High-performance floating-point computing on reconfigurable circuits

Bogdan Mihai Pasca

N° d'ordre : 656
N° attribué par la bibliothèque : 2011ENSL0656

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée et soutenue publiquement le 21 Septembre 2011 par

Bogdan PASCA

pour l'obtention du grade de

Docteur de l'École Normale Supérieure de Lyon

spécialité : Informatique

au titre de l'École Doctorale de Mathématiques et d'Informatique Fondamentale de Lyon

High-performance floating-point computing on reconfigurable circuits

Directeur de thèse : Florent DE DINECHIN

Après avis de : Paolo IENNE
Olivier SENTIEYS

Devant la commission d'examen formée de :

Octavian CRET	Membre
Florent DE DINECHIN	Membre
Paul FEAUTRIER	Membre
Paolo IENNE	Membre/Rapporteur
Martin LANGHAMMER	Membre
Olivier SENTIEYS	Membre/Rapporteur

Acknowledgements

First of all, I want to thank my family for their priceless and unconditioned support all throughout this thesis. They made me what I am today and I will be forever grateful.

Next, I want to kindly thank my girlfriend Mioara for her love and support which helped me to seamlessly overcome all encountered challenges. Thank you for being supportive and encouraging during the late hours we spent at the office on our endless deadlines.

I also want to thank our Romanian community from ENS, for all the brilliant time spent together. The endless polemics on various subjects were both cultivating and helped me significantly improve my argumentation capabilities. The long and challenging bike rides, gym practice, climbing, running ... made me always push myself one step further, which I also tried to apply to research.

I especially want to thank my supervisor Florent for believing in my potential and efficiently using my skills throughout the thesis time. His guidance was excellent, always coming up with interesting subjects to work on, but also providing me with sufficient freedom to tackle my own research subjects.

I also want to thank my thesis reviewers, whose constructive comments helped me further improve this manuscript, and the jury members for their pertinent and challenging questions during the defense.

A big thanks to CompSys members, Alexandru and Christophe, for the interesting discussions carried around the long coffee/tea brakes. These discussions did not only widen my research interests, but also resulted in one research article during the last year of this thesis.

Last but not least, I want to thank the Arenaire team members, for their kindness and for always having the door open for me. I also like to thank our team assistants, Severine and Damien for all their help in simplifying the sometimes overwhelming paperworks.

Many thanks to my friends and everyone else which I haven't mentioned here for their help and support.

Contents

List of Figures

List of Tables

Acronyms

FPGA Field Programmable Gate Array
GPU Graphical Procesing Unit
HPC High-Performance Computing
IC Integrated Circuit
ASIC Application-Specific Integrated Circuit
CUDA Compute Unified Device Architecture
HDL Harware Description Language
PAL Programmable Array of Logic
CPLD Complex Programmable Logic Device
LE logic element
LUT look-up table
CLB Configurable Logic Block
LAB Logic Array Block
MLAB Memory Logic Array Block (LAB)
ALM Adaptative Logic Module
ALUT adaptative look-up table
RAM Random-Access Memory
SRAM Static Random-Access Memory (RAM)
EDIF Electronic Digital Interchange Format
XNF Xilinx Netlist Format
FSM Finite State Machine
FP floating-point
SP single precision
DP double precision
QP quadruple precision
FPU Floating-Point Unit
CR correct rounding
FR faithful rounding
HLS High-Level Synthesis
DSP Digital Signal Processing
FFT Fast Fourier Transform

FIR Finite Impulse Response
PE Processing Element
RCA Ripple-Carry Adder
FA Full-Adder
MSB Most Significant Bit

Publications

1. Florent de Dinechin, Jean-Michel Muller, Bogdan Pasca, and Alexandru Plesco. An FPGA architecture for solving the Table Maker's Dilemma. In *International Conference on Application-specific Systems, Architectures and Processors*, 2011. *Best Paper Award*
2. Hong Diep Nguyen, Bogdan Pasca, and Thomas B. Preußer. FPGA-specific arithmetic optimizations of short-latency adders. In *International Conference on Field Programmable Logic and Applications*. IEEE, 2011.
3. Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design and Test*, 2011.
4. Christophe Alias, Bogdan Pasca, and Alexandru Plesco. Automatic generation of FPGA-specific pipelined accelerators. In *The 7th International Symposium on Applied Reconfigurable Computing*, 2011.
5. Florent de Dinechin and Bogdan Pasca. Floating-point exponential functions for DSP-enabled FPGAs. In *IEEE International Conference on Field-Programmable Technology*. IEEE, 2010.
6. Sebastian Banescu, Florent de Dinechin, Bogdan Pasca, and Radu Tudoran. Multipliers for floating-point double precision and beyond on FPGAs. In *International Workshop on Higly-Efficient Accelerators and Reconfigurable Technologies (HEART)*. ACM, 2010.
7. Florent de Dinechin, Mioara Joldes, and Bogdan Pasca. Automatic generation of polynomial-based hardware architectures for function evaluation. In *International Conference on Application-specific Systems, Architectures and Processors*, 2010.
8. Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. Multiplicative square root algorithms for FPGAs. In *International Conference on Field Programmable Logic and Applications*. 2010.
9. Florent de Dinechin, Hong Diep Nguyen, and Bogdan Pasca. Pipelined FPGA adders. In *International Conference on Field Programmable Logic and Applications*. 2010.
10. Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. Racines carrées multiplicatives sur FPGA. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, 2009.
11. Florent de Dinechin and Bogdan Pasca. Large multipliers with fewer DSP blocks. In *International Conference on Field Programmable Logic and Applications*. 2009.
12. Florent De Dinechin, Cristian Klein, and Bogdan Pasca. Generating high-performance custom floating-point pipelines. In *International Conference on Field Programmable Logic and Applications*, 2009.
13. Florent de Dinechin, Bogdan Pasca, Octavian Creţ, and Radu Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *IEEE International Conference on Field-Programmable Technology*, 2008.

1 Chapter 1
Introduction

The classical version of Moore's Law predicts that the capacity of Integrated Circuits (ICs) doubles every 18 months. Microprocessor manufacturers followed this law by reducing the operating voltages and using smaller and faster transistors. Frequency scaling got to the point that circuits emitted too much heat to be reasonably dissipated – the so called power wall. This lead the main microprocessor manufacturer, Intel, to publicly announce in 2004 that it would dedicate all it future design efforts to multi-core environments. Nowadays, Intel offers a 8-core version of the high-end Xeon processor (V8), while Opteron from AMD is provided in a 12-core version, both at 45nm manufacturing process.

Just doubling the number of cores in a die doesn't guarantee a speedup of two over the initial microprocessor for a given application. Indeed, Amdahl's law [34] suggests that the maximum expected overall improvement of a system using N processors is highly influenced by the amount of sequential execution of the program, but also by the degree of parallelism of the parallel sections. Most of the existing software, developed during the single-core era is essentially sequential and therefore does't benefit from any improvement on a multicore system. One idea, dating back from the 1960s, is to write compilers that would automatically parallelize these sequential programs. The success of these approaches seems to be inversely proportional to the number of targeted cores. One reason for this insuccess is that the sequential solution these tools start with already looses some of the "parallel semantics" of the problem to be solved. Consequently, making efficient use of multiple cores requires recovering some of this lost parallelism. This requires recoding parts of the application using the *thread programming* model or using one of the well known APIs supporting process intercommunication: MPI, PVM or OpenMP. Another reason for the poor performance of these parallelized programs is that in a multicore system inter-process communication, usually resolved by shared-memory techniques, is very costly. In any case, the success of this approach will depend on the data-level parallelism of the initial application.

One success story is computer graphics. Graphics processing is an application domain having massively parallel computational kernels: entire animation scenes and also parts of each frame can be processed in parallel. Traditionally, Graphical Procesing Units (GPUs) consisted of numerous but rather simple Processing Elements (PEs) capable of processing the numerous graphics-related tasks in a flow-like manner. In 2001, with the introduction of first programmable GPU (the NV20 series) programmers could execute custom visual-effects programs using the Shader Language 1.1. In 2007 nVIDIA formalized the GPU's computing capabilities under the name of Compute Unified Device Architecture (CUDA): the parallel computing architecture present in nVIDIA GPUs. General-purpose computations can be expressed using *C for CUDA*, a C subset with nVIDIA extensions. As the PEs of modern GPUs support some of the basic floating-point operators, it is tempting to use them to perform massively parallel scientific computations.

Nevertheless, acceleration degree is very application-dependent (applications should have high data-level parallelism and main computation task should be supported in silicon by the PEs) and obtaining good accelerations requires a significant amount of code refactoring.

Field Programmable Gate Arrays (FPGAs) have also benefited from the advances in circuit integration. With increased capacities, FPGAs moved from being used as glue-logic to prototyping Application-Specific Integrated Circuits (ASICs), and recently to ASIC replacements and application accelerators [111, 65, 57]. If in the past, performance- and power-demanding systems were usually built using ASICs, their use today is being limited by their prohibitive manufacturing price. Moving down in the manufacturing process from 130nm to 90nm has doubled ASIC mask prices and requires millions more in engineering. This restricts the viable use of ASICs to medium and hi-volume markets (more than 100K chips sold). On the other hand, older technology ASICs (130nm and higher) are neither price- nor performance-efficient when compared to the current 45nm FPGAs.

FPGAs have recently been considered as accelerators for a wide spectrum of applications with various computational needs: data mining [39] and genome sequencing [126], logical testing and numerical aggregation operations, medical imaging [57], scientific visualization, physics simulations [114] computational chemistry [88], financial analytics [117, 161]. All these applications involve operations like coordinate mapping, mathematical transformations, filtering etc. and involve massive low and medium-grain parallelism. The architecture of Modern FPGAs have been augmented with "ASIC-like" features: fast-carry chains for enhanced binary addition performance, multiplier blocks for better mapping of digital signal processing applications and arithmetic functions, embedded memories for increasing on-chip throughput etc. These new added features make FPGAs very suitable for accelerating these applications. On one hand, they allow the ad-hoc implementation of the exotic arithmetic operators needed by these computations and not supported in hardware by processors or GPUs. These operators can be deeply pipelined and function at FPGA nominal frequency, yielding significant speedups over their software implementations counterparts. On the other hand, coarser computational data-paths, possibly using these exotic operator instances, may be instantiated. Rather then communicating by means of shared memories wasting computation cycles and power, the data-path components may be simply wired together using the FPGA's reconfigurable interconnect network, allowing the data produced to be directly consumed, thus maximizing efficiency. This thesis studies the FPGA implementation of such arithmetic operators and also the design of coarser arithmetic data-paths.

It's possible to use standard Harware Description Languages (HDLs) (VHDL or Verilog) to manually design arithmetic operators. Handcrafting basic standard operators (not necessarily optimal) is possible using VHDL, but designing exotic operators is a task impossible to perform using VHDL alone. For instance, implementing operators such $\sqrt{x}$ or some elementary functions ($e^x, \log x \ldots$) using polynomial approximations requires pre-computing tables of values which are function- and precision-specific, but also depend approximating polynomial degree. External tools are usually used to pre-compute these values.

Operator generators can naturally alleviate the limitations of VHDL. The design space exploration can be done using a high-level programming language (C++, Java, ...) and operator specific VHDL description can be generated. To our knowledge Xilinx pioneered this approach with Logicore. Nowadays, main FPGA manufacturers (Altera and Xilinx) ship operator generators with their design tools, allowing far more parameters for each operator than one could get using a parametrized operator library.

In their simplest form generators can simply perform some design space exploration and write VHDL code files. Therefore, generators are at least as expressive as VHDL. However, the generator's framework could also allow reusing already designed operators, help signal declarations, possibly facilitate pipelining but also provide specialized assistance: help manage arithmetic ar-

gument reduction and other. Due to their proprietary nature, it is unclear how many of these features are provided by the generator frameworks of Altera and Xilinx. Operator generators can also be used for on-the-fly generation of arithmetic components by High-Level Synthesis (HLS) tools targeting FPGAs. They are more flexible and easier to maintain than VHDL arithmetic operator libraries.

FloPoCo [1] (**Flo**ating-**Po**int **Co**res, but not only) is an open-source C++ framework for the generation of arithmetic datapaths. It provides a command-line interface that inputs operator specifications, and outputs synthesizable VHDL. The main goal of this thesis was to develop and refine the FloPoCo generator framework for the class of arithmetic operators. Consequently, one of the main contributions of this thesis is the framework itself. It assists in designing and testing arithmetic operators which can be flexible in input/output precision, may be easily retargeted to other FPGA devices and allow a user-defined trade-off between operating-frequency and occupied resources. All this significantly shortens the arithmetic operator design cycle therefore enhancing productivity.

The second main contribution concerns the library of flexible arithmetic operators designed using the FloPoCo framework. These operators include basic fixed-point and floating-point operators, an automatic generator of fixed-point function implementations based on polynomial approximation, operators for the floating-point square-root and exponential functions and also one meta-operator allowing the fast assembly of available floating-point operators. The work done for describing these operators has validated the framework and motivated its continuous development. Often, the framework enhancements for one operator have improved the performance of other existing operators.

The third contribution of this thesis is the efficient use of the pipelined arithmetic operators generated by FloPoCo in an application context. FloPoCo optimizes the architecture of the generated operators for user defined application constraints, which causes the operator's latency to be dependent on these constraints. Our main objective was to optimize the execution scheduling of codes using these operators which we successfully achieved for applications described using perfectly nested loops with uniform data dependencies.

The final contribution of this thesis is the validation of the FloPoCo framework for implementing a complete application. The framework is put to the test for implementing a flexible, parametric description of an FPGA-specific architecture for solving the "Table Maker's Dilemma". The final architecture consists of several operator layers, all having multiple flexible parameters, and is designed to fill-up the largest FPGAs available. Thanks to the FloPoCo framework, for an extensive set of parameters including the deployment FPGA, an architecture composed of thousands of lines of code is generated in seconds. It allows exploring a large set of possible implementations to select the one which best fits the target FPGA.

We strongly believe that the FPGA implementation of arithmetic data-paths should make the best use of the FPGA's flexibility and available resources. Our experiences in designing arithmetic operators, both before and during the development of FloPoCo, have helped establish and refine a framework which offers, in our opinion, just the right abstraction level for an FPGA-specific arithmetic circuit description.

The rest of this thesis is organized as follows: after briefly presenting in Chapter 2 an overview of the modern FPGA architecture and particularly, the features relevant for arithmetic operator design, we will give in Chapter 3 a brief introduction to floating-point arithmetic and present the relevant works regarding the implementation of floating-point operators in FPGAs. Chapter 4 will then show the various gains of using FloPoCo for designing custom arithmetic datapaths and present in detail the framework's features. Next, Chapter 5 will present the various binary-adder architectures present in FloPoCo and prove that the optimal architecture for a given

1. http://flopoco.gforge.inria.fr/

scenario can be chosen based on analytically deduced resource estimation formulas. Chapter 6 will then give an insight on how to build binary multipliers and squarers using fewer multiplier resources. Next, we will present in Chapter 7 a generic fixed-point function evaluation implementation based on polynomial approximation which is both scalable and more performant than other available implementations. This function evaluator is used as the main building block of the floating-point square-root operator presented in Chapter 8 and also as a key component in the implementation of the floating-point exponential function presented in Chapter 9. Next, Chapter 10 presents the FloPoCo implementation of a FPGA-specific floating-point accumulator and of a dot-product operator based on this accumulator. In Chapter 11 we focus on efficiently scheduling the computations of computing kernels described by specific loop nests on pipelined-operators, in the context of using FloPoCo as a back-end for semi-automatic HLS. Finally, in Chapter 12 we show that FloPoCo can effectively be used for describing the architecture of a complete computing application for solving the Table Maker's Dilemma.

CHAPTER 2
Field Programmable Gate Arrays

FPGAs are memory-based integrated circuits whose functionality can be programmed after manufacturing. They were commercially introduced in 1985 by Xilinx [11] with the XC2064 product, and are natural descendants of Complex Programmable Logic Devices (CPLDs).

Unlike CPLDs which are organized as small arrays of PALs, FPGAs have a much finer granularity. An FPGA is structured as large bidimensional array (>100K) of logic elements (LEs). The LEs contain small programmable memories (most FPGAs are SRAM-based) and are interconnected by a configurable wiring network. One possible layout of an FPGA architecture which matches this description is presented in Figure 2.1. The reconfigurability of both LE and the interconnect network allows the implementation of any logical circuit provided it fits in the FPGA.

However, reconfigurability comes at a price. Despite an equivalent technological processes, the typical frequency of FPGA designs is in the low hundreds of MHz, whereas the microprocessor counterpart runs at several GHz. A one-to-one comparison between an arithmetic operator supported in silicon in a microprocessor (FP addition for example) and its FPGA counterpart will roughly yield a speedup of 10 in favor of the microprocessor. FPGAs may recover this thanks to the massive parallelism and fine-grain flexibility.

We review next architectural features of FPGAs introduced by the two market leaders Altera and Xilinx between 2004 (with the introduction of the Stratix-II and Virtex-4 devices) up to 2011. We focus on the elements which concern the design of arithmetic operators and we ignore features like: transceivers and embedded processors whose documentation takes more than two thirds of the device handbooks.

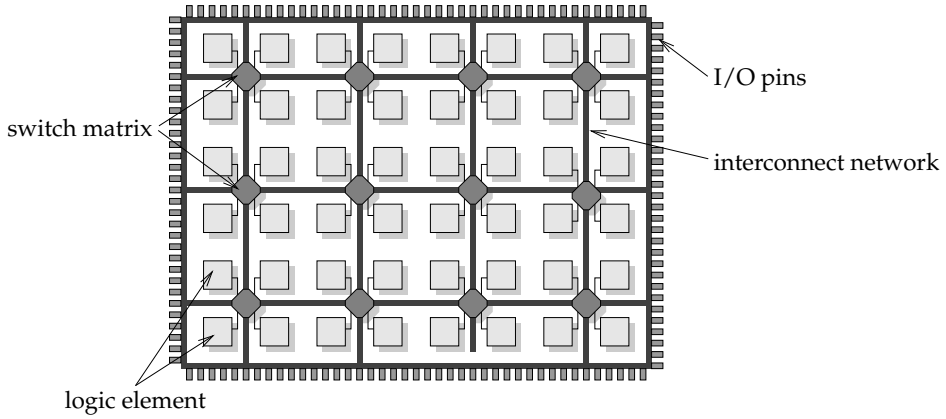


Figure 2.1 *Very simplified view of a generic FPGA layout*

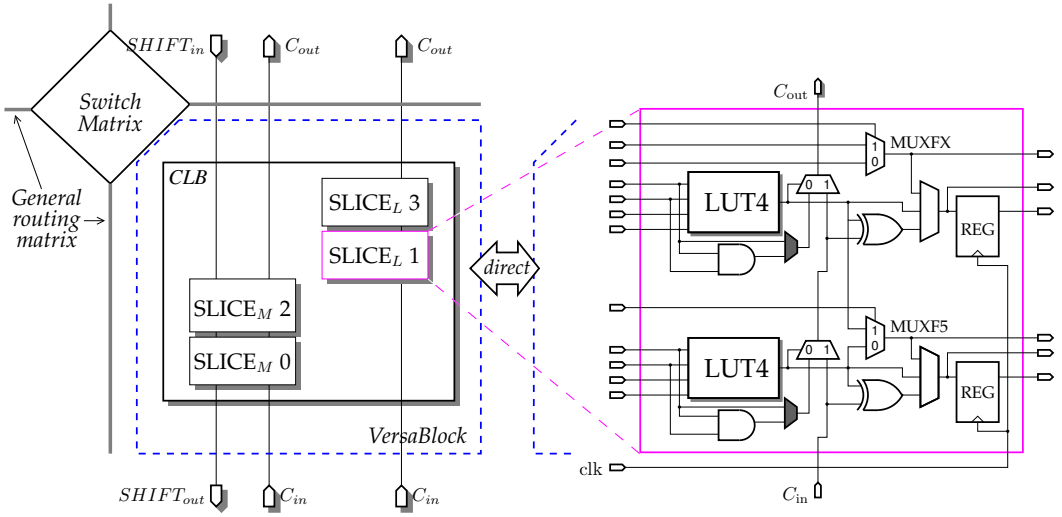


Figure 2.2 *Left: CLB composition and interconnect in Virtex-4 devices Right: Detailed view of a Virtex-4 Slice*

2.1 Architecture

The architecture of modern FPGAs is composed of logic elements (implemented as look-up tables), embedded memories, embedded multipliers and several other components as well. Some of these features like look-up tables (LUTs) and small multipliers (in the case of Altera devices) are regrouped into clusters. The advantage of clustering these components is that it reduces the routing pressure. Within a cluster the elements can be fully interconnected while keeping a relatively low number of wires for connecting the cluster to the general routing network. Moreover, some direct connections to neighboring clusters can exist, allowing clusters to interact while bypassing the slower general routing network.

In the case of Xilinx devices, the clusters of LUTs are are called Configurable Logic Blocks (CLBs) whereas in the case of Altera these are called Logic Array Blocks (LABs). The granularity of these clusters is both manufacturer and FPGA-family dependent. It greatly impacts routing and therefore the performance of the FPGA. Modern FPGAs from Xilinx use CLBs of eight LUTs (again, the LUTs size and features vary among device families) whereas Altera uses a slightly larger LAB, with 16 LUTs for Stratix-II and 20 LUTs for Stratix-III/-IV.

Both manufacturers use a second hierarchical regrouping of elements within a cluster: Slices for Xilinx and Adaptative Logic Modules (ALMs) for Altera. The elements of this level regroup together two LUTs together with several other enhanced features which will be reviewed next.

2.1.1 Logic elements

Xilinx

The CLB structure varies between FPGA generations. In the case of Virtex-4 devices [18] the CLB is made out of four slices grouped in pairs. The pairs are organized in two columns, as presented in Figure 2.2. The slices in the right column are called SLICE$_L$s (the L comes from *logic*) and those in the left column are SLICE$_M$ (M comes from *memory*). SLICE$_M$s provide the same functions as SLICE$_L$s but additionally feature a superset of memory-related functions. A simplified overview of the layout of a SLICE$_L$ is presented on the right of Figure 2.2.

There are two function generators in each Virtex4 slice, denoted by LUT4 in Figure 2.2. Each function generator is implemented as a programmable 4-input LUT totaling 16-bits of memory.

This allows the implementation of any 4-input boolean function.

Moreover, slices also contain multiplexers (denoted by MUXF5 and MUXFX in Figure 2.2). They can play two roles: (1) in combination with fast local routing resources they allow implementing functions of more than four variables and (2) can implement multiplexers of up to 16:1 in one CLB and up to 32:1 in two neighboring CLBs.

Slices also provide enhanced performance of binary adder and subtracter implementation using the RCA scheme. In this configuration, each half-slice assumes the role of a full-adder. Dedicated fast carry lines traversing vertically the CLBs allow the carry-bit to ripple faster than using the general routing network. Figure 2.3 highlights the slice configuration and presents the implemented full-adder equations necessary for performing binary addition.

Slices also contain storage elements. These can be configured either as flip-flops or as latches. They allow for fine-grain pipelining of logic designs that increases circuit throughput. For example, if the storage elements in Figure 2.3 are configured as D-Q flip-flops, then the signals $S_{(0)}\_d1$ and $S_{(1)}\_d1$ are available one clock cycle later after signals $S_{(0)}$ and $S_{(1)}$.

