IEEE Transactions on*, C-19(8), 702–706.

[46] Gaide, B. (2014). Methods of pipelining a data path in an integrated circuit. US Patent 8,893,071.

[47] Gal, S. (1986). Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations* (pp. 1–16). Springer.

[48] Gal, S. (1991). An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17.

[49] Ganusov, I., Fraisse, H., Ng, A. N., Possignolo, R. T., & Das, S. (2016). Automated extra pipeline analysis of applications mapped to Xilinx UltraScale+ FPGAs. In *Field Programmable Logic and Applications*.

[50] Gutierrez, R., Torres, V., & Valls, J. (2010). FPGA-implementation of atan(Y/X) based on logarithmic transformation and LUT-based techniques. *Journal of Systems Architecture*, 56.

[51] Gutierrez, R. & Valls, J. (2008). Low-Power FPGA-Implementation of atan(Y/X) Using Look-Up Table Methods for Communication Applications. *Journal of Signal Processing Systems*, 56.

[52] Hauck, S. & DeHon, A. (2007). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[53] Hilaire, T. & Lopez, B. (2013). Reliable implementation of linear filters with fixed-point arithmetic. In *SiPS, Workshop on Signal Processing Systems*: IEEE.

[54] Hsiao, S.-F., Wu, P.-H., Wen, C.-S., & Meher, P. (2015). Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 62(5), 466–470.

[55] Hwang, D. & Willson, A. (2003). A 400-MHz processor for the conversion of rectangular to polar coordinates in 0.25-μm cmos. *IEEE Journal of Solid-State Circuits*, 38.

[56] Hwang, J. & Ballagh, J. (2002). Building custom FIR filters using system generator. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, FPL '02 (pp. 1101–1104).

[57] IEEE (2006). Ieee standard for information technology– local and metropolitan area networks– specific requirements– part 15.4: Wireless medium access control (mac) and physical layer (phy) specifications for low rate wireless personal area networks (wpans). *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, (pp. 1–320).

[58] Ishikawa, M., Edahiro, M., Yoshimura, T., Miyazaki, T., Aikoh, S. I., Nishitani, T., Mitsuhashi, K., & Furuichi, M. (1990). Automatic layout synthesis for FIR filters using a silicon compiler. In *Circuits and Systems, 1990., IEEE International Symposium on* (pp. 2588–2591 vol.4).

[59] Ito, M., Takagi, N., & Yajima, S. (1997). Efficient initial approximation for multiplicative division and square root by a multiplication with operand modification. *Computers, IEEE Transactions on*, 46(4), 495–498.

[60] Jaberipur, G., Parhami, B., & Ghodsi, M. (2006). An efficient universal addition scheme for all hybrid-redundant representations with weighted bit-set encoding. *Journal of VLSI signal processing systems for signal, image and video technology*, 42(2), 149–158.

[61] Jain, R., Yang, P. T., & Yoshino, T. (1991). FIRGEN: a computer-aided design system for high performance FIR filter integrated circuits. *IEEE Transactions on Signal Processing*, 39(7), 1655–1668.

[62] Jha, P. K. & Dutt, N. D. (1992). *A Fast Area-Delay Estimation Technique for RTL Component Generators*. Technical report, Information and Computer Science, University of California, Irvine.

[63] Kailath, T. (1980). *Linear Systems*. Prentice-Hall.

[64] Kaiser, J. (1974). Nonrecursive digital filter design using the I_0-sinh window function. In *Proc. 1974 IEEE International Symposium on Circuits & Systems* (pp. 20–23).

[65] Karatsuba, A. & Ofman, Y. (1963). Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7 (pp. 595).

[66] Karkooti, M., Cavallaro, J. R., & Dick, C. (2005). FPGA implementation of matrix inversion using QRD-RLS algorithm. In *Asilomar Conference on Signals, Systems, and Computers*.

[67] Kilts, S. (2007). *Advanced FPGA Design: Architecture, Implementation, and Optimization*. John Wiley & Sons.

[68] Koomey, J., Berard, S., Sanchez, M., & Wong, H. (2011). Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3), 46–54.

[69] Kumm, M., Möller, K., & Zipf, P. (2013). Dynamically reconfigurable fir filter architectures with fast reconfiguration. In *8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)* (pp. 1–8).

[70] Kumm, M. & Zipf, P. (2014a). Efficient high speed compression trees on xilinx FPGAs. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)* (pp. 171–182).

[71] Kumm, M. & Zipf, P. (2014b). Pipelined compressor tree optimization using integer linear programming. In *24th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1–8).

[72] Leiserson, C. E. & Saxe, J. B. (1991). Retiming synchronous circuitry. *Algorithmica*, 6(1), 5 – 35.