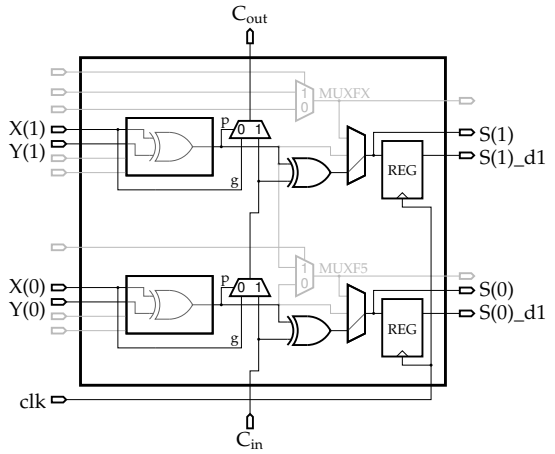
Additional memory-related functionalities are featured by SLICE_Ms:

– the 16-bit LUT memory can also be configured as a synchronous RAM. Consequently, the CLB can be configured as a 16×4, 32×2, 64×1 single-port or 16×2 dual-port (two pairs of ports for reading and writing) memory.

– the 16-bit LUT memory can play the role of a shift-register, often denoted in Xilinx terminology by SRL16. The *16* suffix specifies that one element can delay serial data from one to 16 clock cycles. In order to build larger shift registers (often needed in digital signal processing but also in datapath synchronization in deeply pipelined designs) the Virtex-4 fabric also contains dedicated cascade lines, ripping vertically from top to bottom (Figure 2.2). Consequently. one single CLB (2 SLICE_M) may produce delays of up-to 64 cycles. Moreover, these cascade lines also ripple beyond CLB borders allowing to extend the shift-register length at minor delay increases.

The CLB configuration of the more modern Virtex-5 [23] and Virtex-6 [20] devices differs only slightly from that of Virtex-4 devices. The CLB still contains eight function generators, however these are split into two larger slices (one SLICE_M and one SLICE_L).

The function generators in both Virtex-5 and Virtex-6 devices are implemented as six-input LUTs having two independent outputs (O6 and O5). They can implement any six-input boolean function. In this context the O6 output is used exclusively. Nevertheless, as shown by Rose [33] who searched the optimal LUT size in FPGAs, the optimal LUT size is somewhere between 4 and



$$
\begin{aligned}
s &= a \oplus b \oplus c_{\text{in}} \\
&= p \oplus c_{\text{in}} && \text{– XORCY} \quad (2.1) \\
c_{\text{out}} &= ab + (a \oplus b)c_{\text{in}} \\
&= a\overline{(a \oplus b)} + (a \oplus b)c_{\text{in}} \\
&= \overline{p}a + pc_{\text{in}} && \text{– MUXCY} \quad (2.2) \\
\text{where} \quad & p = a \oplus b && \text{– LUT} \quad (2.3)
\end{aligned}
$$

Figure 2.3 *Ripple-Carry Adder (RCA) implementation in Virtex-4 devices*

6 with a cluster size ranging from 4 to 10, depending on the application. Consequently, in order to maximize the utilization of the LUT memory, two five-input functions can be implemented in the same LUT provided that they share inputs. In this case both the O5 and the O6 outputs are used.

It is often that in pipelined designs, when both O5 and O6 outputs are used, they are synchronized. Either both outputs bypass the storage elements, or they both need storage elements. For a Virtex-5 device, when both LUT outputs need to be registered, the second storage element needs to be used from a close-by free register of a LUT-FF pair. Nevertheless, this introduces important routing delays, especially when no free registers are not found within CLB borders.

In order to overcome this inconvenience, an extra storage element was added to the Virtex-6 slice. Consequently, both LUT outputs have independent storage elements. When used in LUT6 configuration, the second register is unused and is therefore accessible via the general routing network for area efficient design packing.

Several multiplexers are also available allowing multiplexers of up to 16:1 to be implemented in one single slice. Wider multiplexers are possible but require going through the general routing network and are therefore much slower.

There are some differences in the additional functions provided by SLICE_Ms when compared to Virtex-4 devices:

- with an increased memory of 64-bits per function generator the Virtex-5/6 FPGAs provide 4 times more distributed memory per SLICE_M. The supported configurations are numerous allowing single-port memories of up 256 bits with a configuration of 256×1, dual-port memory with configurations 64×3 and 32×6 and quad-port 64×1 or 32×2 for quad-port memories.
- the size of the shift-registers has increased to 32-bit per LUT (SRL32) from the 16 bits per LUT in Virtex-4 devices. However, the cascading connections stop at CLB borders allowing shift-registers of maximum 128 bits (32 bits × 4 function generators for each SLICE_M) to be implemented swithout going through the slow general routing network.

Altera

Each ALM in Stratix-II [14], Stratix-III [22] and Stratix-IV [27] devices is composed out of several LUT resources (one 4-input and two 3-input LUTs for each half ALM) and up-to eight input lines that can be shared between two adaptative look-up tables (ALUTs). As shown in Figure 2.4 each ALUT disposes of 32 bits of programmable memory ($2^4 + 2 \cdot 2^3$) and can therefore implement any function of 4 inputs (16 out of 32 bits used), as for Virtex4 devices. Moreover, the 64-bits of memory corresponding to the two ALUTs in an ALM can be combined to implement any 6-input function . There are several other combinations possible in sharing the 64-bits of data among eight inputs, including the LUT5-LUT3 configuration with independent inputs.

Additional to the flexible ALUT resources, the ALM also contains two registers, one per ALUT. The lack of a supplementary register for the case when the ALUT is configured as two function generators affects performance in pipelined designs, similarly to Virtex-5 devices. The ALMs also features a register chain used to build variable length shift-registers. The register chain stops at LAB boundaries and needs to use the general routing network when its size exceeds 16 bits for Stratix-II and 20 bits for Stratix-III/-IV.

Two dedicated full-adders and a carry-chain are present in each ALM. They provide enhanced hardware support of the RCA scheme. The fast carry chain, similarly to the register chain, does not exceed LAB boundaries. Therefore, binary adders of at most 16 bits for Stratix-II and of 20 bits for Stratix-III/-IV can be instantiated within one LAB. Wider adders are affected by the inter-LAB routing delays.

Additionally, a separate shared arithmetic chain combined with the flexible logic resources allows implementing 3-operand adders in one ALM level. All these presented features are depicted

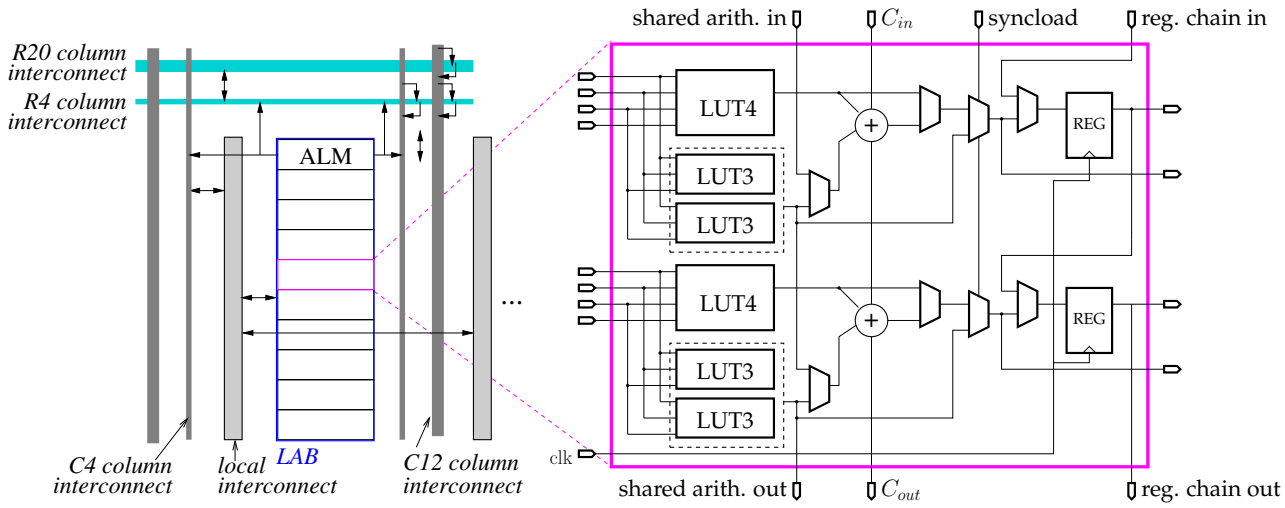


Figure 2.4 *Architectural overview of the ALM block present in Stratix devices*

in Figure 2.4.

Other operating modes supported by Stratix ALMs also include the extended LUT mode. In this mode specific 7-input functions can be implemented in one ALM. The function must follow the 2:1 multiplexer template where each of the two inputs of the multiplexer is being fed by a 5-input line sharing 4-inputs).

Another particularly useful function supported by Stratix ALM is the implementation of max function between two numbers: `R=(X<Y)?Y:X` in one ALM level. This function is often used for exponent management in floating-point operations, but not only. On Xilinx devices this function would require two slice levels, one for obtaining the boolean value of the comparison $X < Y$ (obtained via the Most Significant Bit (MSB) after a 2's complement subtraction) and the second one for multiplexing the two inputs. However, the comparison requires computing only the MSB. Once this bit is computed via regular subtraction, its value is then fed back to the LAB via the syncload line. This line is used for the select line of a multiplexer whose inputs are X and Y.

Stratix-III/-IV offer more user memory than Stratix-II devices. In this devices each LAB is paired with a Memory Logic Array Block (LAB) (MLAB), the CLB equivalent of a SLICE_M. MLABs offer a set of supplementary memory-related features. They allow using the 64-bit ALUT memory in different configurations: either as a 64×1 or a 32×2 dual-port memory block. As these devices contain ten ALMs per LAB, allowing configurations of 64×10 or 32×20 can be implemented in one LAB.

2.1.2 DSP blocks

When first introduced in 2000 by Xilinx in VirtexII devices, these blocks were in fact 18x18-bit embedded multipliers. The first embedded DSP-blocks especially designed with DSP capabilities was introduced in 2003 with the Altera Stratix device: it consisted of four 18x18-bit multipliers, an adder network and a cascading network, the necessary components for most digital filter designs. DSP blocks not only do enhance the performance of these applications but make routing more predictable.

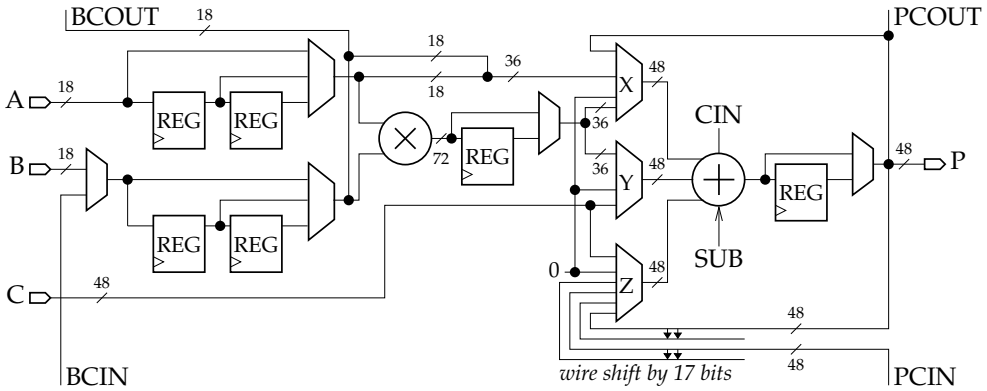


Figure 2.5 *Overview of the Xilinx DSP48*

Table 2.1 *Main operating modes of the Virtex-4 DSP48 block*

$P = Z \pm (X + Y + C)$
$P = Z + A : B$
$P = Z \pm (A \times B + C)$

Xilinx

Nowadays, the Digital Signal Processing (DSP) block of Virtex-4 devices (*DSP48* [16]) basically consist of one 18x18-bit two's complement multiplier followed by a 48-bit sign-extended adder/-subtracter or accumulator unit. The simplified overview of its architecture is depicted in Figure 2.5. The multiplier doesn't output the full 36-bit product, but rather two subproducts aligned on 36-bits. The reason for this is that the adder unit is in fact a 3-operand adder which can be used in this mode whenever the DSP's multiplier is not used. When used in multiplier mode (two adder inputs are occupied by the two sub-products), the third input can either come from global routing (via the C-line) or from the neighboring DSP via the cascading line (PCIN). The possible operating modes of the Virtex-4 DSP48 are presented in Table 2.1.

When in cascaded mode, the result of one block is fed directly into the adder/subtracter unit of the neighboring block via the PCIN input line (Z=PCIN or Z=shiftRight$_{17}$(PCIN)). The possible shift amount of the PCIN input is fixed to 17 bits. The accumulations via cascading lines will allow us to enhance the performance of large integer multipliers by mapping inside DSP blocks most sub-product reductions.

Virtex-5/-6 feature DSP blocks (DSP48E [21] for Virtex-5 and DSP48E1 [25] for Virtex-6 in Xilinx terminology) with larger two's complement 18x25-bit multipliers. The adder/accumulator unit can now perform several other operations such as logic operations or pattern detection. Additionally, the DSP48E1 of Virtex-6 devices includes pre-multiplication adders within the DSP slice. These can be useful for various algorithm, including those in signal processing.

All these DSP blocks feature multiple pipeline registers (up to four levels) which can be used to enhance their performance.

Altera

The Altera DSP blocks have a larger granularity than Xilinx DSPs. In a Stratix-II device a DSP block essentially consists of four 18 x 18 bit multipliers and a flexible adder tree. The DSP blocks are organized into columns, with one element having a height of four LABs, as depicted on the left of Figure 2.6. The DSP block is connected to the rest of the FPGA is by 144-bit in/144-bit out

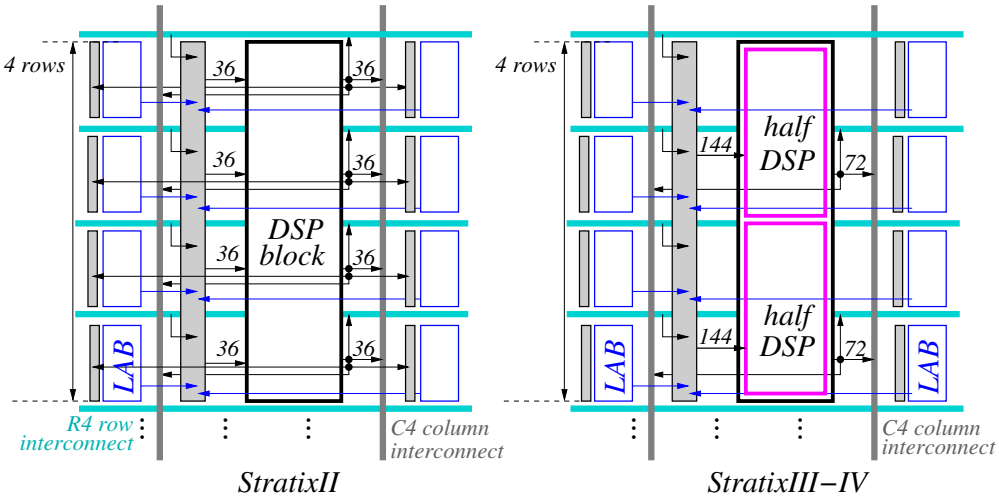


Figure 2.6 *Interconnect of the DSP blocks Left: StratixII and Right: StratixIII-IV devices*

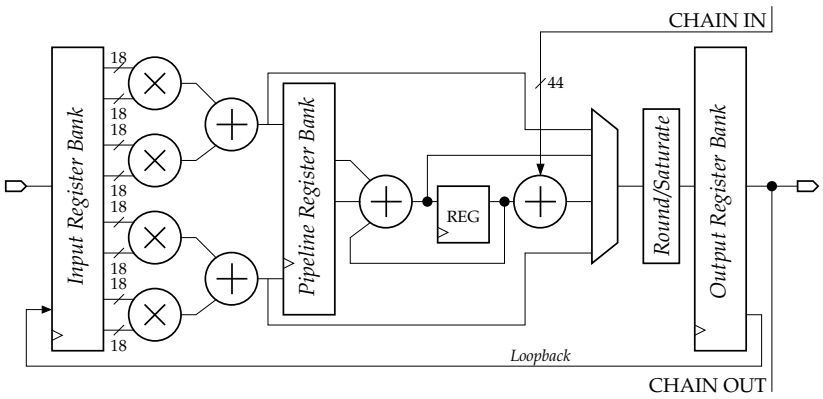


Figure 2.7 *Very simplified overview of the Stratix-III half-DSP block*

data buses. These are sufficient to independently use the four multipliers.

The DSP block in StratixIII-IV device is still four LABs high (the LABs of these devices contain 10 ALMs whereas the StratixII had only 8 ALMs). However, in this devices the DSP block is composed out of two rather independent half-DSP blocks, each of which having similar features to the StratixII DSP block. All in all, the multiplier density is roughly doubled on these devices. The DSP block's input data bus has been correspondingly increased to 288 bits (each half receives 144-bits) but, due to I/O limitations, the DSP's output bus has has the same width as for StratixII devices: 144 bits (72-bits for each half-DSP). The increased multiplier density in the DSP-block greatly benefits DSP applications which rarely need the independent multiplier outputs.

A simplified overview of a half-DSP block architecture is a StratixIII device is presented in Figure 2.7.

The flexibility of the adder tree allows multiple operational modes, among which the 36x36-bit multiplier. Some of the allowed functional modes for Altera Stratix-II/-III/-IV DSP blocks are given in Table 2.2. The *Two Multiplier-Adder* mode can be described by $\sum_{i=0}^{1} a_i b_i$ and the *Four Multiplier-Adder* mode is $\sum_{i=0}^{3} a_i b_i$. Other functionalities (not mentioned in Table 2.2) include cascading the output of one half-DSP to the neighbor's accumulator unit and multiple filter-related enhancements, including hardware support for the 18-bit complex product.

In order to better support floating-point multiplication Stratix-III/-IV devices feature the *double mode* for DSP blocks. In *double mode* the one half-DSP block can perform the reduction of the

Table 2.2 *Operational modes supported by the Stratix-II, Stratix-III and Stratix-IV DSP blocks*

Mode	Width	**Stratix-II** (DSP)	**Stratix-III** (half-DSP)	**Stratix-IV** (half-DSP)
Independent Multiplier	9 x 9 bit	8	4	4
	12 x 12 bit	-	3	3
	18 x 18 bit	4	2	2
	36 x 36 bit	1	1	1
Two Multiplier-Adder	9 x 9 bit	4	*	*
	18 x 18 bit	2	2	2
Four Multiplier-Adder	9 x 9 bit	2	*	*
	18 x 18 bit	1	1	1

sub-products described in equation 2.4. This mode could be useful for other applications as well. Unfortunately, it is currently unavailable as a stand-alone mode using Megawizard.

$$\begin{aligned} X[52:0] \cdot Y[52:0] = & X[35:0] \cdot Y[35:0] + \\ & 2^{36}(X[52:36] \cdot Y[17:0] + X[17:0] \cdot Y[52:36]) + 2^{18}(X[52:36] \cdot Y[35:18] + X[35:18] \cdot Y[52:36])) \\ & 2^{72} X[52:36] \cdot Y[52:36] \end{aligned} \quad (2.4)$$

2.1.3 Block memory

Many FPGA applications require interacting with some memory in order to read/store computation values. Embedded memory blocks are fast, on-chip memories which can be used is such situations. These blocks generally support numerous configurations from RAM, ROM, FIFO, true-dual port memory etc, depending on the application requirements. Their granularity is manufacturer and device dependent. Embedded memory blocks are a essential resource when using FPGAs to evaluate functions using the polynomial approximation technique. One needs adapt the technique (number of intervals, coefficient width) to the target FPGA by accounting for block-memory size in order to maximize the use of these resources.

Xilinx

The embedded memory blocks of Virtex4 FPGAs have a capacity of 18 Kbits of data. Each memory block has two symmetrical and totally independent ports, sharing only the stored data. The ports can independently take different aspect ratios ranging from 16K x 1, 8K x 2, to 512 x 36.

The content of the BRAM memory can be defined by the configuration bitstream. This is a useful feature when using BRAMs as initialized tables, such as those needed for storing precomputed values, eg. coefficients in the polynomial approximation of functions.

In order to achieve higher performance, a pipeline register is available, for optional use, at the data read output inside the memory block. Block RAMs also contains optional address sequencing and control circuitry to operate as a built-in Multi-rate FIFO memory. The FIFO configurations vary from 4Kx4, 2Kx9, 1Kx18, or 512x36. FULL and EMPTY flags are hardwired in Virtex-4 FIFOs.

The block RAMs of Virtex-5 and Virtex-6 [24] FPGAs have an increased capacity of 36K bits. They may be either be configured as two 18Kb RAMs or as one 36Kb. The possible aspect ratios range from 16K x 2 to 1K x 36 for the 36KB RAM and from 16K x 1 to 1K x 18 for the 18Kb RAMs.

Aside from the standard dual-port mode, where two read/write ports are available for each memory content, the Virtex-5/-6 BRAMs also allow simple dual-port mode. This mode is defined as having one read-only port and one write-only port with independent clocks. When operating in this mode, the data-width of the BRAM is doubled to 1K x 72 bits for the 36Kbit version

and to 1K x 36 bits for the 18Kbit version, doubling its capacity. The simple dual-port mode is particularly useful when using BRAMs to store data (coefficients in our case) which are either rarely modified (one port for writing suffices) or are not modified at all during the throughout the entire program execution. The mode allows doubling the amount of data storage at no area or performance penalty.

Altera

The StratixII devices contain three types of memory blocks: M512, M4K, and M-RAM. Their capacities grow from 512bits for M512, to 4Kbits and 144Kbits for the M4K and M-RAM respectively. Out of the three, the M512 block only supports the simple dual-port memory mode whereas both M4K and M-RAM support true dual-port mode. All memory blocks support FIFO functionalities. Moreover, M512 and M4K also feature shift-register functionalities.

The aspect ratio of these blocks is flexible. It varies from 512 x 1 to 32 x 18 for M512, 4K x 1 to 128 x 36 for M4K and 64K x 8 to 4K x 144 for M-RAM. Out of these configurations, for instance, we may use for storing precomputed value for polynomial approximation the 32 x 18 bit mode for M512 and the 128 x 36 bits for M4K.

Stratix-III and Stratix-IV devices provide 3 different types of memory blocks: MLAB, M9K and the M144K. The MLAB (LAB enhanced with memory attributes) has a capacity of 640 bits in ROM mode and 320 bits in RAM mode. The M9K and M144K have capacities of 9Kbits and 144Kbits respectively. They are also the only memory blocks with true dual-port support. All memory blocks provide both FIFO and shift-register support.

The possible aspect ratio vary from 16 x 8 to 16 x 20 for MLAB in RAM mode, 64 x 8 to 32 x 20 for MLAB in ROM mode, 8K x 1 to 256 x 36 for M9K and 16K x 8 to 2K x 72 from M144K.

Now that we have seen the target architectures with their available resources, let us describe the typical flow for in porting an application to an FPGA.

2.2 FPGA design flow

As we have seen, FPGA offer an important number of heterogeneous resources such as logic elements, DSP blocks, block RAMs and many more. Configuring these resources to perform a given computational task requires several steps. These stepsa are represented as a flow-chart in Figure 2.8 and can be performed using vendor tools like the ISE suite from Xilinx or the QuartusII suite from Altera. Circuit synthesis can also be performed using third party tools like Synopsys's Sinplify Pro, Cadence's Encounter RTL compiler, Mentor Graphics's Precision Synthesis or many others. Here is a brief description of these steps.

1. **Design entry**
   The first step of the design flow consists in formally describing the desired functionality of the design using one or a mix of several techniques. Complex designs are composed of components, each with its inputs and outputs and clearly defined functionality. Depending on the component to be described, there are several ways to do this:
   – one of the most common solutions of describing components is using schematics. Design tools such as Xilinx ISE [32] and Altera Quartus II [30] offer integrated schematic editors. The advantage of schematic editors is the ease of porting ideas from the drawing board directly into functional designs. They allow for a system-like description and allow a higher level view on the project. However, managing large projects using the schematic editor can be cumbersome.
   One can use schematic editors to take advantage of FPGA-specific primitives yielding good performance on these platforms. This negatively affects portability: re-targeting the

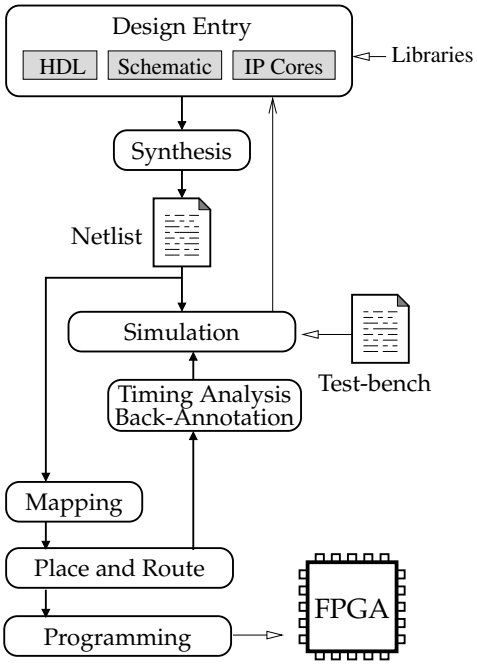


Figure 2.8 *Classical FPGA design flow*

design requires hard work. Moreover, working at gate-level becomes tedious as designs get more complex.
Schematic editors suffer from another great drawback: pipelining. Pipelining designs described using the schematic editor is tedious and error prone. It gets sometimes difficult to follow all the wire and to synchronize them.

– another solution is to use an advanced schematic editor, such as the DSP Builder Advanced from Altera [29]. It works at a higher level of abstraction and features several enhancements with respect to classical schematic editors, such as post-design pipelining, but also comes with a substantial set of optimized primitives. Its downside, as for any schematic editor is the difficulty of describing new, lower-level components.
– a common solution for describing Finite State Machines (FSMs) are state diagrams. Again, most vendor design tools offer specialized editors for this task. Using these editors one can specify in a graphical form the system states, state transitions, and output signals in each state. Once described, these diagrams are compiled towards HDL. The upside of these editors is the visual approach on the task. Its drawback is that is sometimes more time consuming than using VHDL or Verilog.
– the most widely used solution of describing digital circuits is using Harware Description Languages (HDLs) such as VHDL or Verilog. They allow for a higher-level functional description, without detailing the structure at the basic gate level. This significantly reduces the time required to describe complex systems and unless using device-specific libraries, it allows portability between FPGA devices. Although widely used, describing flexible components (fully parametrized and optimally pipelined for each parametrization) is a task out of the reach of HDLs. This type of components is needed in order to take advantage of the FPGA flexibility.
– another solution is to use High-Level Synthesis (HLS) tools to convert specific type of code (usually C) into HDL. Although a working circuit is obtained much faster than for the other approaches, it usually has significantly lower performances.

2. **Synthesis**
   The next step after describing the circuit is synthesis. It consists in the translating the HDL description of the circuit into a netlist. The obtained netlist represents a compact description of the circuit. It is basically a file where the system components, and the interconnection between them and the I/O pins are specified. There exist different formats, such as the Xilinx Netlist Format (XNF) from Xilinx but the industry standard (used by Altera tools) is the Electronic Digital Interchange Format (EDIF).
3. **Simulation**
   Once the netlist with the compact circuit notation is obtained, a simulator is usually used to verify its functionality, before passing to the more time-consuming phases of the process. This verification phase is functional and does not consider implementation-specific signal delays. Essentially, this phase consists in feeding test vectors (each test vector is a new set of circuit inputs) to the simulator and checking the result against the specifications. If errors are found at this phase the designs needs to be refined.
4. **Mapping**
   The mapping step consists of a series of operations which process the netlist and adapt it to the features and available resources of the target FPGA. Some of the most common operations are: adapting to the physical resources of the device, optimizations, and checking the designing rules (such as the available number of pins).
5. **Place and Route**
   The *placing* phase consists in selecting the modules obtained during the mapping phase and assigning them to specific locations on the FPGA device. Once this process is completed, routing consists in interconnecting these blocks using the available routing resources. Both placing and routing are NP-hard problems.
6. **Timing Analysis**
   Having a designed mapped, placed and routed yields new informations on the delays of the signals (interconnection delays) and of the components of the design. This information can be used to produce a new, more detailed netlist (back-annotation) leading to a timing accurate simulation.
7. **Programming**
   At the end of *placing and routing*, a file is generated that contains all the necessary information for configuring the device: logic block configuration and interconnections. This information is stored under the form of bitstream, where each bit indicates the open or close state of a switch on the device. As, most FPGA devices are SRAM based (loose their configuration at power-down), the configuration bitstream is usually loaded into a non-volatile flash-memory (on the same platform as the FPGA) from where it is transfered to the FPGA at power-up.

2.3 Application markets

Programming an FPGAs to perform a given task can easily be performed by following the steps described in section 2.2. However, obtaining optimal performances requires a great deal of expertise and time. Nevertheless, due to their good performances combined with their low cost FPGAs are being used in an increasing number of domains:

ASIC Prototyping
FPGAs have been traditionally used to prototype ASICs. It allows fast RTL testing and substantially decreases development time and reduces the risk of errors in the final ASIC circuit.

ASIC Replacements for medium-volume series and/or future-proofness.

- **Networking**
  The number of users requiring high-performance networks is increasing. At the same time, new services such as video-on-demand, voice over IP and others have hard quality-of-service requirements. The solution is a new generation of intelligent routers, which due to rapidly changing requirements are best implemented using FPGAs.
- **Automotive**
  A new growing application market for FPGAs is automotive. As auto-vehicles become ever more complex features like navigation systems, rear-seat entertainment including movies, audio and even game consoles are being introduced into entry-level cars. Enhanced driver-assistance including safety features such as night-vision, line tracking and pedestrian-detection are also being integrated into top-range cars. The features of recent FPGAs allow high-definition video and audio processing on a single chip. Due to its inherent parallelism, the FPGA can match the throughput of DSPs at lower frequencies, yielding less power consumption and dissipated heat. Driver-assistance safety features require a significant amount of image-processing which FPGAs are particularly well suited for.

High-Performance Computing (HPC)
Nowadays, accelerating the execution of key applications is in ever-increasing demand. Applications requiring vast amounts of calculations such those in bioscience, medical imaging, financial trading and others require significant processing power. The common implementation of these applications in the microprocessor environment is based on floating-point arithmetic (the basic operations are supported in hardware). While a solution based on assembling standard floating-point operators on an FPGA will probably speed-up the computation to some extent, other application-specific solutions exist. These consist of using a mix of both fixed and floating-point arithmetic using custom precisions as dictated by the final required accuracy. Some floating-point data-paths may be fused reducing latency and resource consumption.

All in all, using the FPGA's flexibility to better implement the arithmetic behind the problem is what makes FPGAs viable solutions for accelerating computations and reducing expenses. In this thesis we have particularly focused on the design of efficient and portable floating-point operators, as basic building blocks for accelerating scientific computations using FPGAs.

In the next chapter we introduce the basic notions regarding floating-point arithmetic and explore the state of the art regarding its implementation in the context of FPGAs.

CHAPTER 3
Floating-point arithmetic

3.1 Generalities

Representing and manipulating real numbers efficiently by computers is required in many field of science, engineering, finance and more. There exist several representations for approximating real numbers: fixed-point (chap. 12 [148]), logarithmic [95], continued fractions [97], floating-point (FP) and many more. Out of these representations, FP is the most popular in modern computer systems.

Each of these representation formats promises a different compromise between speed, accuracy, dynamic range and implementation cost. In modern computer systems, the FP representation seems to provide the best balance between these requirements. A detailed description of FP arithmetic in modern computer systems can be found in [124].

The first computer-system to use the binary FP representation of real numbers is Konrad Zuse's Z3 computer [50] which dates from 1941. The format used in the Z3 computer comprised of 22 bits, out of which 14 for the significand, 7 for the exponent and one for the sign.

3.1.1 Representation

Definition 3.1.1 *Let $\xi(\beta, p, e_{min}, e_{max})$ be a FP format where:*
- *β denotes the radix, $\beta \geq 2$*
- *p denotes the precision (the number of significant digits of the representation)*
- *e_{min} and e_{max} are two extremal exponents such that $e_{min} < 0 < e_{max}$*

Definition 3.1.2 *Given a FP format $\xi(\beta, p, e_{min}, e_{max})$ and a real number X we denote by x the best approximation of X representable in the FP format ξ. x is represented by a triplet (s, m, e) such that:*

$$x = (-1)^s \cdot m \cdot \beta^e \tag{3.1}$$

s represents the sign (0 stands for positive and 1 for negative), m is the normal significand having one digit before the radix point and at most $p - 1$ after, and e denotes its exponent.

Using the above definition does not guarantee the uniqueness of representing X in format ξ. Take for instance, two equivalent representations of $X = 177$ in a toy-format $\xi(10, 4, -3, 3)$: $x_1 = 1.77 \cdot 10^2$ and $x_2 = 0.177 \cdot 10^3$.

Table 3.1 *IEEE-754 2008 binary ($\beta = 2$) FP formats*

Common name / Standard	p	e_{min}	e_{max}	w_e	bias
half precision / binary16	10+1	-14	15	5	15
single precision / binary32	23+1	-126	127	8	127
double precision / binary64	52+1	-1022	1023	11	1023
quadruple precision / binary128	112+1	-16382	16383	15	16383

In practice, it is often required that the FP number representation is *normalized*. A normalized representation $x = (s, m, e)$ of a number X in format ξ requires that $m \geq 1$. In other words, normalization requires that the digit before the radix dot is not zero.

In the case of *binary* FP arithmetic ($\beta = 2$) this leads to $m \in [1, 2)$. Thus, the normalized representation when $\beta = 2$ always has a leading '1'. Because this bit is constantly '1', the binary-floating point format used in many computer systems don't store it (it if often referred to as the "hidden bit" or the implicit bit).

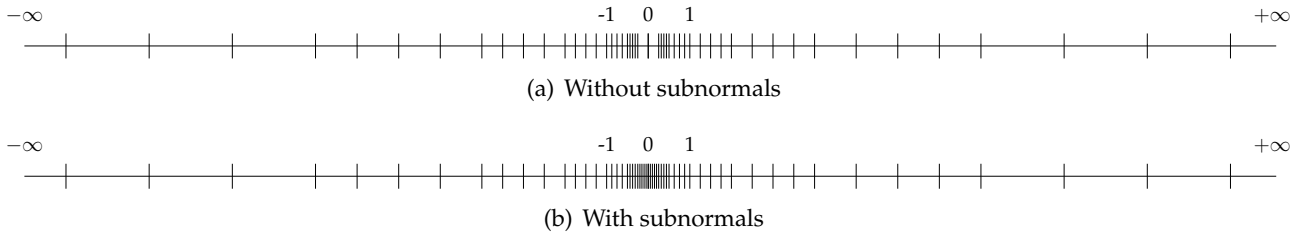
The IEEE-754 standard for FP arithmetic, introduced in 1985 and revised in 2008 [17] defines the formats of several FP representations. In addition to the *single precision* and *double precision* formats present in the 1985-version of the standard, the new revision introduces two new formats for binary: *half* and *quadruple precision* and also several equivalent formats for decimal FP. The binary formats of the IEEE-754-2008 standard and their parameters are presented in Table 3.1. Nowadays, the microprocessors offer hardware FPU support for basic arithmetic operations in single and double precision.

We can clearly see that the FP formats used by the standard can be easily generalized for an arbitrary precision p and exponent range. Nevertheless, using such custom FP formats in microprocessors, where the FPU only supports single and double precision will bring no improvements and will probably lead to significant speed penalties due to the custom data-type overhead. As an example, consider an application for which a 26-bit precision datapath suffices to attain the required accuracy. Single precision does not offer the required accuracy and the next best thing, from a performance perspective, is to use double precision. This is a situation where it is perfectly justified to use double precision. On the other hand, consider implementing the same application on an FPGA. Simply instantiating double-precision FP cores would do the job at the expense of a significant implementation size. However, in an FPGA one can instantiate custom operators, having just the right precision (26 is our example). The implementation cost of the 26-bit precision datapath, when compared to that of single-precision (24-bits of precision) is minimal, but compared to the naive implementation using double-precision operators would be significant. Considering that the application's accuracy requirements were met in both cases, its seems obvious to use custom floating point formats in such a situation.

One nice property of the FP formats defined by the IEEE-754 is that it allows comparing two numbers just by considering their fqbinary representation. If the numbers have different signs, a 1-bit comparison suffices whereas if the signs are similar, a binary comparison on the rest of the bits suffices. Take for instance the example of single precision (SP) with numbers having similar signs: a 31-bit binary comparison suffices to decide their order. The trick that allows for this is exponent biasing (the bias value for each format is presented in Table 3.1). Positive exponent represent numbers larger than one, whereas negative exponents represent numbers in $[0, 1)$. Using a signed-magnitude or two's complement representation, instead of the biased exponent representation, would add several additional calculations to the computation's critical path. Take for an example $X = 2^{-2}1.10$ and $Y = 2^{1}1.11$, with the exponent representation on 4 bits: in 2's complement `-2=1110` and `1=0001` and in biased representation: `-2=0101` and `1=1000`. The comparison result is immediate in the biased representation.

Table 3.2 *Binary encodings of exceptions in the IEEE-754 Standard*

Exception	s	e_{biased}	fraction
-0	1	0	0
$+0$	0	0	0
$-\inf$	1	$2^{w_e}-1$	0
$+\inf$	0	$2^{w_e}-1$	0
NaN	0	$2^{w_e}-1$	>0



(a) Without subnormals

(b) With subnormals

Figure 3.1 *Distribution of floating-point numbers in a system $\xi(2,3,-2,3)$, having a IEEE-754 equivalent $p=3$ and $w_e=3$. The -3 and 4 values of e are used to represent the special cases presented in table 3.2*

The IEEE-754 standard uses two exponent values to encode special cases: $e_{\text{biased}}=0$ and $e_{\text{biased}}=2^{w_e}-1$. These exponent values, together with special fraction values are used to encode different exceptions. Table 3.2 presents the different exceptions and their encoding as specified by the IEEE-754 standard.

Encoding exceptions using a combination of exponent-fraction value reduces the range of representable numbers, increases implementation size but also ensures trade-off with the format size (in the case of SP for example, some numbers are lost but the format still fits in 32 bits). In the case of FPGAs we can extend the format with a few bits (2 bits will suffice) in order to encode exceptions. The two extra bits will have little impact on the circuit's area but the compact encoding will reduce the hardware necessary for their decoding and improve the latency. Their significance can be: 00 - zero, 01 - normal number, 10 - $\inf$ and 11 - NaN. This way of encoding exceptions was first introduced by Detrey and de Dinechin in FPLibrary [67], a parametrized library of floating-point operators for FPGAs.

The distribution of FP numbers in a toy system $\xi(2,3,-2,3)$ is given in Figure 3.1(a). One can observe that roughly half the representable FP numbers are found in the interval [-1,1]. As one moves away from this interval, the gap between successive FP numbers increases by powers of two. Nevertheless, one can observe that between zero and the first representable number there is a gap.

This gap leads to one of the most controversial parts of the 1985 standard, especially regarding hardware implementations: *subnormals*. Subnormals are FP numbers which are not normalized (hence sub-normals). They are obtained by using the minimum value of the exponent (the same that codes zero) but have a non-normalized significand (no more hidden '1') different than zero. These numbers allow covering the gap between zero and the first representable numbers and allow for a *gradual underflow*. It is due to them that some algorithms are numerically stable.

Early works proved that subnormals are the costliest part to implement in FP units [139]. At the time, when transistors were still expensive, this made their mainstream acceptance controversial for microprocessors. Nowadays, with the more than 80% of the microprocessor's die occupied by cache memories, the their presence in the microprocessor's FPUs (even in GPUs) has become universal.

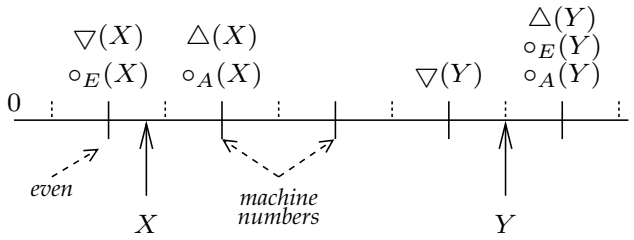


Figure 3.2 *The rounding modes specified by the IEEE-754 2008 standard*

On the other hand, FPGAs provide another alternative to the (still) costly support of subnormals. Extending the exponent width by one bit doubles the possible representable values. Not only that all subnormals are now representable, but also larger magnitude values as well. The width increase of 1 bit has little impact on the operator's cost (in any case much less costly than the effective support of subnormals) and provides the FPGA-specific alternative to subnormal support.

3.1.2 Rounding

Rounding errors are inherent in floating-point computations. The simplest operations, like the addition and multiplication do not always generate results representable in the target FP format. The obtained result needs to be *rounded* to a FP value in that format. More formally, given a FP format ξ, and two members of this format a, b, the result $X = a \text{ op } b$ is not usually representable in ξ. The operation of approximating X to a number x, $x \in \xi$ is called *rounding*. We generically denote by *machine number* a FP number which can be exactly represented in a FP format ξ.

The IEEE-754-2008 [17] standard defines three directed rounding modes:

- round towards **negative**: $\bigtriangledown(X)$ is the largest machine number less than or equal to X;
- round towards **positive**: $\bigtriangleup(X)$ is the smallest machine number greater than or equal to X;
- round towards **zero**: $Z(X)$ is $\bigtriangledown(X)$ when $X > 0$ and $\bigtriangleup(X)$ when $X \leq 0$

and two variations of the round towards **nearest** mode. Both modes return the closest machine-number to X. The differ in output only when X is exactly half-way between two FP numbers:

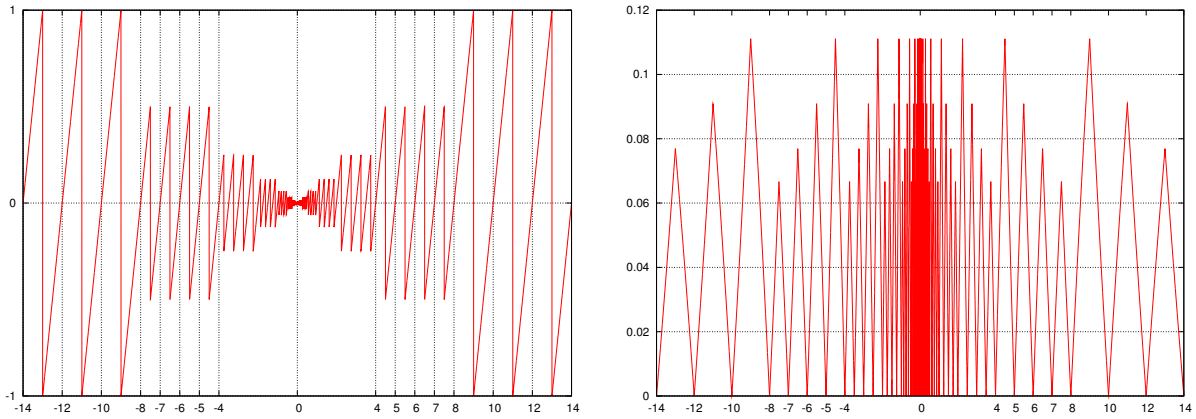
- **roundTiesToEven**: $\circ_E(X)$ is the closest machine number to X. When X is exactly in the middle of two machine numbers then the one with an *even* significand will be returned.
- **roundTiesToAway**: $\circ_A(X)$ is the closest machine number to X. When X is exactly in the middle of two machine numbers then the one with the *larger magnitude* will be returned.

The five rounding modes are illustrated in Figure 3.2.

When the result of a function is rounded according to a given rounding mode one says that the function is *correctly rounded*. A *rounding breakpoint* is defined as a value when the rounding function changes. In the case of **round towards nearest**, for example, the rounding breakpoints are the exact middles of consecutive FP numbers.

For arithmetic operations such as $+, -, /, \sqrt{x}$ it is fairly easy to return a *correctly rounded* result. For other functions, such as the elementary functions, it is quite difficult to return such a *correctly rounded* result. This is because the "gray area" of uncertainty on the computed result may contain one such rounding breakpoint. In such a case one should increase the computation accuracy until either the gray area contains no breakpoint or the gray-area is exactly one such rounding breakpoint. The problem of making a correct decision in such a case is called *Table Maker's Dilemma* [124].

Due to the difficulty of this problem, a relaxed version of this requirement is often used in the literature. In *faithful rounding*, when the result is in a gray-area of uncertainty either of the two



(a) absolute error (b) relative error

Figure 3.3 *The absolute and relative errors of our representation*

possible results can be returned. The name is misleading as *faithful rounding* is not a rounding mode: the result of the operation is not uniquely specified which implies that using this mode will break portability: two different platforms may not return bit-for-bit identical results.

3.1.3 Errors

When trying to represent infinitely precise real numbers using a computer system one needs to approximate these with numbers in a representable format. The choice of the format influences several parameters among which the *range* of representable numbers and the *accuracy* characteristics.

Two of the tools used to measure accuracy are *relative* and *absolute* errors. We denote by $\diamond$ the active rounding mode, which can be any of those presented in section 3.1.2. The relative error is defined as:

$$\epsilon(X) = \left| \frac{X - \diamond(X)}{X} \right| \tag{3.2}$$

and the *absolute* error is defined as:

$$\epsilon_a(X) = |X - \diamond(X)| \tag{3.3}$$

In the case of our toy FP format $\xi(\beta = 2, p = 3, e_{min} = -2, e_{max} = 3)$ having the representable number distribution depicted in Figure 3.1(b), and choosing the active rounding mode $\diamond = \circ_E$, the relative and absolute errors of representing real numbers in this format are depicted in Figure 3.3.

For the round towards nearest rounding modes, and X in the *normal* range (excluding subnormals) the relative error is bounded by $\frac{1}{2}\beta^{1-p}$ and for the directed rounding modes it is bounded by β^{1-p}. When x is exactly 0 it is considered that the relative error is 0. When X is in the subnormal range, the relative error can get as big as 1.

The absolute error provided that we allow subnormals, is bounded by:

$$\epsilon_a(x) \leq \begin{cases} \frac{1}{2}\beta^{e_{min}-p+1} & \text{when } \diamond = \circ \\ \beta^{e_{min}-p+1} & \text{otherwise.} \end{cases}$$

and when no subnormals are allowed this bound becomes:

$$\epsilon_a(x) \leq \begin{cases} \frac{1}{2}\beta^{e_{min}} & \text{when } \diamond = \circ \\ \beta^{e_{min}} & \text{otherwise.} \end{cases}$$

Errors are very often expressed in terms of relative errors. However, it is sometimes desirable to be able to express errors in a more atomic way: the weight of the last bit of the significand. One definition, by Harrison [89], of the *unit in the last place* states:

Definition 3.1.3 *$ulp(X)$ is the distance between the closest standing FP numbers a and b ($a \leq X \leq b$, with $a \neq b$) assuming that the exponent range is not upper-bounded.*

Now, given the error in ulps of a given computation we can easily translate this error into a relative error. We take a computation where X denotes the real result and x is the machine-number representation of X with $|X - x| = \alpha \cdot ulp(X)$. The relative error, provided that there is no underflow is:

$$\left|\frac{X - x}{X}\right| \leq \alpha \cdot \beta^{-p+1} \tag{3.4}$$

And the other way around, given the relative error $\epsilon(X)$, the error in ulps is:

$$|X - x| \leq \epsilon \cdot \beta^{p} \cdot ulp(X) \tag{3.5}$$

3.2 Floating-point arithmetic on FPGAs

Floating-point seems to be a good compromise between dynamic range, accuracy and implementation complexity when trying to manipulate real numbers. This is one of the main reasons why FP arithmetic is extensively used in scientific algorithms.