[73] Lenstra, A. K., Lenstra, Jr., H. W., & Lovász, L. (1982). Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4), 515–534.

[74] Levin, L. A. (1984). Problems, complete in “average” instance. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84 (pp. 465–). New York, NY, USA: ACM.

[75] Levin, L. A. (1986). Average case complete problems. *SIAM Journal on Computing*, 15(1), 285–286.

[76] Lopez, B., Hilaire, T., & Didier, L.-S. (2014). Formatting bits to better implement signal processing algorithms. In *Pervasive and Embedded Computing and Communication Systems*: ScitePress.

[77] Lotze, J., Fahmy, S. A., Noguera, J., Doyle, L., & Esser, R. (2008). An FPGA-based cognitive radio framework. *IET Conference Proceedings*, (pp. 138–143(5)).

[78] Low, J. & Jong, C. C. (2013). A memory-efficient tables-and-additions method for accurate computation of elementary functions. *Computers, IEEE Transactions on*, 62(5), 858–872.

[79] Marlow, R., Dobson, C., & Athanas, P. (2014). An enhanced and embedded gnu radio flow. In *24th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1–4).

[80] Matsunaga, T., Kimura, S., & Matsunaga, Y. (2013). An exact approach for gpc-based compressor tree synthesis. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 96(12), 2553–2560.

[81] McCanny, J. V., Hu, Y., & Yan, M. (1994). Automated design of DSP array processor chips. In *Application Specific Array Processors, 1994. Proceedings. International Conference on* (pp. 33–44).

[82] McClellan, J. H., Parks, T. W., & Rabiner, L. (1973). A computer program for designing optimum FIR linear phase digital filters. *IEEE Transactions on Audio and Electroacoustics*, 21(6), 506–526.

[83] Mehendale, M., D.Sherlekar, S., & Venkatesh, G. (1995). Synthesis of multiplier-less FIR filters with minimum number of additions. In *IEEE/ACM International Conference on Computer-Aided Design* (pp. 668–671). San Jose, CA USA.

[84] Meher, P. K., Valls, J., Juang, T.-B., Sridharan, K., & Maharatna, K. (2009). 50 years of CORDIC: Algorithms, architectures, and applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(9), 1893–1907.

[85] Mencer, O., Semeria, L., Morf, M., & Delosme, J.-M. (2000). Application of reconfigurable CORDIC architectures. *Journal of VLSI signal processing systems for signal, image and video technology*, 24(2-3), 211–221.

[86] Milder, P., Ahmad, M., Hoe, J. C., & Püschel, M. (2006). Fast and accurate resource estimation of automatically generated custom DFT IP cores. *Proceedings of the internation symposium on Field programmable gate arrays - FPGA'06*, (pp. 211).

[87] Mohanakrishnan, S. & Evans, J. B. (1995). Automatic implementation of FIR filters on field programmable gate arrays. *IEEE Signal Processing Letters*, 2(3), 51–53.

[88] Montgomery, P. L. (2005). Five, six, and seven-term karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3), 362–369.

[89] Moore, C. & Robson, J. M. (2001). Hard tiling problems with simple tiles. *Discrete & Computational Geometry*, 26(4), 573–590.

[90] Moore, G. E. (2006). Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(5), 33–35.

[91] Muller, J.-M. (1999). A few results on table-based methods. *Reliable Computing*, 5(3), 279–288.

[92] Muller, J.-M. (2006). *Elementary functions*. Springer.

[93] Narayan, D. A. & Schwenk, A. J. (2002). Tiling large rectangles. *Mathematics magazine*, 75(5), 372–380.

[94] Nayak, A., Haldar, M., Choudhary, A., & Banerjee, P. (2002). Accurate area and delay estimators for FPGAs. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition* (pp. 862–869).

[95] Nguyen, H. D., Pasca, B., & Preußer, T. B. (2011). FPGA-specific arithmetic optimizations of short-latency adders. In *21st International Conference on Field Programmable Logic and Applications* (pp. 232–237).

[96] Oklobdzija, V. G., Villeger, D., & Liu, S. S. (1996). A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Transactions on Computers*, 45(3), 294–306.

[97] Oppenheim, A. V. & Schafer, R. W. (2010). *Discrete-Time Signal Processing*. Prentice Hall.

[98] Parandeh-Afshar, H., Neogy, A., Brisk, P., & Ienne, P. (2011). Compressor tree synthesis on commercial high-performance FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4), 39.

[99] Parks, T. W. & McClellan, J. H. (1972). Chebyshev Approximation for Nonrecursive Digital Filters with Linear Phase. *IEEE Transactions on Circuit Theory*, 19(2), 189–194.

[100] Pasca, B. (2012). Correctly rounded floating-point division for dsp-enabled FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)* (pp. 249–254).