Popular programming languages used in scientific computing C and Fortran provide dedicated datatypes for manipulating FP variables. These languages natively support the IEEE-754 single-precision (datatype name is `float` in C) and IEEE-754 double precision (DP) (`double` in C). Depending on the target architecture, some of the basic operations on these datatypes are directly supported by the Floating-Point Unit (FPU) while less frequently used operations, not supported in hardware, are implemented by means of mathematical libraries (libms). The overhead of going through a libm is translated into roughly two orders of magnitude slowdown over the same operator implemented in silicon.

Scientific computations usually also involve more operations than just additions and multiplications (supported in hardware by most FPUs). They require divisions, trigonometric functions, exponentials, logarithms, square-roots, accumulations and other. One example of such application is circuit modeling in SPICE. Figure 3.4 presents the distribution of these operations for modeling the electronic components. These electronic components basic blocks of the SPICE circuit modeling tool [94]. When simulating these circuits using microprocessors, most of the time is spent evaluating elementary functions (log, exp) which are not supported in silicon. Performance drops even more if these are found deep inside in the inner loops of the code. Nowadays, FPGA implementations of these operators can offer the same throughput as for basic operators, offering as significant speedup compared to the microprocessor counterpart when simulating these models. However, this was not always the case.

Early FPGAs had such low densities that basic IEEE-754 single-precision operators occupied the entire device. Nevertheless, researchers were keen to prove that, even in these conditions, FPGAs could be used to accelerate applications that required FP arithmetic. Their solution was to use custom "smaller-precision" FP operators.

Models	Instruction Distribution					
	Add	Mult.	Div.	Sqrt.	Exp.	Log
bjt	22	30	17	0	2	0
diode	7	5	4	0	1	2
hbt	112	57	51	0	23	18
jfet	13	31	2	0	2	0
mos1	24	36	7	1	0	0
vbic	36	43	18	1	10	4

Figure 3.4 *Instruction distribution in SPICE circuit modeling using FPGAs [94]*

A pioneering work in this context is due to Shirazi et al. [140] in 1995. The work proposed two custom FP formats: (1) an 18-bit format having 7 bits of exponent and 10 bits of fraction, ideal for packing 2 operands onto the 36-bit wide datapath width of the Splash-2 system [36] (2) a 16-bit format with 6 bits if exponent and 9 bits of fraction, ideal for the 16-bit wide external memories available for each FPGA of the Splash2 system. The chosen formats proved to provide sufficient dynamic range and accuracy for implementing Fast Fourier Transforms (FFTs) and Finite Impulse Response (FIR) filters using basic operators for this formats. We believe that adapting the working FP formats: (1) to account for the application's accuracy needs and (2) to better fit the deployment FPGA, is part of the recipe of obtaining good accelerations using FPGAs.

In the spirit of adapting the arithmetic datapath width to the application's accuracy requirements, Gaffar et al. [85] proposed a tool for automatically customizing the FP formats in FPGA designs. The tool inputs a cost function, specifying the maximum allowed output error relative to a reference representation. Heuristically and based on input data vectors the tool evaluates several datapaths, using combinations of intermediary formats. The results obtained on representative application data confirm that good savings in terms of area and increased performances can be obtained for a user-defined accuracy criteria. What was still needed were libraries of FP arithmetic operators which better used the FPGA's resources.

Lee and Burgess [104] proposed Virtex-II optimized architectures for the basic operations: $+, \times, \div$ and $\sqrt{}$ for IEEE-754 single precision, double precision and also other custom formats. Their implementations make good used of the multiplexers, XOR gates, carry chains, embedded multipliers which allows better implementation performances. To our knowledge, the presented architectures were never made publicly available. Roesler and Nelson [134] also explore the impact of embedded multipliers and shift registers (SRL16) in the context of deeply pipelined FP units. The results obtained on adders, multipliers and multiplication-based dividers are enough to conclude that the IEEE-754 single-precision is poor match for the multiplier and divider architectures due to the 17-bit width of the embedded multiplier blocks of Xilinx devices. Due to the FPGA features the architectures present *sweet spots* – formats for which the use of these features is maximized. Roesler and Nelson suggest that these sweet-spot formats should be preferred whenever IEEE-754 compliance is not mandatory, provided that the obtained datapath meets the application's accuracy requirements.

In this spirit is the recent work by Langhammer on the Altera Floating-Point Datapath Compiler [100]. The compiler inputs and outputs numbers in IEEE-754 format (SP is discussed) but uses alternative internal representations and fuses similar operations into clusters. The representation format within these clusters are operation-dependent, for instance multiplier clusters use extended fractions (32-bit) which better fit the DSP blocks of Altera devices (36-bit for StratixII-IV). Using this extended formats allows relaxing the normalization stages within a cluster which reduces resource occupation and latency.

Several other libraries of parametrized FP operators were developed in the context of accelerating applications: Lienhart et al. [114] in the case of N-body simulations and Belanović and Leeser [43] for K-means clustering. FPLibrary [67], the precursor of FloPoCo was also released at the time: it contained the basic operators: $+, \times, \div, \sqrt{x}$ but also conversion operators between fixed to FP, and from the internal FP format to IEEE-754 format. In was only years later, in 2005, that main FPGA manufacturers Altera and Xilinx shipped their first FP cores.

An exploration of the performances of IEEE-754 compliant double-precision operators on modern FPGAs is given by Govindu et al. [86]. IEEE-754 compliance includes subnormal support, exception management and rounding-mode support. The implementation results prove that IEEE-754 compliance have a strong impact on implementation size.

However, when an FPGA implementation targets the accuracy of final result, rather than bit-to-bit compatibility with the IEEE-754 compliant software [1], then several FPGA-specific methods can be applied. We list here just a few, having direct connection with the FP representation:

- using a mix-and-match between custom fixed and floating point formats can significantly save resources, when applicable.
- extending datapath width by 1 bit and employing rounding towards zero (truncation); the accuracy of a result obtained using truncated rounding mode on $wF + 1$ bits is similar to that obtained using the more costly round-to-nearest rounding mode on wF bits of precision. This saves an (possibly pipelined) addition proportional to the FP format's size
- faithful rounding ensures the same accuracy as the directed rounding modes but breaks portability. Faithfully rounded FP multipliers on $wF + 1$ bits of precision offer the same accuracy as those implementing round-to-nearest on wF bits. They can significantly save resources (see Chapter 6). The same holds for implementing a faithfully rounded $\sqrt{x}$ (see Chapter 8) and other elementary functions using polynomial approximation (for e^x see Chapter 9). The overhead of these solutions is that the 1-bit datapath increase which has little impact on the final area.

A complementary approach to take advantage of the FPGA's flexibility is to go beyond basic FP operations and formats. Two works in this direction are due to Detrey and de Dinechin [70, 68] in the context of implementing FPGA-specific architectures for the FP exponential and logarithm. By using this approach the estimation by Underwood that FPGAs will surpass microprocessors by one order of magnitude by 2009 was already attained in 2005.

Exploring this flexibility when implementing FP operators is difficult and sometimes impossible to do just by using VHDL, as Detrey and de Dinechin concluded when implementing the exponential and logarithm operators. For these special architectures they made uses of several external programs: one of them for populating tables with values, another for the application-specific instances of specialized operators (for example constant multipliers). All this proves that efficient operator generation needs the power high-level programming languages, but also require the fine-grain control over generated code the VHDL offers: it needs operator generators.

The goal of the FloPoCo framework is to provide such an environment where FPGA-specific operators can be developed. Its philosophy is that FP on FPGAs should not rely on operators that mimic those available in processors, but on radically different new operators, which may obtain more accurate, have shorter latency and require less hardware resources.

The advantages of using FloPoCo to design custom arithmetic data-paths, and its features will be detailed in the next chapter.

1. the obtained result might also be more accurate than the software counterpart

Chapter 4
Custom arithmetic data-path design

There are several different ways to arithmetic pipelines. One solution consists in assembling and synchronizing the pipeline by hand using as support operator libraries or operator generators such as Xilinx Logicore [6], Altera Megawizard [9], FPLibrary [67], VFLOAT [153] and many other. In this approach the user has full control over the choice of subcomponents and their characteristics: input/output precision, latency etc. which potentially allows building efficient circuits. The drawback of this approach lies in the long design cycles needed to build such pipelined system by hand: if the performance is not acceptable some components need to be pipelined deeper and the system resynchronized.

Another, definitely more expensive, solution is to use High-Level Synthesis (HLS) tools. These tools start with a `C`-language description of the arithmetic datapath and produce either a VHDL or RTL description of the circuit. Tools like Cynthesizer [8] are quite flexible and allow the description to use a mix of integer, fixed and floating-point data-types. They are mostly used in ASIC design and lack the optimization support for FPGAs. Others tools like ImpulseC [3] are more FPGA-specific and can use floating-point operators from Logicore or Megawizard. They support integer, fixed-point and floating-point data-types and assemble arithmetic pipelines out of these components. Nevertheless, the generated arithmetic datapaths lack fine-grain optimizations. Other tools like CatapultC [1] provide datapath support only for integer and fixed-point internally convert floating-point description into fixed-point. To sum up, current HLS tools offer sufficient data-types to allow users to express simple arithmetic pipelines. However, using `C` to express the fine-grain signal manipulation is problematic, and these manipulations are predominant in describing efficient arithmetic operators and coarser arithmetic datapaths in FPGAs. Moreover, parameterizing such a description is another challenge.

A more FPGA-specific solution which is arguably better when the data-type representation is *restricted to floating-point* is given by the Floating-Point Datapath Compiler by Langhammer [100]. The tool uses an internal data-path precision which maximizes the resource usage by targeting the sweet-spots of Altera devices. It fuses similar operators in clusters (intermediary denormalization/normalization steps are saved) in order to reduce latency and implementation size over alternative implementations based on operator assembly (HLS tools).

Between these approaches, none of them is really suited for an efficient and complete arithmetic datapath description: using libraries and assembling datapaths by hand is slow, non-portable and error prone; using HLS tools is too high-level to effectively express the semantics needed in arithmetic pipeline design; using the Floating-Point Datapath Compiler is restricted to floating-point pipelines.

Our work focuses on providing an extensible open-source framework which is well suited for describing efficient FPGA-specific arithmetic pipelines. The FloPoCo framework embeds the full

power of HDL, necessary for fine-grain manipulation of signals. It also allows, by means of sub-components, to mix the description's granularity level from low-level pipelined binary adders, to the evaluation of elementary functions in fixed and floating point and also the implementation of custom-precision floating point data-paths from `C` code using the FPPipeline macro operator. Additionally, FloPoCo assists in this description by automating signal management, path synchronization, and also provides abstract delay primitives which allow the described circuit to be pipelined at a user-specified frequency (automatic frequency-directed pipelining) and to be retargeted to other FPGA devices. Once description is finished, the FloPoCo framework also assist in numerical validation of the designed operators by providing a test-bench generation suite.

4.1 Arithmetic operators

In this chapter we consider *arithmetic operators* as being circuits that can be described by a function:

$$f(X) = Y$$

where $X = x_0, ..., x_{n-1}$ is a set of inputs and $Y = y_0, ..., y_{m-1}$ is a set of outputs. Any sequential code without feedback loops performing some computations fits this description. Take for example the circuit performing the complex multiplication: $(a + bj)(c + dj)$. The circuit inputs are a, b, c, d and the output is the pair $r = ac - bd, i = ad + bc$. The restriction to this class of arithmetic operators allows us to build provably correct-by-construction pipelines for these circuits.

4.1.1 FPGA-specific arithmetic operator design

Two of the factors characterizing the quality of arithmetic operators on FPGAs are *circuit area* and *operating frequency*. Generally there is a monotonic dependency between the two: the faster a circuit is, the more resources it takes. It is often that the target frequency f part of the project's specifications so the designer's goal is to build the circuit taking the "smallest" area (a maximum value for the area is also accepted) matching this frequency in a given amount of time.

The success in achieving this goal relates to the engineer's *productivity* and depends on his prior expertise and with the performance of the tools used in this process.

Depending on the required quality and the given time-frame one solution is to *assemble the datapath from custom components* built all for for same frequency f. By construction, the system's frequency will also be close to f, depending on the routing congestion.

Provided more development time is available, a better solution is to internally optimize the given datapath. We will prove that this approach can significantly lower the operator's size while improving its accuracy.

The target of the FloPoCo framework is to allow users to explore both ends of the productivity spectrum in designing arithmetic circuits for FPGAs. Unexperienced users are offered the possibility to quickly assemble arithmetic datapaths, pipelined for a given frequency on a given FPGA target. Experienced users can easily explore the arithmetic realm of FPGA devices by having a framework which automates error-prone and tedious tasks such as pipelining.

For complying with these demands, our framework:

- provides quality implementations of all the basic off-the-shelf blocks available in commercial core generators and more;
- provides the mechanisms for easily connecting and synchronizing these blocks;
- is expressive enough to capture low-level FPGA-specific architectural optimizations when needed;
- employs frequency-directed pipelining for minimizing circuit area and pipeline depth;

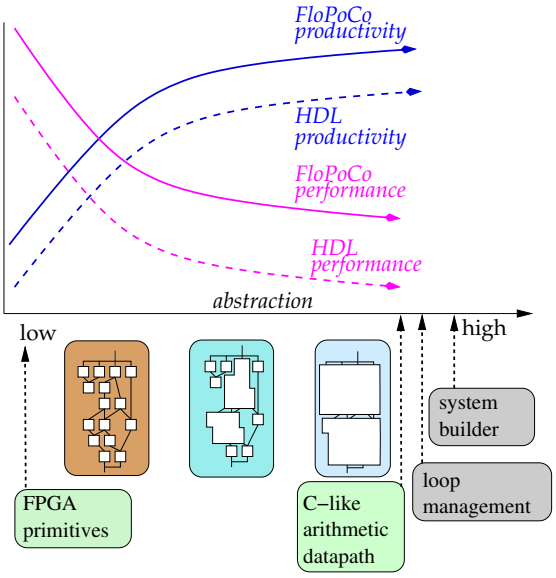


Figure 4.1 *Productivity in porting applications to FPGAs and the relative performance of these circuits provided the different levels of abstraction are provided for circuit description*

– enhances productivity by stressing *reusability*. Each new operator described using the framework becomes part of the ever-increasing *available operators* list;
– encourages parametric description of circuits so they can easily be retuned to changing precision requirements;
– allows to easily re-target existing operator descriptions to new FPGAs by providing a high-level abstraction of FPGA features.

Figure 4.1 presents the target of FloPoCo: enhancing the description productivity at all levels when compared to hardware description languages, while offering better performances due to the enhanced architectural generation, operator construction and design space exploration allowed by the supporting programming language.

4.1.2 From libraries to generators

Although early FP operators were proposed as VHDL or Verilog libraries, the current trend is to shift to generators of operators (see [38] and references therein). A generator is a program that inputs user specifications, performs any relevant architectural exploration and construction (sometimes down to pre-placement), and outputs the architecture in a synthesisable format. Most FPGA designers are familiar with the Xilinx core generator tool [6], which to our knowledge has pioneered this approach, or its Altera MegaWizard [9] counterpart.

A generator may simply wrap a library, and for the simplest operators there is no need for more, but it can also be much more powerful. For instance, the size of a library including multipliers by all the possible constants would be infinite, but the generation of an architecture for such a multiplier may be automated as a program that inputs the constant [49]. Detrey and de Dinechin [69] have shown that the same approach can also be applied for table-based arbitrary function evaluation implementations.

Generators allow for greater parameterization and flexibility. Whether the best operator is a slow and small one, or a fast but larger one, depends on the context. FPGAs also allow flexibility in precision: arithmetic cores should be parameterized by the bit-widths of their inputs and out-

puts. We are also concerned about optimizing the operators for different hardware targets, with different LUT structure, memory and DSP features, etc. The more flexible a generator, the more future-proof.

Lastly, generators may perform arbitrarily complex design-space exploration, automated error analysis, and optimization [69, 74].

4.2 Design choices for FloPoCo

An architecture generator needs a back-end to actually implement the resulting circuit. The most elegant solution is to write a generator as an overlay on a software-based HDL such as SystemC, JBits, HandelC or JHDL (among many others). The advantages are a preexisting abstraction of a circuit, and simple integration with a one-step compilation process. The inconvenience is that most of these languages are still relatively confidential and restricted in the FPGAs they support. Even SystemC synthesizers are far from being commonplace yet.

Basing our generator on a vendor generator would be an option, but would mean restricting it to one FPGA family. We chose less restrictive route by implementing our generator from scratch in a mainstream programming language. The chosen language was C++ due to its popularity, compatibility with existing libraries: MPFR multi-precision library for test-bench suite generation and Sollya [54], a floating-point software environment for generating approximation polynomials.

Thanks to the C++ language, our generator is in theory portable. Nevertheless, due to its dependency to Sollya for generating function evaluators, is only functional on Linux platforms. On the other hand, the generated operators are printed in vendor-independent VHDL which allows it to be easily integrated into existing FPGA projects, simulated using mainstream simulators (for the purpose of testing) and synthesized for any FPGA using the vendor back-end tools. Moreover, the generated VHDL may be specifically optimized for the target FPGA.

4.3 A motivating example

This framework has been used to write and pipeline very large components, such as the floating-point exponential described in Chapter 9. Nevertheless, we choose for clarity in this chapter to discuss a simpler, but still representative example: the implementation of a sum-of-squares operator used in the implementation of 3D norms: $r = x^2 + y^2 + z^2$.

A first option is to assemble three FP multipliers and two FP adders. For this, the command line:

```
flopoco -target=Virtex4 -frequency=200 FPAdder 10 36
```

will generate synthesizable VHDL for a floating-point adder, pipelined to run at 200MHz on a Xilinx Virtex4, using a custom floating-point format with 10 bits of exponent and 36 bits of significand (this format is intermediate between the standard single- and double-precisions).

For design exploration, the 4 parameters we have in this example (target FPGA, frequency, exponent size and significand size) can be changed within sensible range. The frequency and the precision are orthogonal parameters, as they should be, and the pipeline depth is reported.

More complex datapaths can be obtained in seconds using the `FPPipeline` meta-operator of FloPoCo. Assume the file `SumOfSquares.fpc` contains the following pseudo-program:

```
R = X*X + Y*Y + Z*Z;
output R;
```

The command line:

```
flopoco -target=Virtex4 -frequency=300 FPPipeline SumOfSquares.fpc 9 31
```

will generate the VHDL for a complete floating-point pipelined datapath.

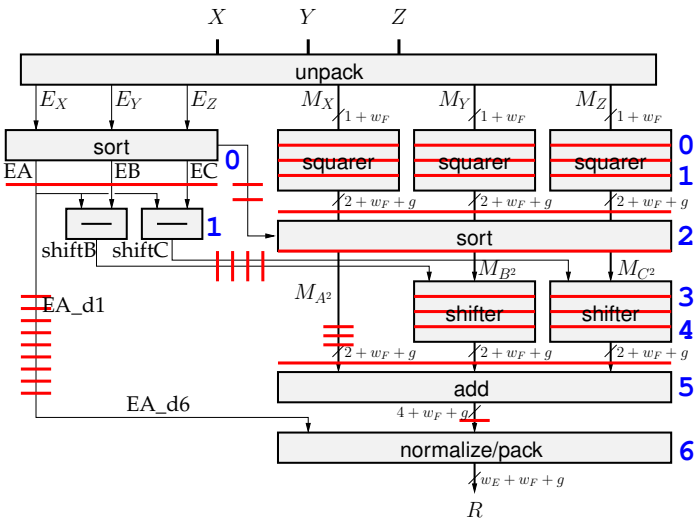


Figure 4.2 *Optimized architecture for the Sum-of-Squares operator*

A better implementation can be obtained by designing a specific operator for this computation. There are many optimization opportunities with respect to the previous solution:

- Squarers are simpler than multipliers. They will in particular use less DSP blocks.
- We only add positive numbers. In an off-the-shelf floating-point adder, roughly half the logic is dedicated to effective subtraction, and can be saved here. An optimizing synthesizer would probably perform this optimization, but it will not, for instance, reduce the pipeline depth accordingly.
- The significands of x^2, y^2 and z^2 are aligned before addition. This may be done in parallel, reducing pipeline depth and register usage with respect to an implementation using two floating-point adders in sequence.
- A lot of logic is dedicated to rounding intermediate results, and can be saved by considering the compound operator as an atomic one [101]. We obtain an operator that is not only simpler, but also more accurate.

It is common for a fused operator to be more accurate than the combination of FP ones by using an extended internal precision and saving on the number of roundings. More subtly, with an operator built by assembling two FP adders, there are some rare cases when the value of the sum will change when one exchanges x and z due to the order of the two consecutive roundings. Our proposed design won't have such asymmetries and will be more accurate than the one obtained by assembling FP operators.

The chosen architecture is depicted on Figure 4.2. Here, we wish to evaluate $x^2+y^2+z^2$ with 1-ulp (unit in the last place) accuracy, knowing that the final rounding will introduce 0.5-ulp when the round-to-nearest rounding mode is employed.

This architecture computes the three squares, truncates them to $wF+g$ bits, then aligns them to the largest one and adds them. Worst-case error analysis shows that there are 5 truncation errors in this computation (the three products, and bits discarded by the two shifters). The number of guard bits g is therefore set to $g = 1+\lceil log_2(5) \rceil = 4$ so that the accumulated error is always smaller than 0.5 ulp of r. The final rounding is upper bounded by 0.5 ulp yielding an architecture having a maximum error of 1 ulp.

Table 4.1 presents the different trade-offs offered by the FloPoCo framework in terms of performance, productivity, flexibility and portability. The first part of the table presents the results for the productivity-performance metric. Assembling Logicore operators for each of the listed

Table 4.1 *Some synthesis results for $x^2 + y^2 + z^2$.*

Productivity versus performance on Virtex4, target frequency $f = 350$ MHz			
format	approach	performance	cost
(8,23)	LogiCore	34 cycles @ 482 MHz	1356 slices, 12 DSP
	option 1	35 cycles @ 327 MHz	1279 slices, 12 DSP
	option 2	35 cycles @ 333 MHz	1043 slices, 9 DSP
	option 3	11 cycles @ 369 MHz	470 slices, 9 DSP
(11,52)	LogiCore	50 cycles @ 354 MHz	3074 slices, 48 DSP
	option 1	47 cycles @ 319 MHz	3859 slices, 48 DSP
	option 2	45 cycles @ 322 MHz	3137 slices, 18 DPS
	option 3	16 cycles @ 368 MHz	1866 slices, 18 DSP
Performance versus cost on Virtex4, option 3, varying target frequency			
format	target f	performance	cost
(10,36)	200 MHz	6 cycles @ 203 MHz	874 slices, 9 DSP
	100 MHz	2 cycles @ 109 MHz	809 slices, 9 DSP
	50 MHz	0 cycles @ 51 MHz	751 slices, 9 DSP
(11,52)	200 MHz	7 cycles @ 187 MHz	1285 slices, 18 DSP
	100 MHz	3 cycles @ 102 MHz	1272 slices, 18 DSP
	50 MHz	2 cycles @ 64 MHz	1130 slices, 18 DSP
Portability to different FPGAs, , target frequency $f = 200$ MHz			
format	FPGA	performance	cost
(10,36)	Virtex 5	5 cycles @ 196 MHz	1444L, 762 R, 9 DSP48E
	Stratix II	8 cycles @ 179 MHz	1395L, 1295 R, 18 9-bit elem
	Stratix IV	4 cycles @ 213 MHz	1529L, 792 R., 18 9-bit elem

Format is given as (exponent size, significand size). We provide a reference as LogiCore operators assembled by hand. Option 1 is FPPipeline, using multipliers.
Option 2 is FPPipeline, using squarers. Option 3 is the fused datapath of of Figure 4.2. All these numbers were obtained in empty FPGAs using ISE 11.5 for Xilinx and QuartusII 9.1 for Altera.

precisions (SP and DP) took some minutes, however, each time the precision changes the operators need to be regenerated and the computation paths resynchronized. Assembling the operators using the `FPPipeline` meta-operator took a few seconds, both for the multiplier version (Option1) and for the squarer version(Option2). The design is parametrized in terms of exponent and fraction size, frequency and deployment FPGA. Obtaining the fused operator (Option3) took about two days to code together with the associated testbench. One can clearly see that the FloPoCo high productivity approaches match that obtained by using manufacturer specific core generators, and that the expert option (Option3) brings significant improvements over the assembling FP operators approach.

The second metric allowed to be explored using the FloPoCo framework is performance-cost one. As one requires a larger frequency from one operator, the area and latency of this operator increase.

Lastly, the FloPoCo framework allows optimizing the designed operators for different FPGAs, so that good performance is obtained for both Xilinx and Altera FPGAs.

The next sections will present in more details the features of the framework used to obtain these results.

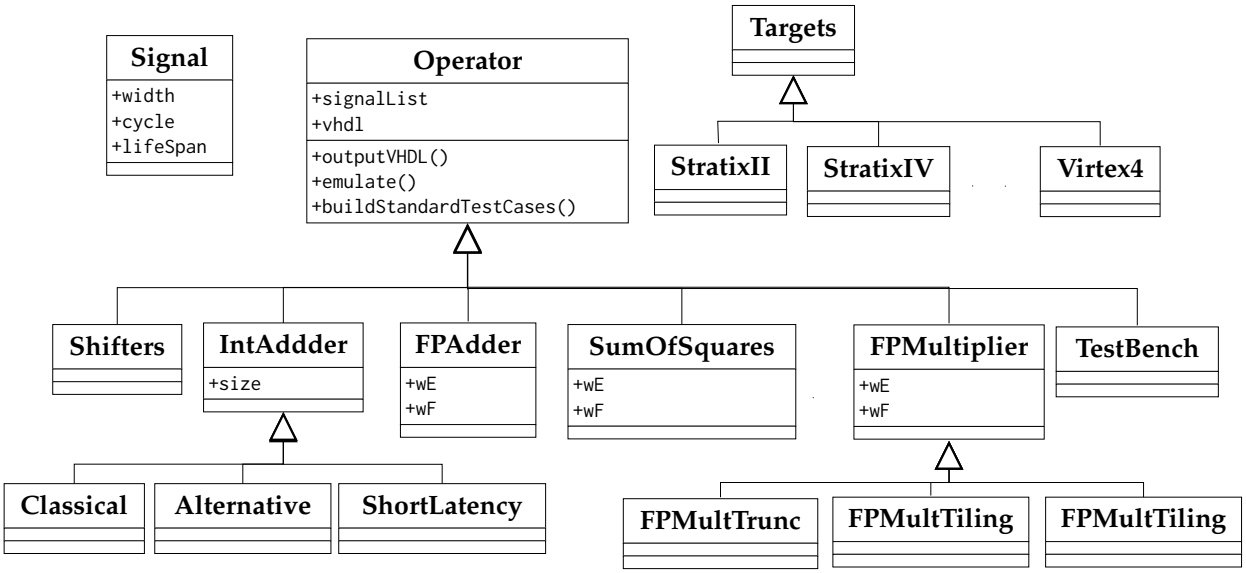


Figure 4.3 *Very simplified overview of the FloPoCo class hierarchy*

4.4 The FloPoCo framework

In the following, we assume basic knowledge of object-oriented concepts with the C++ terminology. Figure 4.3 provides a very simplified overview of the FloPoCo class hierarchy.

4.4.1 Operators

The core class is Operator. From the circuit point of view, an Operator corresponds to a VHDL entity, but again, with restrictions and extensions specific to the arithmetic context. All the operators extend this class, including SumOfSquares, but also some of its sub-components (shifters, squarers and adders) seen on Figure 4.2.

The main method of `Operator` is `outputVHDL()` whose purpose is to print out `VHDL` code in a standard C++ stream. Each operator inheriting from the `Operator` class can either override `outputVHDL()` to manually print VHDL in that stream (including signal and component declarations, library includes, etc.) or rely on the default implementation of this method provided in the `Operator` class (the standard way of using the framework). The default implementation takes the VHDL code of the operator architecture from the `vhdl` stream attribute of `Operator` together relevant information from other attributes (signal and component lists, lifespan and so on) and prints to the output stream the full VHDL code (entity, architecture, subcomponents, register management and the code from `vhdl`). The `vhdl` stream and other relevant attributes are populated during the execution of the arithmetic operator's constructor.

When printed to the `vhdl` stream, signals may be wrapped in several methods of the `Operator` class. A first method is `declare()` through which signals are declared. Consider the following code that computes the difference of two exponents of size `wE`, in order to determine in `XltY` which is smaller.

```
vhdl << declare("DEXY", wE+1) << " <= ('0' & EX) - ('0' & EY);" << endl;
vhdl << declare("XltY")       << " <= DEXY("<< wE <<");"        << endl;
```

Unlike VHDL, where signals are declared in the architecture's header and can be first used hundreds of lines of code away, FloPoCo supports inline declaration of signals: the signal as-

signment is also joined with its declaration. Here the `declare()` method adds the signal to the `signalList` of the operator so that the default implementation of `outputVHDL()` will automatically deal with its declaration in the operator's architecture header. The width of the signal is given as the second argument of `declare()`. If this argument is missing, the default signal size will default to 1. Aside from the background jobs the `declare()` method actually returns its first parameter, the signal's name.

The resulted VHDL code is presented in the Listing below:

```
(...)
signal DEXY: std_logic_vector(8 downto 0);
signal XltY: std_logic;
(...)
DEXY <= ('0' & EX) - ('0' & EY);
XltY <= DEXY(8);
```

The simple obfuscation of VHDL code using `declare()` allows automatically generating the signal declaration lists which account for about one third of the total lines of code in a classical VHDL architecture description. We will see next how this can also enable us to easily pipeline our designs.

4.4.2 Automatic pipeline management

Building a working pipeline for a given set of parameters is conceptually simple: for each operator synchronize its operands by inserting delay registers on the operand coming from the short datapath. FloPoCo allows to express exactly that in a generic way. Consider the following code describing the pipelined addition of three inputs:

```
addInput("X",k);
addInput("Y",k);
addInput("Z",k);
addOutput("R", k+1)

//current cycle = 0
vhdl<<declare("XpY", k+1)<< " <= (X"<<of(k-1)<<" & X) + (Y"<<of(k-1)<<" & Y);"<<end;
nextCycle(); //current cycle will be 1
vhdl<<declare("ZpXpY", k+1)<< " <= (Z"<<of(k-1)<<" & Z) + XpY;"<<end;
vhdl<<"R <= ZpXpY;"<<endl;
```

While sequentially printing the VHDL description of the circuit in the `vhdl` stream, a `currentCycle` attribute which denotes the current cycle in the description is maintained. The value of the `currentCycle` attribute is initially set to zero and can be changed by means of several methods: `setCycle(k)` sets its value to `k`, `nextCycle()` increments its value, `setCycleFromSignal("s")` sets its value cycle when signal `s` was declared, `syncCycleFromSignal("s")` advances the current cycle to the declaration cycle of signal `s` if this value is larger than the current cycle.

Every signal declared through `addInput()` or `declare()` has a associated a `cycle` attribute, which represents the cycle at which this signal is computed. It is `0` for the inputs (`X,Y,Z`) and is equal to `currentCycle` at the time `declare()` is invoked for signals declared through `declare()` (`cycle(XpY)=0`, `cycle(ZpXpY)=1`).

Every signal also possesses a `lifeSpan` attribute, useful for generating the correct number of register levels a signal needs to be delayed with. This attribute is initialized to `0` and holds the maximum number of cycles between the declaration of `s` and its uses. Therefore, for each right hand side occurrence of signal `s` the value `currentCycle-cycle("s")` is computed; if this value is larger than the signal's current `lifeSpan` it will become the new `lifeSpan`. In the case of signal `XpY` its `lifeSpan` is equal to `1`.

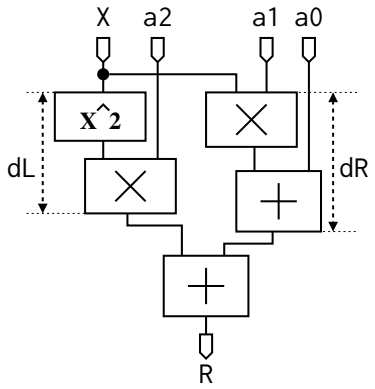


Figure 4.4 *Parallel evaluation of the polynomial* $a_2x^2 + a_1x + a_0$

When the `lifeSpan` of a signal `s` is greater than zero, outputVHDL() will create `lifeSpan` new signals, named `s_d1`, `s_d2` and so on, and insert registers between them. In other words, `s_d2` will hold the value of `s` delayed by 2 cycles. In the case of our simple example, the signals with `lifeSpan>0` are `XpY` but also `Z` with `lifeSpan=1`. The generated code will contain, for `k=8`:

```
(...)
signal XpY, XpY_d1: std_logic_vector(8 downto 0);
signal ZpXpY: std_logic_vector(8 downto 0);
signal Z_d1: std_logic_vector(7 downto 0);
(...)
  process(clk)
  begin
    if clk'event and clk='1' then
      if rst = '1' then
        XpY_d1 <= XpY;
        Z_d1   <= Z;
      else
        XpY_d1 <= (others => '0');
        Z_d1   <= (others => '0');
      end if;
    end if;
  end process;

XpY   <= (X(7) & X) + (Y(7) & Y);
-- entering cycle 1
ZpXpY <= (Z_d1(7) & Z_d1) + XpY_d1;
R     <= ZpXpY;
(...)
```

The impact of this simple technique of keeping track of signal lifespans has little overhead overhead in the generated VHDL (some signals are suffixed by d_XXX) but automates the generation of roughly one third of the lines of code of an architecture declaration.

4.4.3 Synchronization mechanisms

The type of synchronization we have talked about is valid for describing the execution of a single execution path. However, consider we would like to write the architecture which evaluates in parallel the second degree polynomial $a_2x^2 + a_1x + a_0$ using floating-point arithmetic by assembling library operators (Figure 4.4).

Suppose we are about to describe the final addition $a_2x^2+(a_1x + a_0)$. At this point we need to

ensure that the two signals entering the adder are synchronized. The pipeline depth of the internal components (squarer, adder and multipliers) depend on factors such as frequency, precision parameters and deployment target, making it impossible to say beforehand which of the two paths will be longer.

We denote by `dL` and `dR` the pipeline depths of the signals exiting the multiplier and the adder respectively (Figure 4.4). This yields three synchronization cases:

- `dL>dR` need to delay the `FPAdder`'s output with `dL-dR` cycles
- `dL=dR` two signals are already synchronized, nothing to do
- `dL<dR` need to delay the `FPMultiplier`'s output with `dR-dL` cycles

Managing these cases is trivial thanks to the `cycle` attribute associated with each signal declaration. The task of synchronizing the datapaths reduces to setting the value of the `currentCycle` to the maximum `cycle` value of all involved signals.

This could be tested by hand by examining the `cycle` value of each signal using the `getCycleFromSignal("s")` method. However, synchronizing datapaths is such a common task that FloPoCo provides a method to facilitate this: `syncCycleFromSignal("s")` advances the `currentCycle` to the declaration cycle of signal `s` if `cycle(s)>currentCycle`. All one has to do is then call this method for each of the signals which input the next computation. This is exactly what happens at lines 47-48 in Listing 4.1 which presents the synchronization of the two main computing paths from Figure 4.4.

At this point we know how to synchronize multiple computing paths. What we need is a way to start describing a new thread once the old thread's description is finished. We can apply the same mechanism: describing a new thread reduces to advancing `currentCycle` to the maximum `cycle` values of the signals involved in the first calculation of that thread.

Consider for example that we have finished describing one thread and `currentCycle=15`. If the inputs of the first computation of the new thread are also global inputs (therefore having `cycle=0`) one could just use `setCycle(0)` and then start the description. The situation now changes if these are not inputs and have cycle values `cycle(a)=dA` and `cycle(b)=dB`. Calling twice `syncCycleFromSignal()` for `a` and `b` will only yield the desired result if both `dA` and `dB` are larger than `currentCycle`. The way to go in this case is to first set `currentCycle` to either `dA` or `dB` using `setCycleFromSignal()` and then call `syncCycleFromSignal()` on the rest of the inputs. This technique is illustrated in Listing 4.1 at lines 27-28.

4.4.4 Managing subcomponents

FloPoCo provides the infrastructure for managing subcomponents. The code presented in Listing 4.1 makes extensive use of subcomponents. Subcomponents are instantiated just as regular C++ objects are – by calling their constructor. When using VHDL, before a subcomponent to be used its entity header needs to be added to the architecture's header as a subcomponent. The FloPoCo equivalent implies adding the subcomponent's object to the operator's subcomponent list, as Listing 4.1 lines 8-9, 18-19 and 38-39 illustrate. It literally takes two lines of code: one to instantiate the subcomponent (`c=new ...`) and one to add it to the component list (`oplist.push_back(c);`).

Port mapping is very similar to VHDL: `inPortMap(fpm,"X","X2");` maps the signal `X2` to the `fpm`'s `X` input port. The output port mapping is done in a similar way `outPortMap(fpm, "R", "a2x2");`. One major difference between the two commands is that `outPortMap()` also declares the signal denoted by the third argument (signal `a2x2` in our case) and sets its `cycle` attribute correspondingly.

The `instance()` function deals with writing all this information to the `vhdl` stream. One has to note that when instantiating subcomponents the `currentCycle` does not advance even though the output of the subcomponent may be at a later cycle than the current one. The option of advancing

the description cycle to the output of the subcomponent is left to the user by means of the function `syncCycleFromSignal()` (Listing 4.1 lines 15, 35, 47, 56).

```
1 int wE;
2 int wF;
3 addFPInput("X",wE,wF);
4 addFPInput("a2",wE,wF);
5 addFPInput("a1",wE,wF);
6 addFPInput("a0",wE,wF);
7
8 FPSquarer *fps = new FPSquarer(target, wE, wF);
9 oplist.push_back(fps);
10
11 inPortMap (fps, "X", "X");
12 outPortMap(fps, "R", "X2");
13 vhdl << instance(fps, "squarer");
14
15 syncCycleFromSignal("X2");// advance depth
16 nextCycle();//register level
17
18 FPMultiplier *fpm = new FPMultiplier(target,wE,wF);
19 oplist.push_back(fpm);
20
21 inPortMap (fpm, "X", "X2");
22 inPortMap (fpm, "Y", "a2");
23 outPortMap(fpm, "R", "a2x2");
24 vhdl << instance(fpm, "fpMuliplier_a2x2");
25
26 //describe the second thread
27 setCycleFromSignal ("a1");
28 syncCycleFromSignal("X");
29
30 inPortMap (fpm, "X", "X");
31 inPortMap (fpm, "Y", "a1");
32 outPortMap(fpm, "R", "a1x");
33 vhdl << instance(fpm, "fpMuliplier_a1x");
34
35 syncCycleFromSignal("a1x");// advance depth
36 nextCycle();//register level
37
38 FPAdder *fpa = new FPAdder(target, wE, wF);
39 oplist.push_back(fpa);
40
41 inPortMap (fpa, "X", "a1x");
42 inPortMap (fpa, "Y", "a0");
43 outPortMap(fpa, "R", "a1x_p_a0");
44 vhdl << instance(fpa, "fpAdder_a1x_p_a0");
45
46 //join the threads
47 syncCycleFromSignal("a1x_p_a0");//advance
48 syncCycleFromSignal("a2x2");//possibly advance
49 nextCycle();//register level
50
51 inPortMap (fpa, "X", "a2x2");
52 inPortMap (fpa, "Y", "a1x_p_a0");
53 outPortMap(fpa, "R", "a2x2_p_a1x_p_a0");
54 vhdl << instance(fpa,"fpAdder_a2x2_p_a1x_p_a0");
55
56 syncCycleFromSignal("a2x2_p_a1x_p_a0");
57 vhdl << "R <= a2x2_p_a1x_p_a0; " << endl;
```

Listing 4.1– FloPoCo parametric floating-point description for the circuit in Figure 4.4

4.4.5 Sub-cycle accurate data-path design

All basic FloPoCo operators have flexible pipelines, adapting to the user specified frequency, target FPGA and input/output precisions. By considering each operation atomic, we guarantee that its inputs come from registers and its output is registered. However, this is not optimal.

Consider again Listing 4.1, line 16 which will cause the insertion of a register at the output of `FPSquarer`. What if the squarer already has a registered output? Or if not, what if the current combinatorial path delay at its output is small enough so that some other calculations in `FPMultiplier` could be performed during the same cycle? If either answer true we are in a situation where we have over-pipelined our design. Globally this involves a possible latency increase and most of the time an area increase as well. What if the `nextCycle()` is removed? If there is some significant combinatorial logic on the squarer's output this could double the critical path delay, significantly affecting performance.

The problem in both cases is that `FPMultiplier` has no way to know what's the situation at the `FPSquarer`'s output, so we chose the conservative solution by calling `nextCycle()`, expecting the worst case.

A simple solution is to pass to `FPMultiplier` the combinatorial delays on its inputs as a parameter. Then, in its constructor `FPMultiplier` could explore if there are any more operations which could be performed in the remaining amount of time and insert a register level only if no other operations can be performed without exceeding the target delay ($1/f$). The solution involves replacing line 18 from Listing 4.1 with:

```
18 FPMultiplier *fpm = new FPMultiplier(target,wE,wF, inDelayMap("X",fps->getOutputDelay("R")));
```

Additionally, for such a system to be functional library operators should report their output delay which corresponds updating the `outDelayMap[]` map attribute.

4.4.6 Frequency-driven automatic pipelining

The previous section gave us the flavor of sub-cycle accurate pipelining in the context of assembling operators. This section generalizes the fine-grain pipelining technique to general VHDL code.

Consider that we now want to pipeline our datapath for a given frequency f. When done by hand, this task consists in identifying the critical path of the combinatorial circuit, then inserting enough synchronization barriers to split it into sub-paths, each of delay smaller than $1/f$.

In FloPoCo, code generation progresses from input to output, so the idea is to maintain an estimation of the current critical path delay, and insert synchronization barriers when needed. This is essentially what `manageCriticalPath()` does. This function takes as argument an estimation of the critical path delay of the logic generated by the `C++` code that follows it (up to the next `manageCriticalPath()`). It adds this argument to a variable `currentCriticalPath`, and if the resulting delay is larger than $1/f$, it inserts a synchronization barrier: it increments `currentCycle`, and resets the critical path delay to its argument.

In Listing 4.2 we have defined two atomic blocks that correspond respectively, on Figure 4.2, to the expSort box (lines 7 to 20), and to the two parallel subtraction boxes (lines 25 to 26). Depending on the target frequency, the code of Listing 4.2 will fuse these two blocks in a single cycle, or will insert a synchronization barrier between them.

The designer has the freedom to chose the granularity of these atomic boxes. This is a matter of expertise. Here, for instance, we know that we are subtracting exponents, which will therefore remain relatively small (even the 128-bit quadruple precision format has only `wE=16` exponent bits), so it make sense to consider the expSort box as atomic.

Incorporating subcomponents with this methodology is simple. Line 28 in Listing 4.2 shows how the following shifter whose input is `shiftB` would be instantiated to account for the input delay. The possible existing delay on `shiftB` is available via the `getCriticalPath()` method which returns the current value of the `criticalPathDelay`. Additionally, instantiated components should also set the value of the `criticalPathDelay` corresponding to the component's output (line 34).

4.4.7 The Target class hierarchy

The Target class abstracts the features of actual FPGA chips. Classes representing real FPGA chips extend this class – we currently have classes for very different FPGAs, Xilinx Virtex-4/5/6, Spartan3 and Altera StratixII-IV). The idea is to declare abstract methods in Target, which are implemented in its subclasses, so that the same generator code fits all the targets.

Of course, it is also possible to have a conditional statement that runs completely different code depending on the target - this will be the case for instance for the IntMultiplier class that builds large multipliers, because DSP capabilities are too variable from one target to the other.

A Target is given as argument to the constructor of any operator. The methods provided by the Target class can be semantically split into two categories:

– **Architecture-related methods** provide information about the architecture of the FPGA and are used in architectural exploration. For instance, `lutInputs()` returns the number of inputs of the FPGA's LUTs. This method is used by the Chapman's constant multiplication algorithm [51] implementation of FloPoCo.

```
1 // The expSort box
2 manageCriticalPath( // evaluate the delay
3   target->adderDelay(wE+1) // exp. diff.
4  + target->localWireDelay(wE) // wE is the fanout
5  + target->lutDelay() );   // MUX
6
7 // determine the max of the exponents
8 vhdl << declare("DEXY", wE+1) << " <= ('0' & EX) - ('0' & EY);" << endl;
9 vhdl << declare("DEYZ", wE+1) << " <= ('0' & EY) - ('0' & EZ);" << endl;
10 vhdl << declare("DEXZ", wE+1) << " <= ('0' & EX) - ('0' & EZ);" << endl;
11
12 vhdl << declare("XltY") << "<= DEXY(wE);" << endl;
13 vhdl << declare("YltZ") << "<= DEYZ(wE);" << endl;
14 vhdl << declare("XltZ") << "<= DEXZ(wE);" << endl;
15
16 // rename exponents to A,B,C with A>=(B,C)
17 vhdl << declare("EA", wE) << " <= EZ when (XltZ='1') and (YltZ='1') else "
18                           << "EY when (XltY='1') and (YltZ='0') else EX;" << endl;
19 vhdl << declare("EB", wE) << " <= " << (...);
20 vhdl << declare("EC", wE) << " <= " << (...)
21
22 // the parallel subtractions
23 manageCriticalPath( target->adderDelay(wE-1) );
24
25 vhdl << declare("shiftB", wE-1) << " <= (EA(wE-2 downto 0) - EB (wE-2 downto 0);";
26 vhdl << declare("shiftC", wE-1) << " <= (EA(wE-2 downto 0) - EC (wE-2 downto 0);";
27
28 Shifter *rightShifter=new Shifter(target,wF+g,wF+g,Shifter::Right,inDelayMap("S",getCriticalPath()));
29 (...)
30 inPortMap (rightShifter, "S", "shiftValB");
31 vhdl << instance(rightShifter, "ShifterForB");
32
33 syncCycleFromSignal("shiftedB"); //advance currentCycle to the shifter's output cycle
34 setCriticalPath( rightShifter->getOutputDelay("R"));
```

Listing 4.2 *Exponent difference and sorting in Figure 4.2*

– **Delay-related methods** provide approximative informations about computation time. For example, `adderDelay(int n)` returns the delay of an n-bit addition. Thus for instance, `adderDelay(16)` will return different values for a Spartan-3 or a Virtex-5, and eventually the pipeline will be deeper for a slower FPGA.

All the delays passed to `manageCriticalPath()` are evaluated thanks to methods of such `target` object. This object holds the current target FPGA (which can be specified by the `-target` option of the FloPoCo command line).

Modeling FPGAs is an endless effort, all the more as new models appear each year, but the reliance on the virtual Target class ensures that FloPoCo datapaths are designed in a reasonably future-proof way.