[101] Perrelet, D., Villanueva, A., Sundal, M., Brischetto, Y., Oberson, D., & Damerau, H. (2015). *White-Rabbit Based Revolution Frequency Program for the Longitudinal Beam Control of the CERN PS*. Technical report, CERN.

[102] Potkonjak, M., Srivastava, M., & Chandrakasan, A. (1994). Efficient substitution of multiple constant multiplications by shifts and additions using iterative pairwise matching. In *ACM IEEE Design Automation Conference* (pp. 189–194). San Diego, CA USA.

[103] Rabinowitz, P. (1961). Multiple-precision division. *Commun. ACM*, 4(2), 98–.

[104] Rajan, S., Wang, S., & Inkol, R. (2006). Efficient Approximations for the Four-Quadrant Arctangent Function. In *Canadian Conference on Electrical and Computer Engineering*: IEEE.

[105] Remes, E. (1934). Sur le calcul effectif des polynomes d'approximation de Tchebichef. *Comptes rendus hebdomadaires des séances de l'Académie des Sciences*, 199(199), 337–340.

[106] Saber, M., Jitsumatsu, Y., & Kohda, T. (2009). A low-power implementation of arctangent function for communication applications using FPGA. In *Signal Design and its Applications in Communications*: IEEE.

[107] Sait, S. M. & Youssef, H. (1994). *VISI Physical Design Automation: Theory and Practice*. New York, NY, USA: McGraw-Hill, Inc.

[108] Schlessman, J., Chen, C.-Y., Wolf, W., Ozer, B., Fujino, K., & Itoh, K. (2006). Hardware/-software co-design of an FPGA-based embedded tracking system. In *2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06)* (pp. 123–123).

[109] Schulte, M. & Stine, J. (1997). Symmetric bipartite tables for accurate function approximation. In *ARITH'13* (pp. 175–183).

[110] Schumacher, P. (2008). Fast and accurate resource estimation of RTL-based designs targeting FPGAs. *Field Programmable Logic and Applications, International Conference on*.

[111] Shen, J. & Strang, G. (1999). The asymptotics of optimal (equiripple) filters. *IEEE Transactions on Signal Processing*, 47(4), 1087–1098.

[112] Smyk, R. (2013). FIReWORK: FIR filters hardware structures auto-generator. *Journal of Applied Computer Science*, 21(1), 135–149.

[113] Stanislaus, J. L. V. M. & Mohsenin, T. (2013). Low-complexity FPGA implementation of compressive sensing reconstruction. In *Computing, Networking and Communications (ICNC), 2013 International Conference on* (pp. 671–675).

[114] Stelling, P. F., Martel, C. U., Oklobdzija, V. G., & Ravi, R. (1998). Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, 47(3), 273–285.

[115] Stine, J. & Schulte, M. (1999). The symmetric table addition method for accurate function approximation. *VLSI*, 21, 167–177.

[116] Story, S. & Tang, P. T. P. (1999). New algorithms for improved transcendental functions on IA-64. In *14th IEEE Symposium on Computer Arithmetic*: IEEE Comput. Soc.

[117] Swartzlander Jr, E. E. (1980). Merged arithmetic. *IEEE Transactions on Computers*, 29(10), 946–950.

[118] Sweeney, P. E. & Paternoster, E. R. (1992). Cutting and packing problems: a categorized, application-orientated research bibliography. *Journal of the Operational Research Society*, 43(7), 691–706.

[119] Takagi, N. (1997). Generating a power of an operand by a table look-up and a multiplication. In *ARITH'13* (pp. 126–131).

[120] Timmermann, D., Hahn, H., & Hosticka, B. (1989). Modified CORDIC algorithm with reduced iterations. *Electronics Letters*, 15(25), 950–951.

[121] Valls, J., Kuhlmann, M., & Parhi, K. K. (2002). Evaluation of CORDIC algorithms for FPGA design. *Journal of VLSI signal processing systems for signal, image and video technology*, 32(3), 207–222.

[122] van Emde Boas, P. & Savelsbergh, M. (1984). Bounded tiling, an alternative to satisfiability. In *Proceedings of the 2nd Frege Memorial Conference* (pp. 401–407).

[123] Venkataramani, G. & Gu, Y. (2014). System-level retiming and pipelining. In *Field-Programmable Custom Computing Machines (FCCM), International Symposium on* (pp. 80–87).

[124] Verma, A. K., Brisk, P., & Ienne, P. (2008). Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10), 1761–1774.

[125] Volder, J. E. (1959). The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8(3), 330–334.

[126] Volkova, A., Hilaire, T., & Lauter, C. (2015a). Reliable evaluation of the Worst-Case Peak Gain matrix in multiple precision. In *IEEE Symposium on Computer Arithmetic*.