4.4.8 The bottom-line

The presented technique for pipeline management has many advantages:

– It is simple to implement, as it involves only comparisons and subtractions of integers.

– It clearly separates two very different issues: building a functional combinatorial datapath (on the left of Fig. 4.5), and pipelining it (on the right). From a combinatorial datapath, we are guaranteed to obtain a correctly synchronized pipeline with the same functionality,

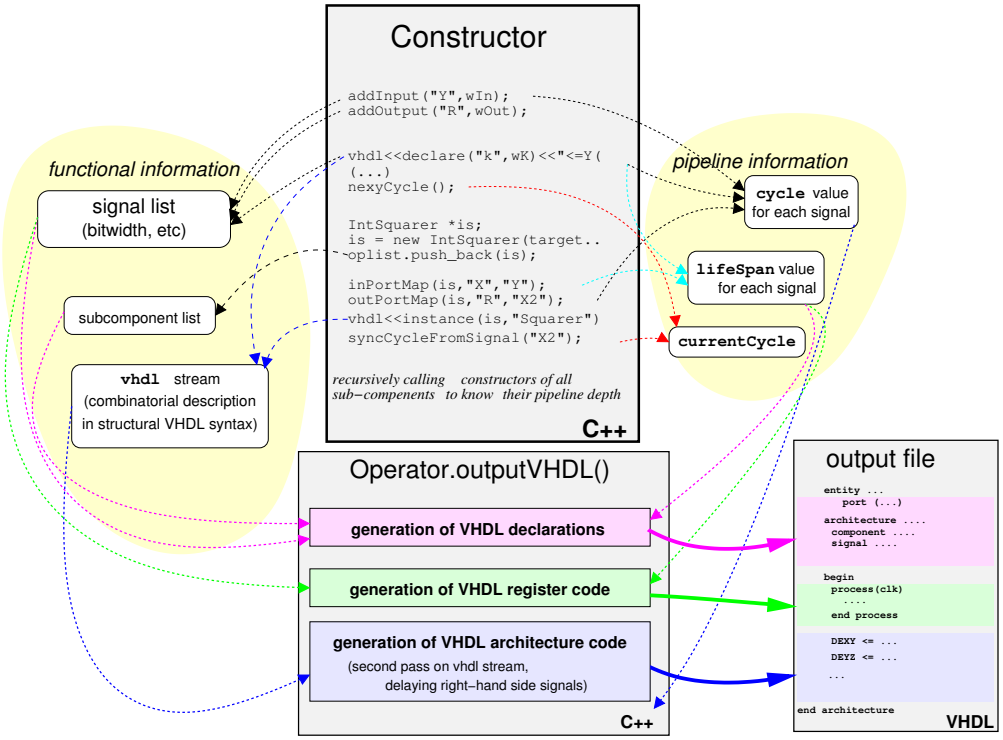


Figure 4.5 *Simplified overview of VHDL generation flow*

without touching any of the lines that define this datapath (the lines starting with `vhdl`).

– Its complexity is linear in the size of the generated code. Two passes are necessary: The first one writes the combinatorial VHDL code in the `vhdl` stream, and builds a dictionary of signals with their `cycle` and `lifeSpan`. The second one delays right-hand side signals.
– It adapts to arbitrary, dynamical placement of synchronization barriers, which is what we need for frequency-directed pipeline. It also gracefully degrades to a unpipelined, combinatorial implementation.
– The overhead of pipeline management in the generated code is minimal (some signal names posfixed by `_dxxx`), and this code remains as easy to read as the unpipelined version, especially compared to a pipeline of similar flexibility that would be written using VHDL `GENERATE` constructs.
– Finally, since this technique only involves post-processing signal names, it works for arbitrary VHDL.

4.4.9 Test-bench generation

Testing the designed arithmetic circuits is an essential feature of FloPoCo. Arithmetic circuit design starts with a mathematical specification which is then translated into an architecture that can be significantly different from the initial specification. Testing the implementation against its mathematical specification is not only simpler than mimicking the architecture, but it also minimizes the possibility of making the same mistake in both the operator's architecture and its test bench.

In FloPoCo, each operator can be associated with an `emulate()` method. Its purpose is to describe the operator's functionality starting from its mathematical specification. The method receives a set of inputs and returns the associated output for those inputs. For testing basic inte-

```
1 void FPExp::emulate(TestCase * tc)
2 {
3     /* Get I/O values */
4     mpz_class svX = tc->getInputValue("X");
5
6     /* Compute correct value */
7     FPNumber fpx(wE, wF);
8     fpx = svX;
9
10    mpfr_t x, ru,rd;
11    mpfr_init2(x, 1+wF);
12    mpfr_init2(ru, 1+wF);
13    mpfr_init2(rd, 1+wF);
14    fpx.getMPFR(x);
15    mpfr_exp(rd, x, GMP_RNDD);
16    mpfr_exp(ru, x, GMP_RNDU);
17    FPNumber fprd(wE, wF, rd);
18    FPNumber fpru(wE, wF, ru);
19    mpz_class svRD = fprd.getSignalValue();
20    mpz_class svRU = fpru.getSignalValue();
21    tc->addExpectedOutput("R", svRD);
22    tc->addExpectedOutput("R", svRU);
23    mpfr_clears(x, ru, rd, NULL);
24 }
```

Listing 4.3 `emulate()` *for* e^x

```
1 TestCase* FPExp::buildRandomTestCase(int i){
2     TestCase *tc;
3     tc = new TestCase(this);
4     mpz_class x;
5     mpz_class normalExn = mpz_class(1)<<(wE+wF+1);
6     mpz_class bias = ((1<<(wE-1))-1);
7     /* Fill inputs */
8     if ((i & 7) == 0) { //fully random
9         x = getLargeRandom(wE+wF+3);
10    }else{
11        mpz_class e = (getLargeRandom(wE+wF)
12                       %(wE+wF+2))-wF-3;
13        e = bias + e;
14        mpz_class sign = getLargeRandom(1);
15        x  = getLargeRandom(wF)
16                  + (e << wF)
17                  + (sign<<(wE+wF))
18                  + normalExn;
19    }
20    tc->addInput("X", x);
21    /* Get correct outputs */
22    emulate(tc);
23    return tc;
24 }
```

Listing 4.4 *Specialized test-cases for* e^x

ger arithmetic operators such as high-precision adders, multipliers but also shifters, leading-zero counters etc. we use the GNU Multiprecision (GMP) library [2]. It allows manipulating large integers without worrying about overflow. When implementing `emulate()` for variable-precision both fixed and floating-point operators we use the MPFR multi-precision floating-point library [4].

Listing 4.3 presents the code of the corresponding `emulate()` function for the faithful floating-point exponential operator. For one input value x this operator allows two valid outputs (the two floating-point numbers closer to e^x). The random value for x is received via the `tc` (test case) parameter (line 4). This value (from a random stream of bits) is then converted to a `mpfr` floating-point variable (line 14). Next, e^x is computed with infinite precision and then rounded towards $-\infty$ (line 15) and towards $+\infty$ (line 16). The two output floating-point values are converted back to bit-streams (lines 19,20) and then returned in the same `tc` (lines 21-22) parameter. The code of this function is simple and is much less error prone than the designed architecture for the floating-point exponential operator (see Chapter 9, Figure 9.3).

Test-bench suites, consisting of a user-defined number of test-cases (`tc`), can be generated for all operators having and `emulate()` function. As exhaustive testing is not usually possible, the problem boils down to choosing the test-vectors which best test the given operator.

For some operators such as fixed-point $+$, $\times$, floating-point $\times$, the test-vectors can be generated using the classical random-number generators. The probability of testing all the data-paths of the circuit suffices. Other floating-point operations are more sensitive:

- $+$. The architecture usually consists of two main data-paths, one for the case when the difference in exponents is $\in \{-1, 0, 1\}$. The probability of generating a test-vector which tests this data-path using an random-number generator with a uniform distribution is approximatively $1/170$ for single-precision and $1/1365$ for double-precision.
- e^x. The exponential returns zero for input numbers smaller than $\log(2^{(2^{w_F-1}-1)})$, and should return $+\inf$ for all inputs larger than $\log((2-2^{-w_F})\cdot 2^{2^{w_E-1}-(2^{w_F-1}-1)})$. In single precision the set of input numbers on which a computation will take place is just [-88.03, 88.72]. In

addition, as for small x we have $e^x \approx 1 + x + x^2/2$, the exponential will return 1 for all the input x smaller that 2^{-w_F-2}. One consequence is that the testing of a floating-point exponential operator should focus on the this range of the input. More details can be found in chapter 9.

FloPoCo offers the possibility of overriding the default behavior of the test-bench generation suite which fills test-cases using random-numbers having a uniform distribution. The new function which generates the random test-cases for e^x if given in Listing 4.4. This new version of the random generator function generates $1/8$ truly random inputs, and the rest of $7/8$ tests generate inputs where the exponent is in the range $x \in [X_{\min}, X_{\max}]$, where $X_{\min} = 2^{-E_0}$ and $X_{\max} = (2 - 2^{-w_F}) \cdot 2^{2^{w_E}-1-E_0}$.

For the case of the floating-point addition one could decide that testing the two data-paths with the same probability suffices. Implementing this change is trivial, but might not be enough. Consider the extreme case $X + (-X)$. This causes a massive cancellation of the mantissas and is therefore a difficult case to cover. Probabilistically, this has a $1/2^{w_F}$ chances of happening with a uniform distribution. In order to capture all these corner-cases, FloPoCo allows manually defining a set of standard test-cases which make it possible specify the extreme cases. The standard test-cases for floating-point addition are presented in Listing 4.5.

4.4.10 Framework extensions

Managing feedback loops

Up to this point we have constrained our definition of arithmetic operator to functions. In fact, the current implementation of FloPoCo can also manage feedback loops. This is especially important as the accumulation [1] circuit which falls in this category is considered by many *the 5th basic operation*. The subtlety in this case is using a signal which may be declared cycles later. Say for example that the accumulation circuit takes has 5 pipeline stages. The result signal of this accumulation is declared only at cycle 5 in the design, however, it needs to be fed back to the first cycle, at the accumulator's input. From the framework's point of view there's no problem with this: the lifespan computation does not insert any registers as the signal is used at an earlier cycle. However, using a signal cycles before it's declared leads to errors in designs not having feedback loops. Consequently, at circuit generation, our framework signals as a *warning* the signals having this property. If indeed the signals are feedback signal this may be ignored; otherwise, the described circuit may not be what the user planed-for.

An extension for VLSI ALU design

The initial purpose of FloPoCo was to provide a flexible environment for describing purely arithmetic operators for FPGAs. Nevertheless, FloPoCo may be extended to be used in VLSI ALU design. The extension is in fact a simplification of all basic components for the VLSI target. The VHDL code generated for the basic operators will simply be "+,-,*".

FloPoCo will be used perform and initial pipelining of the ALU. The code generated will then be passed through VLSI specific tools which replace the "+,-,*" operators by VLSI-specific instantiations and perform register retiming.

Backend for HLS

Our framework can also be used as a back-end for high-level synthesis as it offers an important basis of arithmetic operators optimized for different types of contexts. The tool itself is open-

1. [63] presents a detailed implementation specific to FPGAs

```
1 void FPAdder::buildStandardTestCases(TestCaseList* tcl){
2         TestCase *tc;
3 
4         // Regression tests
5         tc = new TestCase(this);
6         tc->addFPInput("X", 1.0);
7         tc->addFPInput("Y", -1.0);
8         emulate(tc);
9         tcl->add(tc);
10 
11        tc = new TestCase(this);
12        tc->addFPInput("X", 1.0);
13        tc->addFPInput("Y", FPNumber::plusDirtyZero);
14        emulate(tc);
15        tcl->add(tc);
16 
17        tc = new TestCase(this);
18        tc->addFPInput("X", 1.0);
19        tc->addFPInput("Y", FPNumber::minusDirtyZero);
20        emulate(tc);
21        tcl->add(tc);
22 
23        tc = new TestCase(this);
24        tc->addFPInput("X", FPNumber::plusInfty);
25        tc->addFPInput("Y", FPNumber::minusInfty);
26        emulate(tc);
27        tcl->add(tc);
28 
29        tc = new TestCase(this);
30        tc->addFPInput("X", FPNumber::plusInfty);
31        tc->addFPInput("Y", FPNumber::plusInfty);
32        emulate(tc);
33        tcl->add(tc);
34 
35        tc = new TestCase(this);
36        tc->addFPInput("X", FPNumber::minusInfty);
37        tc->addFPInput("Y", FPNumber::minusInfty);
38        emulate(tc);
39        tcl->add(tc);
40 }
```

Listing 4.5 *Standard test-cases for floating-point addition*

source and extensible allowing an on-demand update of the available operator basis. This is as flexible as being able to add a new instruction to the instruction set of a microprocessor. Work is undergoing in experimenting in this research direction at ENS de Lyon and in several other places.

4.5 Conclusion

The FloPoCo framework improves the productivity of designing flexible and efficient arithmetic datapaths for FPGAs. It offers designers some unique features: state-of-the-art arithmetic operator library, a novel methodology for the generation of correct-by-construction pipelines matching a given frequency on a given FPGA target, and arithmetic-oriented test-bench generation.

Combined, these features provide a competitive solution both for novice users building computational pipelines by assembling operators, but also for experienced users who need full control over the entire design process. The next chapters will present some of the flexible operators we

have implemented using FloPoCo.

The long-term plan is to use FloPoCo for ever coarser datapaths such as signal-processing filters. We also hope it will be used as a back-end for high-level synthesis tools. Both the library and the framework will be developed to address the needs of these application fields. Potential future work also includes adding to the framework resource estimation, floorplaning support, fixed-point support, support of sequential circuits, and ASIC targets.

Thanks

I would like to kindly thanks all the FloPoCo developers for their respective contributions: Cristian Klein for integrating Jérémie Detrey's HOTBM generator in the early stages of the project, and also for his initial work on the testing infrastructure; Nicolas Brunie which has improved and extended the testing framework; Mioara Joldes, one of the main developers of Sollya, for her work on developing the polynomial table generator, one of the key features of FloPoCo; Sebastian Banescu and Radu Tudoran for their work on the FloPoCo multipliers; Sylvain Collange for contributing with the LNS operators; Alvaro Vasquez for contributing with his decimal operators.

Chapter 5

Binary addition in FloPoCo

Integer addition is used as a building block in many coarser operators. Examples which require large adders include integer multipliers, most floating-point operators, and modular adders used in some cryptographic applications. In floating-point, the demand in precision is now moving from double (64-bit) to the recently standardized quadruple precision (128-bit format, including 112 bits for the significand) [17]. In elliptic-curve cryptography, the size of modular additions is currently above 150 bits for acceptable security.

This chapter presents the binary addition operator generator part of FloPoCo. When the FloPoCo project was initiated, it was not expected that we would need to dedicate so much work to something as seemingly simple as integer addition on FPGAs. The reason why it became important is that addition is so pervasive. The presented adder generator provides subcomponents for integer multipliers and constant multipliers, and for most floating-point cores, including addition, multiplication, division and square root, and elementary functions. If we want these cores to work at a high frequency for double precision and beyond, we need high-performance adders, but we also need them to consume as little resources as possible. Therefore, the adder generation described here is frequency-driven (possibly inheriting the frequency from the wider context) and minimizes resource consumption, based on accurate resource estimation formulas of the architectures.

Adders differ in the way they propagate carries. Modern FPGAs include special hardware dedicated to carry propagation [12, 19, 18, 23, 14, 14, 22, 27]. Sending a carry to a neighboring cell through the dedicated carry line is much faster than sending a bit to the same cell through the general reconfigurable routing fabric. Therefore, proven solutions for VLSI designs like carry look-ahead or prefix adder trees [81] bring little speed improvement on FPGAs over the ripple carry adder (RCA) except for very wide addition sizes [159]. These speed improvements are small, and they come at a cost penalty exceeding a factor 2 over the RCA. Therefore, a binary addition is expressed in VHDL as a + and is implemented by default as an RCA.

In this chapter we re-evaluate this situation when a *pipelined* adder is needed but also propose several *short-latency* adder architectures as alternatives to the deeply-pipelined RCA in the case of wide adders. We restrict our discussion the architectures which can be described using *portable* VHDL as we believe this will make our core-generator more future-proof.

5.1 Related work

The simplest pipelining of binary addition [151, 58, 81] consists in buffering the carry-out of each full-adder (FA) along the carry propagation path, and inserting synchronization registers for

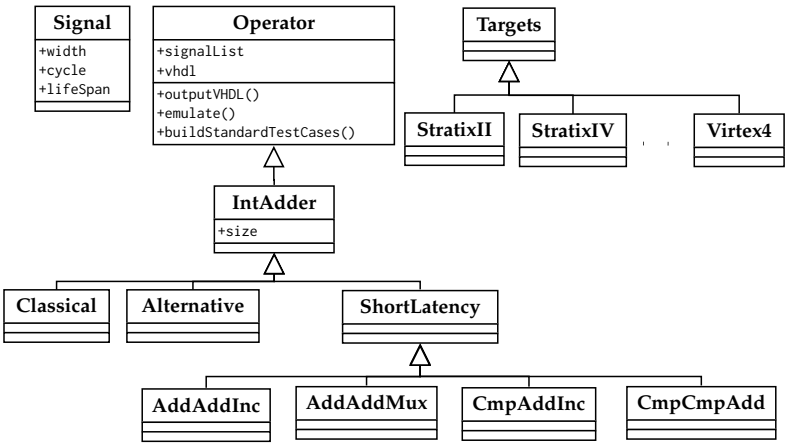


Figure 5.1 *FloPoCo class structure for binary addition*

I/O. This technique is wasteful when the objective period is larger than the delay of a 1-bit carry propagation. For these cases, a better version [120, 81, 42] consists in registering carries only every α FA cells. This technique will be detailed in section 5.3.1, and is referred to as the *classical* RCA pipelining technique.

Faster techniques than the previous classical architecture have been developed for VLSI. A first idea is to speed up the logic on the carry propagation path [122, 58]. Other, more algorithmic approaches include carry-select, carry-skip, and the family of prefix adders [81]. These designs map poorly on FPGAs, however they have served as an initial source of inspiration for the proposed alternative pipelining technique from section 5.3.3.

An initial study evaluating the performance of fast addition schemes on FPGAs is presented by Xing and Yu [159] back in 1998. The study concludes that among the numerous fast addition schemes, the only ones mapping reasonably well to FPGAs are carry-skip and the carry-select, the latter providing the best performances. The optimizations applied by Xing and Yu to the classical carry-select architectures are structural, speculative carry-bit computations being addressed by carry-skip structures. The carry-in computation for each carry-select block is done using the classical multiplexer network, which is slow in FPGAs.

A discussion on the synthesis of carry-select adders in modern FPGAs is presented by Naik and Shah [125]. The study proposes bitwise computation of the speculative sums using XOR gates and an inverters. The impact of these optimizations in modern FPGAs is little, if any, as presented in section 5.6.2.

Another variation of the carry-select architecture is presented by Devi et al. [75]. It is based on the idea of time-multiplexing the same adder resource for computing the two speculative sums and carry-bits. The design manages to reduce the area at the expense of latency. Its implementation requires low-level directives for mapping the circuit to hardware, thus lacking portability. The results are presented for a maximum addition size of only 32bits which makes it impossible to compare against.

In this chapter we provide several *efficient mappings* of the carry-select addition architecture in modern FPGAs. For wide enough additions, the proposed adder family can consume less resources than the pipelined RCA schemes, while having an unpipelined architecture. The presented adder architectures are part of the FloPoCo class hierarchy as Figure 5.1 presents.

5.2 Design-space exploration by resource estimation

Modern FPGA resources are heterogeneous, including LUT-based logic, embedded memories, embedded DSP blocks, and others. Pipelined adders generally require logic and registers. Several chained registers, often encountered in pipelined designs form shift-register. Shift-registers can be easily implemented by chaining the registers available in the SLICE/ALM of the device. However, this technique is inefficient for implementing deep shift-registers.

Modern Xilinx FPGAs have been enhanced with hardware support for shift-registers: the LUTs in SLICE_Ms can be configured as variable length shift-registers (up to 16 levels for Virtex-4 and 32 levels for Virtex-5/-6). Counting LUTs and registers will suffice for adder resource estimation on these devices, either if the shift-registers will be implemented using LUTs in the SRL configuration, or just using regular registers.

In the case of Altera devices the available embedded memories (Altera devices have 3 degrees of granularity, see Section 2.1.3 for more details) provide hardware support for shift-register implementation. Their granularity is slightly larger than the SLICE_M of Xilinx (see section 2.1.3) which restricts their efficient usage in this context to longer register chains (i.e. wider additions). When embedded memories are used to implement shift-registers one may need to also count these, alongside with LUTs and registers for adder resource estimation.

The default behavior of the Altera QuartusII synthesis tool is reluctant in assigning the precious embedded memories for shift-register implementations. It roughly requires seven levels of registers for StratixIII devices for such a shift-register to start using embedded memories. In the case of adders, this number of pipeline levels is usually associated with very wide adders, for which we will provide specialized low-latency architectures in Section 5.4. Therefore, we have decided to count LUTs and registers on these devices as well, although clearly the synthesis options may prove that some of the registers will be implemented using embedded memories.

The resource estimation formulas for each architecture allow choosing the best adder architecture for a given situation on-the-fly. Once the decision is made, the VHDL code of the adder can be generated and synthesized. It is obvious that in order for this method to provide the expected results, the estimation formulas must effectively predict the performance and resource consumption of the operator after synthesis and technology mapping. The results presented in Section 5.6.1 will validate this assumption, proving that in practice, these formulas are accurate to 1-3% in all cases.

5.3 Pipelined addition on FPGA

Let X, Y be two integers representable on w bits either in unsigned representation or in 2's complement. These numbers are either zero/sign extended to $w+1$ bits in order to absorb the possible overflow. This is the usual technique used for addition in FPGAs:

$$S \leftarrow \text{signExtend}(X, w+1) + \text{signExtend}(Y, w+1) + c_{in}.$$

There are numerous addition architectures that compute this sum. The most popular in the FPGA context is the Ripple-Carry Adder (RCA). A w-bit RCA with a carry-in is composed of $w+1$ chained Full-Adders (FAs) [1] [2], as presented in Figure 5.2. The Full-Adder (FA) equations are:

1. in Xilinx devices it is possible to intercept the carry-out bit of the $S(w-1)$ cell; however, this has no associated register and is of little use in our context
2. in Altera devices the carry-in bit requires a supplementary FA with one zero input for introduction in the carry-chain

$$s = a \oplus b \oplus c_{\text{in}}$$
$$c_{\text{out}} = a \cdot b + c_{\text{in}} \cdot (a \oplus b)$$

These equations can also be written using the classical *generate*, *propagate* signals:

$$s = p \oplus c_{\text{in}}$$
$$c_{\text{out}} = g + c_{\text{in}} \cdot p, \text{ with } p = a \oplus b \text{ and } g = a \cdot b$$

or for Xilinx FPGAs

$$c_{\text{out}} = \overline{p} \cdot a + c_{\text{in}} \cdot p$$

The RCA adder delay is proportional to the addition size w. It generally has three components:
– the delay to compute the *generate* and *propagate* signals δ_{p}
– the *gate-delay* for carry-out bit propagation δ_{c}
– the delay of computing the most significant sum bit (equation 2.1) δ_{s}.
The worst case delay for the RCA is:

$$\delta_{\text{S(w)}} = \delta_{\text{p}} + (w-1)\delta_{\text{c}} + \delta_{\text{s}} \tag{5.1}$$

In Xilinx devices, these delays directly map to the architecture: (1) the LUT is used to compute the propagate delay so $\delta_{\text{p}} = \delta_{\text{LUT}}$ (2) the $\delta_{\text{c}} = \delta_{\text{MUXCY}}$ and (3) the sum delay is $\delta_{\text{s}} = \delta_{\text{XORCY}}$.

Altera devices provide dedicated FAs in the ALMs. According to the delay-informations extracted using Chip-Planner, the hardware FA is implemented using the *generate-propagate* equations, and thus has different delays for carry-out bit and sum-bit computations. Due to an elaborated scheme allowing the summation of 3 inputs in one LUT level, the FA inputs pas through the LUT logic. Consequently, for these devices $\delta_{\text{p}} = \delta_{\text{LUT}} + \tilde{\delta_{\text{p}}}$. Due to the ASIC-like nature of the FA, the delay for computing p is much smaller than the LUT delay, allowing for a reasonable estimation $\delta_{\text{p}} = \delta_{\text{LUT}}$. The sum bit computation is also very fast, however, the signal needs to pass through the ALM's output multiplexer network $\delta_{\text{s}} = \tilde{\delta_{\text{s}}} + \delta_{\text{outMUX}}$.

The major difference to Xilinx FPGAs is the estimation of the carry-delay which is not fixed. This is due to the fact inter LAB transitions and mid-LAB carry amplifiers introduce larger delays. Consequently, the worst-case delay equation in the case of Altera devices is:

$$\delta_{\text{S(w)}} = \underbrace{\delta_{\text{LUT}} + \tilde{\delta_{\text{p}}}}_{\delta_{\text{p}}} + \underbrace{\frac{w}{2w_{\text{LAB}}}}_{i}\delta_{\text{iLAB}} + \underbrace{\frac{w}{w_{\text{LAB}}}}_{b}\delta_{\text{buf}} + (w-i-b)\delta_{\text{c}} + \underbrace{\tilde{\delta_{\text{s}}} + \delta_{\text{outMUX}}}_{\delta_{\text{s}}} \tag{5.2}$$

As w increases the addition frequency decreases as illustrated in Figure 5.3 for three FPGAs.

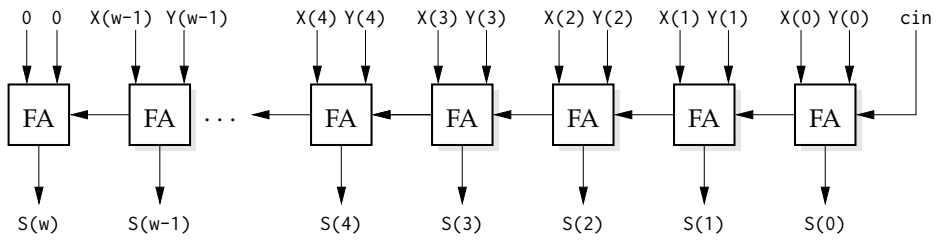


Figure 5.2 *Ripple-Carry Adder implementation*

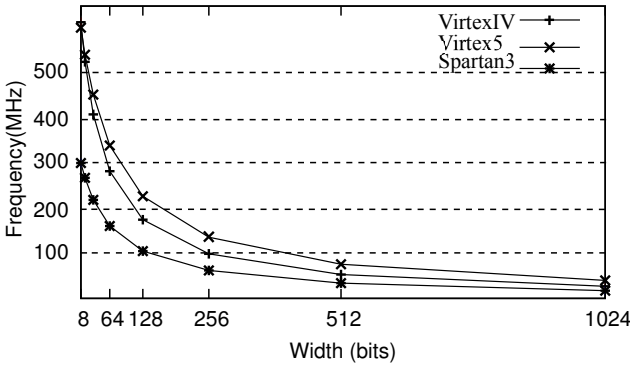


Figure 5.3 *Ripple-Carry Addition Frequency for VirtexIV, Virtex5 and Spartan3E*

In the context of frequency-driven pipelining, a pair (w, f) which is under the corresponding curve in Figure 5.3 meets the frequency constraint. There are two solutions for additions not meeting this constraint: (1) Pipeline the adder design such that the critical path of the circuit is less than the target period $T = 1/f$ or (2) We can choose a different addition architecture that is able to reach the frequency without too much of a cost penalty; such architectures will be discussed in Section 5.4. In this chapter we will focus on developing the best solutions for both alternatives. Then, based on preliminary resource estimation values, the best architecture will be chosen for a given context.

5.3.1 Classical RCA pipelining

A tight frequency-driven pipelining is obtained by first determining the maximal addition size α in equation 5.1 for which the critical path delay is less than the target period T (finding α for Altera devices is similarly done by solving equation 5.2):

$$\alpha = 1 + \left\lfloor \frac{T - \delta_{\mathrm{p}} - \delta_{\mathrm{s}}}{\delta_{\mathrm{c}}} \right\rfloor \quad .$$

Next, the addition is split into k chunks of α bits (except the last chunk denoted by β, $\beta \leq \alpha$) such that $w = (k-1)\alpha + \beta$.

An instantiation of this architecture highlighting the previously discussed parameters is presented in Figure 5.4 for $k = 4$. As k decreases, the number of registers used for synchronization decreases. When the critical path of the w-bit addition is $\leq T$, no pipelining is required ($k = 1$) and the addition may be expressed as a simple $+$ in VHDL.

The column labeled Classical in Table 5.1 presents the resource estimation formulas function of α, β, w, k, respectively with and without allowing shift-register packing in LUTs (SRL). Let us now explain how such formulas were built.

5.3.2 Resource estimation techniques

Let us take as a running example the previous classical architecture, annotated on Figure 5.5.

The LUTs of the Xilinx FPGAs can be be used either as a function generator, or as a variable length shift-register, as previously presented in Section 5.2.

For classical architecture, the addition diagonal uses w LUTs configured as function generators (Figure 5.5, σ). The LUT SRL configuration is used when two or more flip-flops are cascaded to form a shift register, if one of the two does not immediately follow one LUT. This is the case of the $(k-3)\alpha$ SRLs under the addition diagonal (Figure 5.5, ξ), together with the 2β SRLs corresponding

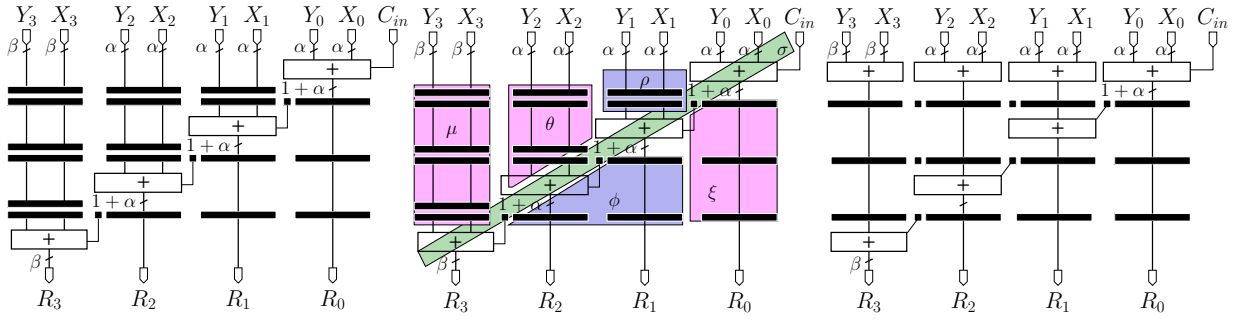


Figure 5.4 *Classical addition architecture [81]*

Figure 5.5 *Annotated classical architecture*

Figure 5.6 *Proposed FPGA architecture*

to the last column of width β (Figure 5.5, μ) and of the $2(k-3)\alpha$ SRLs above the diagonal (Figure 5.5, θ). In addition, one also has to count the $k-1$ extra LUTs needed to extend the α additions by one bit in order to buffer the carry-out. These sum up to $w + (3k-9)\alpha + 2\beta + k - 1 = (4k-10)\alpha + 3\beta + k - 1$, which is the value reported in Table 5.1.

There is one consideration to be made before counting registers: each time an SRL is used, the corresponding slice flip-flop is also used. In other words, for a p-level shift-register, $p-1$ levels are pushed into the SRL and one into the flip-flop. Hence, we count $(3k-9)\alpha + 2\beta$ registers for the same number of SRL, and, in addition, 3α registers under the diagonal (Figure 5.5, ϕ), 2α registers above the diagonal (Figure 5.5, ρ) plus the $k-1$ registers for the carry-bit propagation. These total $(3k-4)\alpha + 2\beta + k - 1$, the value reported in Table 5.1.

The next task is to count elementary LUT-FF pairs which correspond to half-slices for Virtex-4, quarter-slices for Virtex-5/-6 and half-ALMs in Altera FPGAs. This corresponds to a dense placement of the pipelined adder, which the tools are expected to favor. Experimental results given in Section 5.6.1 will validate this assumption.

The number of LUT-FF pairs used by the classical implementation is: w for the diagonal addition, $(3k-9)\alpha + 2\beta$ for the SRL and corresponding flip-flops, and $5\alpha + k - 1$ for the independent registers. However, we subtract 2α as the left-most 2 additions of α bits include the registers in the same pair with the LUT. The number totals $(4k-7)\alpha + 3\beta + k - 1$, which is reported in Table 5.1.

All the formulas presented here were deduced using these techniques. Relative errors of these estimation formulas are given in Table 5.6. The worst case relative error is of the order of one percent which makes them sufficiently accurate for estimation formulas.

5.3.3 Alternative RCA pipelining

The classical pipelining technique requires a significant amount of registers for input synchronization. This number may be lowered by performing the chunk additions at the first pipeline level and then propagating these sums instead. When no SRL are allowed, the number of registers propagated above the diagonal will be approximatively halved, and may still be packed in shift registers. An instantiation of this architecture for $k = 4$ is presented in Figure 5.6.

Each adder on the addition diagonal takes as input an operand on $\alpha + 1$ bits and a 1-bit carry in and returns a $\alpha + 1$-bit wide result. This addition does not overflow, as the $\alpha + 1$-bit input was the result of an addition of two α-bit numbers with a carry-in of 0.

The resource estimation formulas for this architecture are presented in Table 5.1.

Table 5.1 *Resource estimation formulas for the pipelined adder architectures with shift-register extraction (SRL) (Xilinx only) and without SRL (Xilinx and Altera)*

		Classical	Alternative
SRL	**REG**	$\begin{cases} \alpha + 2\beta & : k = 2 \\ (4k-7)\alpha + 2\beta + k - 1 & : k \geq 3 \end{cases}$	$\begin{cases} (k-1)w + (k-1)k/2 & : k \leq 3 \\ (2k-2)\alpha + b\beta + (k-1)k/2 & : k \geq 4 \end{cases}$
	LUT	$\begin{cases} \alpha + \beta & : k = 2 \\ (4k-10)\alpha + 3\beta + k - 1 & : k \geq 3 \end{cases}$	$\begin{cases} (k-1)w - \alpha - (k-1)k/2 & : k \leq 3 \\ (4k-10)\alpha + 3\beta + 2k - 1 & : k \geq 4 \end{cases}$
	LUT-FF	$(4k-7)\alpha + 3\beta + k - 1$	$\begin{cases} (k-1)w + \beta + (k-1)k/2 & : k \leq 3 \\ (4k-8)\alpha + 3\beta + 2(k-2) & : k \geq 4 \end{cases}$
No SRL	**REG**	$\frac{3k^2-7k+4}{2}\alpha + 2(k-1)\beta + k - 1$	$(k-1)w + k^2 - 2k + 1$
	LUT	$w + k - 1$	$2w - \alpha + 2k - 3$
	LUT-FF	$w + \frac{3(k^2-3k+2)}{2}\alpha + 2(k-1)\beta + k - 1$	$(k-1)w + \beta + k^2 - 2k + 1$

5.3.4 Area-complexity of the pipelined designs

Table 5.1 presents the resource estimation formulas for LUT, Register and LUT-FF costs for both the Classical and Alternative architectures in the case when the inputs arrive from a register, but there is a clear isolation between hierarchy boundaries: in such case the chunk-splitting strategies are similar for both architectures.

The formulas in the top part of the table (SRL) are valid only for Xilinx devices. The bottom part of the table presents the estimation formulas for the case no shift-registers are extracted, and are valid for both Xilinx and Altera (see Section 5.2 for a discussion). These formulas have been experimentally validated in Section 5.6.2.

A close analysis of these formulas reveals that the alternative architecture slightly outperforms the classical one in terms of LUT-FF pairs. One might then argue that there is no design-space exploration to perform, and one should use the alternative architecture whenever the LUT-FF metric is targeted.

However, in larger designs, softening the hierarchy boundaries allows cross-boundary shift-register inference which significantly reduces component cost. This is the case of the Classical architecture when some of its inputs arrive from a shift-register register level. Work is undergoing in order to evaluate the possibility of integrating this feature in the FloPoCo framework.

Table 5.2 presents the corresponding estimation formulas for the Classical architecture when there exist combinatorial delays on the inputs. In such a case, the size of the first addition, now denoted by $\gamma \leq \alpha$, has to be reduced in order for this addition to meet the frequency target.

The impact of this scenario in the case of the Alternative architecture is that all input chunks have to be reduced to size γ ($\alpha = \gamma$ for the formulas in Table 5.1 and $\beta = \beta'$ with $\beta' \leq \alpha$). This has the potential to increase the number of chunks we need to split the addition in, therefore affecting the resource usage, possibly making the Classical architecture more attractive.

The bottom line is that, in order to properly integrate our adder architectures in the FloPoCo framework, and automatically select best architecture for a context, we need to evaluate the cost of all these architectures.

5.4 Short-latency addition architecture

Given a target frequency f, the pipeline depth of the previously presented architectures increases linearly with addition size. In this section we propose a scalable low-latency addition architecture based on the textbook carry-select architecture, whose novel feature is to make efficient use of the fast-carry chains for the carry-bit computations.

Table 5.2 *Advanced resource estimation formulas for the pipelined classical architecture, when shift-register extraction is activated*

Input delays	**REG**	$\gamma + 2\alpha + 1 \quad : k = 2$ $2\gamma + 3\alpha + 2\beta + 2 \quad : k = 3$ $2\gamma + (4k-9)\alpha + 2\beta + k - 1 \quad : k \geq 4$
	LUT	$\gamma + \alpha + 1 \quad : k = 2$ $w + 2\beta + 2 \quad : k = 3$ $w + \gamma + (3k-10)\alpha + 2\beta + k - 1 \quad : k \geq 4$
	LUT-FF	$\gamma + 3\alpha + 1 \quad : k = 2$ $w + 2\beta + \gamma + 2 \quad : k = 3$ $w + \gamma + (3k-7)\alpha + 2\beta + k - 1 \quad : k \geq 4$

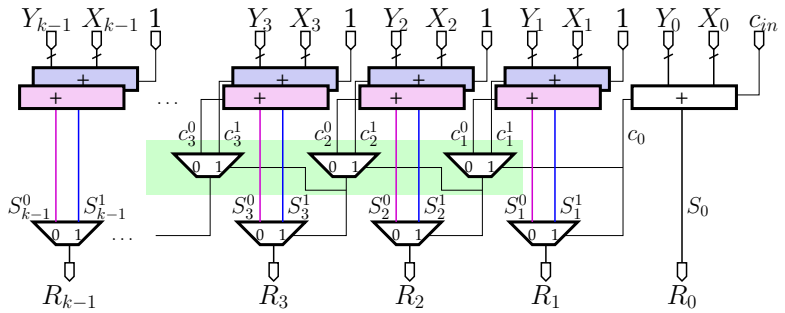


Figure 5.7 *Classic Carry-Select Architecture*

5.4.1 Classic carry-select adder

The classic carry-select adder [81] block consists of two RCAs and one multiplexer. Each pair of adders computes the two possible block results, one speculating on a carry-in of 0 and one on a carry-in of 1. The carry-in then feeds the select line of the multiplexer to choose the correct sub-sum and carry-out bit.

Large additions can be split into multiple carry-select adder blocks (k). The speculative sub-sums S_k^1, S_k^0 and corresponding carry-out bits c_k^1, c_k^0 are computed all in parallel. Please note that $c_k^1 : S_k^1 > c_k^0 : S_k^0$ (where the : operator denotes the concatenation of the carry-out bit to the sum) so that c_k^0 always implies c_k^1.

The carry-in ripples through the multiplexer network to propagate the correct carry-outs. Figure 5.7 presents the architecture of such an addition that is split into multiple carry-select blocks. For clarity, the block carry-out multiplexers have been separated from the block result multiplexers. The multiplexer network is generally fast. However, if greater performance is needed, a costly but faster carry look-ahead structure can be used for carry-bit computation.

Unfortunately, both the multiplexer network and carry look-ahead adders map poorly on FPGAs. This is because in FPGAs the routing delay exceeds by 3 to 4 times the delay of the logic element. Despite this major drawback, this naive mapping outperforms in latency the highly FPGA-optimized RCA for extremely large additions.

5.4.2 Acceleration of inter-block carries

The inter-block carries of the carry-select adder take a shortcut through the multiplexer network skipping a complete block with a single multiplexer stage. This advantage is mostly given away if the multiplexers are implemented using standard LUTs connected through the general-

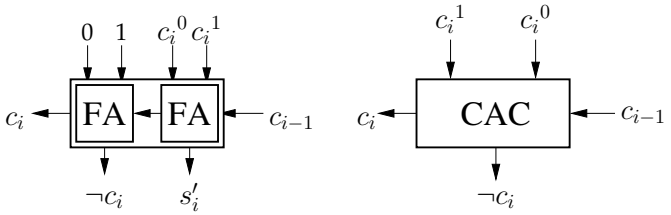


Figure 5.8 *Carry-Add-Cell (CAC) implementation and representation*

Table 5.3 *CAC Truth table. Greyed-out rows are not needed*

c_{i-1}	$c_i{}^0$	$c_i{}^1$	c_i	$\neg c_i$	s'_i
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	0	0
1	1	1	1	0	1

purpose routing network. To compete with the fast carry propagation within a block, the inter-block carry propagation must also exploit the available carry-chain structures. We will present two different techniques which make use of the fast-carry chains for inter-block carry acceleration.

As shown in Table 5.4, the different cases of the propagation of the inter-block carries can be easily distinguished by the values of the speculative block carry outputs. As c_k^0 implies c_k^1, the line $c_k^0\overline{c_k^1}$ can be neglected in the truth table. All others perfectly coincide with the carry propagation in a full adder so that the plain binary word addition of the bit vectors (c_k^0) and (c_k^1) produces the correct carry propagation.

Having an addition with the correct carries inside is of limited value if these cannot be accessed. While a direct tapping of the carry signals is, indeed, possible on the Virtex architectures, such a solution is not not portable (we did not find it possible to intercept the carry signals for Altera FPGAs) and would require the use of device-specific, low-level component primitives.

One solution to the portability issue is to express the carry-sum in such a way that the internal carry-out bits are also available on the "sum" outputs. This will make the architecture portable and still take advantage of the fast computational data-path ensured by the carry-chains. Therefore we express the carry-out computation under the form of a 2-bit addition addition (Figure 5.8) whose correctness can be verified using the truth table 5.3.

$$c_i \neg c_i s'_i = c_{i-1} + c_i{}^0 + c_i{}^1 + 2$$

The value of s'_i is not used further in the computation but is necessary for correct inference and mapping of the addition on the fast-carry chains of the FPGA.

The disadvantage of this approach compared to a low-level primitive implementation is that the carry-propagation circuit has twice the width. This is not a big impediment for additions whose width determine a relatively low number of chunks. However, in the following we provide an alternative solution for FPGAs offering 5-input LUTs which solves this inconvenience. This is the case of most modern FPGAs from Xilinx: Virtex-5/-6 and Altera StratixII-IV.

The following mapping will be achieved thanks to the technique described by Preußer and Spallek [133] for mapping general computations on the fast carry-chain structures. We start from the equation $s = p \oplus c_{\text{in}}$, which allows to infer the incoming carry from the obtained sum bit s_k, so that a standard addition operator suffices to implement the core carry-chain implementation:

Table 5.4 *Inter-Block Carry Propagation Cases*

c_k^0	c_k^1	c_k	–	Case
0	0	0	–	**K**ill
0	1	c_{k-1}	–	**P**ropagate
1	0	$*$	–	*Impossible*
1	1	1	–	**G**enerate

$$\begin{aligned} c_{k-1} &= s_k \oplus p_k \\ &= s_k \oplus \overline{c_k^0} c_k^1 \end{aligned} \tag{5.3}$$

and hence (see also Table 5.4):

$$\begin{aligned} c_k &= c_k^0 + c_{k-1} c_k^1 && \text{| by Eq. 5.3} \\ &= c_k^0 + \left(s_k \oplus \overline{c_k^0} c_k^1 \right) c_k^1 \\ &= c_k^0 + \overline{s}_k c_k^1 \end{aligned} \tag{5.4}$$

The carry computation circuit with the resulting recovery of the carries from the sum bits is depicted in Figure 5.9. Note that the recovery computation can often be merged into the further processing of the recovered carry signal.

5.4.3 The Add-Add-Multiplex (AAM) carry-select architecture

The AAM architecture derives directly from the classic carry-select architecture. The multiplexer chain computing the carry bits is replaced with the much faster carry-computation-circuit (CCC) and carry-recovery (CR) circuit. Figure 5.10(a) highlights the three stages of the AAM Carry-Select architecture:

1. For each block, two sums are computed, one for each possible value of the block carry-in. Both of these additions are extended to compute the block carry-out.
2. The two bit vectors formed by the block carries speculating on a carry-in of 0 and 1 are added in the CCC using a fast short ripple-carry adder. The output sum bits and their two respective speculative input carries are fed to the CR circuit, which recovers the proper block carry outputs.

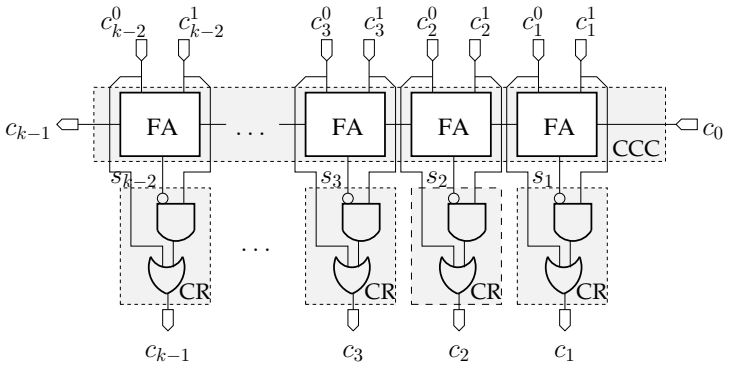


Figure 5.9 *Carry Computation Circuit with Carry Recovery*

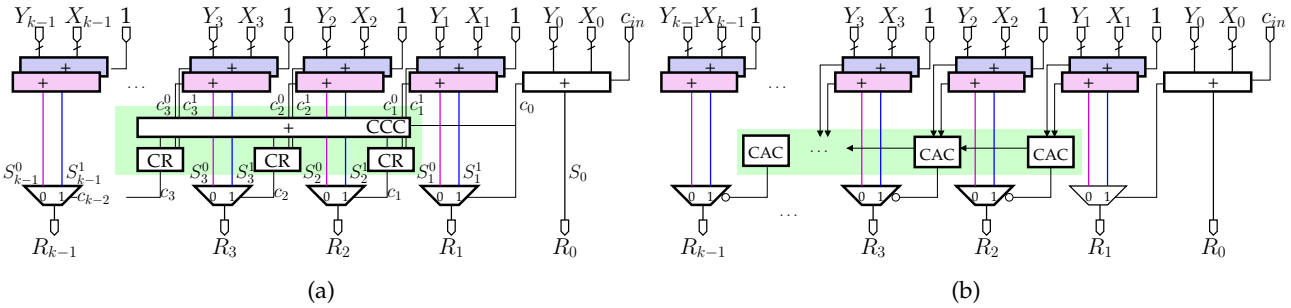


Figure 5.10 *The AAM Carry-Select Architecture using (a) the CCC and CR and (b) the CAC*

3. The computed block carries are used to select the proper speculative block sum for the adder output.

The AAM architecture uses a multiplexer to select among the two block sums. The multiplexer is a 3-input function, the two sum-bits and the carry-bit generated by the CR. For FPGAs with 5-input LUTs, the CR can be merged with the multiplexing. This is the case of FPGAs like Virtex-5 and Virtex-6 having 6-input LUTs, and also Altera Stratix devices whose ALUTs can be configured for supporting 5-input functions. Having only 4-input LUTs available such as on Virtex-4 devices, the CR introduces an extra LUT level and a supplementary wire delay. On these architectures, adders with a low block count and, thus, a short CCC should prefer the first carry-acceleration technique based on the CAC (Figure 5.10(b)). It uses extra intermediate propagating stages but provides direct access to the inverted propagated carry.