[127] Volkova, A., Hilaire, T., & Lauter, C. Q. (2015b). Determining fixed-point formats for a digital filter implementation using the worst-case peak-gain measure. In *Asilomar Conference on Signals, Systems and Computers*.

[128] Voronenko, Y. & Püschel, M. (2007). Multiplierless multiple constant multiplication. *ACM Trans. Algorithms*, 3(2).

[129] Wallace, C. S. (1964). A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1), 14–17.

[130] Walther, J. S. (1971). A unified algorithm for elementary functions. In *Proceedings of the May 18-20, 1971 Spring Joint Computer Conference*, AFIPS '71 (Spring) (pp. 379–385). New York, NY, USA: ACM.

[131] Wang, D., Ercegovac, M., & Xiao, Y. (2014). Complex function approximation using two-dimensional interpolation. *IEEE Transactions on Computers*, 63(12), 2948–2960.

[132] White, S. (1989). Applications of distributed arithmetic to digital signal processing: a tutorial review. *IEEE ASSP Magazine*, 6(3), 4–19.

[133] Willers, F. & Beyer, R. (1948). *Practical analysis: graphical and numerical methods*. Dover Books on Science. Dover Publications.

[134] Wirthlin, M. J. (2004). Constant coefficient multiplication using look-up tables. *Journal of VLSI signal processing systems for signal, image and video technology*, 36(1), 7–15.

[135] Wong, W. & Goto, E. (1995). Fast evaluation of the elementary functions in single precision. *Computers, IEEE Transactions on*, 44(3), 453–457.

[136] Yang, P. T., Jain, R., Yoshino, T., Gass, W., & Shah, A. (1990). A functional silicon compiler for high speed FIR digital filters. In *Acoustics, Speech, and Signal Processing, 1990. ICASSP-90., 1990 International Conference on* (pp. 1329–1332).

UNIVERSITÉ DE LYON

INSA

FOLIO ADMINISTRATIF

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

NOM : IŞTOAN

DATE de SOUTENANCE : 06/04/2017

Prénoms : Matei Valentin

TITRE : High-performance Coarse Operators for FPGA-based Computing

NATURE : Doctorat

Numéro d'ordre : AAAALYSEIXXXX

Ecole doctorale : Informatique Mathématique

Spécialité : Informatique

RESUME : Field-Programmable Gate Arrays (FPGAs) have been shown to sometimes outperform mainstream microprocessors. The circuit paradigm enables efficient application-specific parallel computations. FPGAs also enable arithmetic efficiency: a bit is only computed if it is useful to the final result. To achieve this, FPGA arithmetic shouldn't be limited to basic arithmetic operations offered by microprocessors.

This thesis studies the implementation of coarser operations on FPGAs, in three main directions:

New FPGA-specific approaches for evaluating the sine, cosine and the arctangent have been developed. Each function is tuned for its context and is as versatile and flexible as possible. Arithmetic efficiency requires error analysis and parameter tuning, and a fine understanding of the algorithms used.

Digital filters are an important family of coarse operators resembling elementary functions: they can be specified at a high level as a transfer function with constraints on the signal/noise ratio, and then be implemented as an arithmetic datapath based on additions and multiplications. The main result is a method which transforms a high-level specification into a filter in an automated way. The first step is building an efficient method for computing sums of products by constants. Based on this, FIR and IIR filter generators are constructed.

For arithmetic operators to achieve maximum performance, context-specific pipelining is required. Even if the designer's knowledge is of great help when building and pipelining an arithmetic datapath, this remains complex and error-prone. A user-directed, automated method for pipelining has been developed.

This thesis provides a generator of high-quality, ready-made operators for coarse computing cores, which brings FPGA-based computing a step closer to mainstream adoption. The cores are part of an open-ended generator, where functions are described as high-level objects such as mathematical expressions.

MOTS-CLÉS : FPGA, computer arithmetic, arithmetic operator, digital filter

Laboratoire de recherche : CITI lab

Directeur de thèse: DE DINECHIN Florent

Président de jury :

Composition du jury :

IENNE, Paolo Professeur des Universités, EPFL, Lausanne	Rapporteur
CHOTIN-AVOT, Roselyne Maître de Conférences HDR, UPMC	Rapporteur
THOMAS, David Senior Lecturer Imperial College London, London	Examinateur
PETROT, Frédéric Professeur des Universités, ENSIMAG, Saint-Martin d'Heres	Président
SENTIEYS, Olivier Professeur des Universités, ENSSAT, Lannion	Examinateur
DE DINECHIN, Florent Professeur des Universités INSA-Lyon	Directeur de thèse