5.4.4 The Compare-Add-Increment (CAI) carry-increment architecture

The CAI architecture adopts some features from the carry-increment adder, a widely adopted structural simplification of the carry-select scheme. In particular, the CAI only uses the block sums produced for the case of no incoming block carry. The final multiplexer stage is replaced by another adder, which adds the actual incoming carry and, thus, corrects the produced sum if necessary. Note that the choice of this incrementer instead of a multiplexer does not increase the number of occupied LUTs.

As the CAI does not need the sum speculating on an incoming block carry, the corresponding adder only serves the purpose of computing the associated carry-out of the speculative block sum $X_k + Y_k + 1$. This can, however, be obtained by the simple comparison:

$$c_k^1 \texttt{ <= '1' when } X_k \geq \texttt{not}(Y_k) \texttt{ else '0';} \tag{5.5}$$

All in all, the CAI offers the following improvements:

1. The use of a comparator for the computation of c_k^1 is, at most, as complex as the replaced addition. On Virtex5 and Virtex6 devices, the number of required LUTs is even halved as every stage on the carry chain processes two adjacent input positions rather than just one. This is possible as the sum bits are not asked for.
2. The number of registers required in a pipelined implementation is almost halved as only one of the two speculative block sums must be stored.
3. The wide fanout of the computed block carries for the control of the multiplexers is eliminated.

The resulting architecture is sketched in Figure 5.11. On FPGAs with 5-input LUTs, the CR is merged into the LSB computation of the final addition. On 4-input LUT FPGAs the final addition is extended with one lower bit for computing the CR output signal.

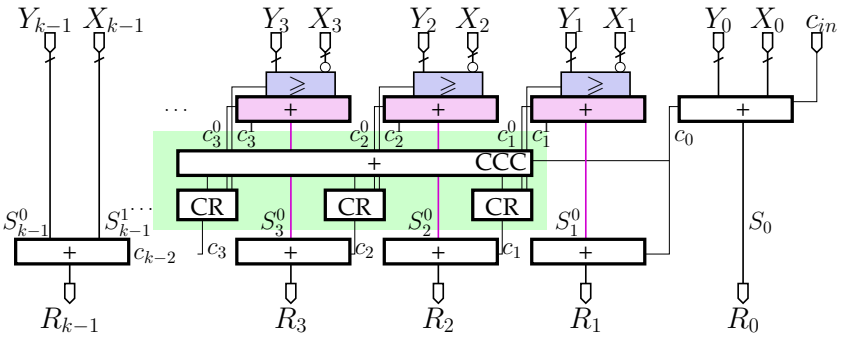


Figure 5.11 *The CAI Carry-Increment Architecture*

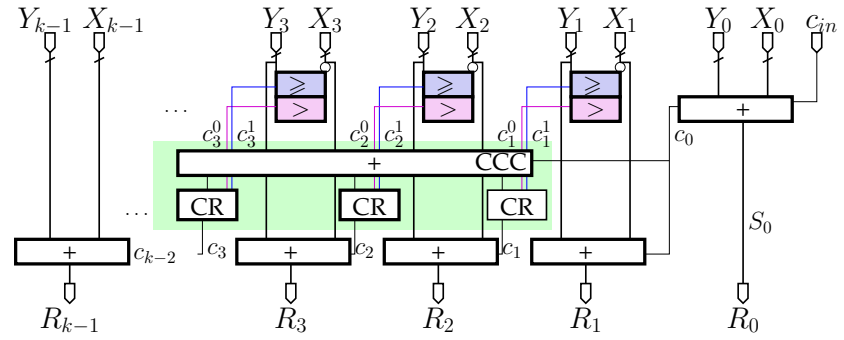


Figure 5.12 *The CCA Carry-Select Architecture*

5.4.5 The Compare-Compare-Add (CCA) carry-select architecture

The CCA architecture takes the CAI architecture one step further. It uses two comparators to generate both c_i^1 and c_i^0.

$$c_k^0 \texttt{ <= '1' when } X_k > \texttt{not}(Y_k) \texttt{ else '0';} \tag{5.6}$$

The final step is turned from an incrementer into a complete adder computing $X_k + Y_k + c_k$.

The greatest benefit of this implementation is achieved on FPGAs with 5-input LUTs. Not only can the CR be merged into the LSB computation of the final addition, but the whole critical path is shortened as the computation of both speculative block carries is only half as wide as a true adder. The architecture is outlined in Figure 5.12.

5.4.6 Block-splitting strategies

The data dependences between stages of the proposed architectures together with the FPGA-specific component timings yield different block-splitting strategies for maximizing adder size for a frequency f.

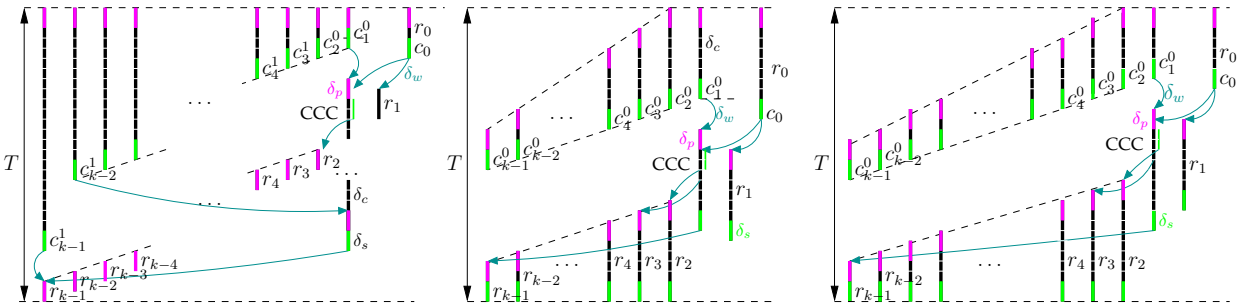
We denote by L the addition size. Our objective is finding a length k vector of block sizes denoted by $(l_{k-1}...l_0)$, $L = \sum_0^{k-1} l_i$, such that the circuit delay does not exceed the target period T.

Let us now recall the delay primitives we will be using next:

– delay of obtaining the j^{th} sum bit:

$$\delta_{s_j} = \delta_p + (j-1)\delta_c + \delta_s \tag{5.7}$$

– for the j^{th} addition bit the inputs x_j, y_j can arrive later that x_{j-1}, y_{j-1}, as long as the produced *propagate* signal gets synchronized with carry c_{j-1}.



(a) AAM (b) CAI (c) CCA

Figure 5.13 *Computation scheduling for the proposed architectures*

The delay of the $j-1^{\text{th}}$ carry-bit is:

$$\delta_{c_{j-1}} = \delta_p + j\delta_c \tag{5.8}$$

resulting that the inputs x_j, y_j can arrive as late as:

$$\delta_{x_j} \leq j\delta_c \tag{5.9}$$

– the delay of the comparator varies among FPGA devices:

$$\delta_{\text{cmp}_k} = \begin{cases} \delta_p + k\delta_c + \delta_s & \text{Virtex-4, StratixII-IV} \\ \delta_p + \lceil (k/2) \rceil \delta_c + \delta_s & \text{Virtex-5/-6} \end{cases} \tag{5.10}$$

On Virtex4 the delay of a k-bit comparator is equal to that of a k-bit RCA while on Virtex5 the same comparator maps in half the LUTs.

– the wire delay, δ_w

The Add-Add-Multiplex architecture

The constraints given by the timing model of this architecture will allow us to determine the optimal block sizes. A visual indication of a tight computation scheduling which optimizes the AAM block-sizes is given in Figure 5.13(a). The length of the segments is proportional to the computation delay of the components (adders and multiplexers for AAM). The length of the RCA delays (first stage) is proportional to the block size.

Considering the timing and architectural constraints, the CCC is a $k-2$-bit RCA having the delay of the MSB $\delta_{s_{k-2}}$ (Eq. 5.7). The MSB inputs the select line of the $k-1^{th}$ block multiplexer (Figure 5.10(a)), having a delay δ_{MUX}.

On the other hand, as CCC is implemented as an RCA, it allows the inputs to be delayed at most as specified by Equation 5.9. As the speculative carries (c_i^1 and c_i^0) are also computed using RCAs, this allows the size of successive blocks to increase by exactly one bit.

We therefore choose to fix the 2^{nd} block size, $l_1 = 1$ bit. For a given frequency f, this sets the maximum value of k, which is the solution of the equation:

$$\delta_{s_1} + \delta_w + \delta_{s_{k-2}} + \delta_w + \delta_{MUX} = T \tag{5.11}$$

As successive-block size increases by exactly one bit, $l_{k-2} = k-2$. Blocks 1 to $k-2$ total $(k-2)(k-1)/2$ bits. The l_{k-1} and l_0 block sizes are the solutions of the equation:

$$\delta_{s_{l_{k-1}}} = T - (\delta_w + \delta_{MUX}) \tag{5.12}$$

$$\delta_{s_{l_0}} = \delta_{s_{l_1}} + \delta_{LUT} \tag{5.13}$$

The maximal addition size for frequency f is $l_0 + (k-2)(k-1)/2 + l_{k-1}$.

The Compare-Add-Increment architecture

The CAI architecture computes the speculative c_i^1 bit using Equation 5.5. On Virtex5 devices this comparison takes half the resources needed to obtain c_i^1 using a RCA. The latency improvement over these devices is given in equation 5.10. However, this latency improvement is lost by using a RCA for computing c_i^0.

The third stage of the CAI architecture is an incrementation of the speculative sum for a 0 carry-in (S_i^0) with the carry-in obtained by the CCC. The incrementation is implemented as a RCA in FPGAs.

The output delays of the sum-bits of CCC are given in Equation 5.7. The difference between successive sum bits is δ_c. The sum-bits are used as carry-in bits for the final stage adder. If we enforce that all the result bits be synchronized (Figure 5.13(b)) this leads to successive blocks having the size decreased by 1-bit.

We choose to fix the size of the $k-1^{th}$ block, $l_{k-1} = 1$ bit which leads to $l_2 = k-2$. Moreover, the difference in input delay between the speculative carry bits of l_2 and of l_1 for CCC is δ_c. This leads to $l_1 = l_2 - 1 = k-3$.

Given the constraint that the carry-out of block 0 is the carry-in of CCC, the size of this block is the solution of the equation:

$$\delta_{s_{l_0}} = \delta_{s_{l_1}} + \delta_p \tag{5.14}$$

The maximal adder size for this architecture for frequency f is $(k-2)(k-1)/2 + k - 3 + l_0$.

The Compare-Compare-Add architecture

The CCA architecture uses comparators for computing the two speculative caries, c_i^0, c_i^1 (Equations 5.6,5.5). When compared to the CAI architecture, the latency of the first stage is reduced on Virtex-5/-6 devices.

However, the block splitting strategy remains the same. The size of the first chunk is now the solution of the equation:

$$\delta_{\text{cmp}_{l_1}} + 2\delta_w + \delta_p + \delta_s + \delta_{s_{l_2}} = T \tag{5.15}$$

where $l_2 = k-2$.

The number of blocks (k) is now the solution of the equation:

$$\delta_{\text{cmp}_{l_2}} + \delta_w + \delta_p + \delta_s + \delta_w + \delta_{s_{l_3}} = T \tag{5.16}$$

The size of block 0 is:

$$\delta_{s_{l_0}} = \delta_{\text{cmp}_{l_1}} + \delta_p \tag{5.17}$$

5.4.7 Area complexity of the designs

Once the block-splitting procedure is finished, we can closely approximate the area of the circuit on the FPGA.

In this section we present the LUT-count formulas for the proposed architectures for Virtex5/6 devices. Similar formulas can be derived for Virtex4 devices and Altera devices. The formulas are deduced based on the resources occupied by the basic blocks:

– 2:1 n-bit multiplexer occupies n LUTs.
– n-bit RCA takes n LUTs
– n-bit comparator takes $\lceil n/2 \rceil$ LUTs on Virtex5/6 and n LUTs on Virtex4/StratixII-IV.

Consequently, based on the chunk-size vector $(l_{k-1}, ..., l_0)$ returned by the previous step and the addition size L, the size of the architectures is:

1. for the AAM architecture

$$LUTs = \sum_{0}^{k-1} l_i + \sum_{1}^{k-1} l_i + k - 2 + \sum_{1}^{k-1} l_i = 3L - 2l_0 + (k-2),$$

2. for the CAI architecture

$$LUTs = \sum_{0}^{k-2} l_i + \sum_{1}^{k-2} \left\lceil \frac{l_i}{2} \right\rceil + k - 2 + \sum_{1}^{k-1} l_i \approx \frac{5}{2}L - \frac{3}{2}l_0 - \frac{3}{2}l_{k-1} + (k-2),$$

3. for the CCA architecture

$$LUTs = l_0 + 2\sum_{1}^{k-2} \left\lceil \frac{l_i}{2} \right\rceil + k - 2 + \sum_{1}^{k-1} l_i \approx 2L - l_0 - l_{k-1} + (k-2).$$

Block sizes $(l_{k-1}, ..., l_0)$ and the number of blocks k are different in the above formulas for the three architectures. One can use Figure 5.13 for the order of magnitude of the block sizes $(l_{k-1}, ..., l_0)$.

Comparison with pipelined-RCA schemes

The immediate advantages of the proposed addition architectures when compared to pipelined RCA architectures is the reduction of pipeline stages of the design. We are interested in the area cost we have to trade to get this advantage. Consequently, we have compared the area magnitude of our architectures against the previously pipelined RCA architectures.

Table 5.5 synthesizes resource estimation formulas for Virtex5 FPGAs. Please note that the values of k and $(l_0, ..., l_{k-1})$ might be different for all these architectures, only the addition size L remains constant. The proposed addition architectures represent very attractive alternatives to the pipelined RCA schemes. For more than two pipeline levels the CCA architecture takes approximately as many resources as the pipelined schemes while at the same time reducing pipeline depth. For a larger number of pipeline levels, the proposed architectures takes fewer resources, providing that it can match the frequency.

Pipelining options

The short-latency architectures presented so far are all combinatorial. They allow reducing the number of pipeline stages by effectively replacing deeply pipelined RCA. However, for very large additions at very-high frequencies the architectures are unable to provide a satisfactory solution. Pipelining them (usually one pipeline level suffices) is a solution for these contexts.

Table 5.5 *Area comparison against pipelined RCA schemes for Virtex5 and addition size L*

Architecture	LUT-FF pairs	Depth
AAM	$3L - 2l_0 + (k-2)$	0
CAI	$\frac{5}{2}L - \frac{3}{2}l_0 - \frac{3}{2}l_{k-1} + (k-2)$	
CCA	$2L - l_0 - l_{k-1} + (k-2)$	
Classical	$8L/3$	2
	$3L$	3
	$16L/5$	4
Alternative	$7L/3$	2
	$11L/4$	3
	$3L$	4

The AAM architecture can be effectively pipelined by inserting the register level after the first addition stage. The registers are combined with the LUTs for free.

For the CAI architecture, the register level can be similarly inserted after the first computations. Although several registers may be combined with LUTs, there is a small increase of $2l_{k-1}$ LUT Flip-Flop pairs for buffering the final block inputs. One solution to save l_{k-1} LUTs would be to perform the final chunk computation for $c_{in} = 0$. Inserting the register before the last computation phase requires in addition buffering the CCC outputs, therefore yielding a less attractive solution.

The CCA architecture can easily be pipelined. The first two levels are regrouped to balance the size of the adders at the last level. Pipelining this architecture is expensive, costing an additional $2L - l_0$ LUT Flip-Flops pairs.

One should only consider the pipelined implementations when none of the combinatorial versions are capable of reaching the requested frequency. When deciding what pipelined architecture to use, one should first try the CAI architecture, and, if this one also fails, one should go with the pipelined AAM architecture.

5.5 Global inference of shift-registers

In the case of Xilinx FPFAs, we have so far relied on the fact that the pipelined addition schemes can make extensive use of the shift-registers available in SLICE_Ms. However, this resource is getting rarer over the years: all VirtexII-Pro slices device were similar to SLICE_Ms, their number was cut to half with respect to the total number of slices in Virtex4 and Spartan3 devices, and is roughly equal to one quarter (with higher density at the input of the DSP48E blocks) in Virtex-5/-6 devices. Moreover, the granularity of these blocks has also increased over the year: the LUT6 of Virtex-5/-6 devices can be configured as a 64-bit memory or SRL32 (shift-register with maximum 32 levels). The effectiveness of using SRL32 for implementing a 2-3 level shift-register is questionable. There may be better uses of these resources for longer-length shift-registers. Moreover, the ISE synthesizer has an option that prevents using this resource. It may therefore be relevant to be able to generate adders with this in view.

Moreover, the larger-granularity of embedded memories in Altera devices also validates the necessity of generating adders in this context.

Out of the presented architectures, the low-latency architectures one will behave best when no shift registers are allowed. On one hand, being strictly combinatorial, it does no use registers. On the other hand, when pipelined, two register levels usually suffice. These registers can naturally be paired with LUTs, bringing no area overhead.

Resource estimations for the pipelined architectures when SRLs are not allowed are presented in Table 5.1.

Table 5.6 *Relative Error for the estimation formulas on a 128-bit adder Virtex4 and StratixIII devices for a requested frequency of 400MHz.*

Freq.	Architecture	SRL	Target	Depth	Results LUTs	Results REG	Results LUT-FF/2	Estimations LUTs	Estimations REG	Estimations LUT-FF*	Relative Error LUTs	Relative Error REG	Relative Error LUT-FF*
400 MHz	Classical	N	Virtex-4	3	131	579	307	131	579	611	0%	0%	0.4%
			Stratix-III	3	135	519	263	131	519	521	2%	0	0.9%
		Y	Virtex-4	3	291	355	195	291	355	387	0	0	0.7%
	Alternative	N	Virtex-4	3	229	390	214	224	393	425	2%	0.7%	0.7%
			Stratix-III	3	219	390	197	219	393	395	0%	0.7%	0.2%
		Y	Virtex-4	3	293	326	182	291	322	356	0.6%	1%	2%

Table 5.7 *Resource usage of 128-bit wide pipelined adders for different utilization contexts for a target frequency of 400MHz (SRL allowed, post place-and-route)*

δ_{in}	Classical k	β	α	γ	Expected L	Expected R	Expected L-FF	Obtained L	Obtained R	Obtained L-FF/2	Alternative k	β	α	Expected L	Expected R	Expected L-FF	Obtained L	Obtained R	Obtained L-FF/2
0	4	32	32	32	191	355	387	191	355	195	4	32	32	291	322	356	293	326	182
1.2e-9	5	14	32	18	338	420	434	338	420	219	8	2	18	417	464	466	421	454	239
1.5e-9	5	23	32	9	347	420	443	347	420	225	15	2	9	494	508	579	502	508	285

5.6 Reality check

5.6.1 Estimation formulas

We have checked our estimation formulas against synthesis results using Xilinx ISE 11.5 and QuartusII 10.1. Results presenting the resource usage estimations, obtained results and relative errors for both with and without SRLs are presented in Table 5.6 for a 128-bit addition synthesized on a Virtex4 (speedgrade -12) and StratixIII (speedgrade C2) with a required frequency of 400MHz.

First, it should be mentioned that all the synthesized adders met the frequency target. In addition, one may observe that the resource estimations are accurate for all criteria. The best estimations are obtained, as expected, for LUTs and registers. The LUT-FF estimations represent the lowest bound obtainable leading to underestimation of the result. Nevertheless, the relative error of the estimation remains small, of the order of one percent.

5.6.2 Synthesis results

The highlighted cells in Table 5.7 indicate the lowest costs for the given metric. We can observe that different context (input delays) greatly influence the size of the architecture. The advantage of the generator is that we can perform this exploration and always choose the best architecture.

Next we have decided to test our proposed short-latency architectures. The largest theoretical adder width for a given frequency is plotted in Figure 5.14 for Virtex5 FPGAs. We have focused on the 200-300MHz frequency range for two reasons: 1) for lower frequencies the highly optimized RCA manages to provide sufficient performance; 2) larger frequencies are hard to obtain due to routing congestion for chip-filling designs. As expected, the proposed architectures provide 1-cycle solutions for a wide range of interesting addition sizes.

Table 5.8 presents a comparison between our proposed architectures and a pipelined RCA

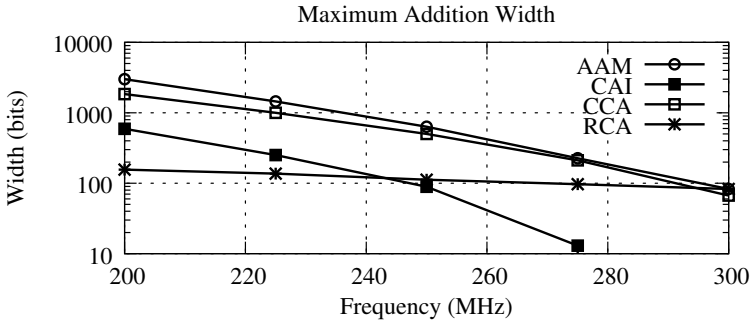


Figure 5.14 *Maximum adder width vs circuit frequency on Virtex5*

Table 5.8 *Post place-and-route results on Virtex5 (-3) for various adder sizes and a target f=250MHz using ISE 11.5. δ_{cp} denotes the length of the design's critical path*

Size	RCA – Coregen			AAM		CAI		CCA	
bits	LUT-FF	$\delta_{cp}(ns)$	cycles	LUT-FF	$\delta_{cp}(ns)$	LUT-FF	$\delta_{cp}(ns)$	LUT-FF	$\delta_{cp}(ns)$
128	145	3.48	1	391	3.93	330	3.99	271	4.16
256	469	4.98	1	795	4.43	641	4.25	534	4.37
384	1042	4.88	2	1187	4.58	972	4.91	818	4.75
512	1541	4.36	3	1581	4.62	1290	5.10	1062	4.98

implementation in terms of occupied resources, critical-path length and number of pipeline levels for addition sizes ranging from 128 to 512-bits, targeting a frequency of 250MHz. The presented numbers have been obtained after place-and-route using ISE 11.5.

The results prove two points: 1) the routing delay penalty for proposed architectures has the same order of magnitude as for a pipelined RCAs 2) for sufficiently large widths, the proposed architectures take less resources while reducing the cycle count to one.

Table 5.9 presents a comparison of the AAM architecture against [125], the Altera `lpm_add_sub` megafunction [9] and the alternative RCA pipelining scheme. Compared to the combinatorial approach presented in [125] the critical path delay of our architecture is much shorter. When compared to the pipelined approaches, the AAM architecture provides a design that does not need pipelining with a competitive area.

5.7 Conclusions

This chapter has presented the binary adder generator part of FloPoCo, comprising of several different adder architectures. The area of these architectures can be computed on-the-fly based on the deduced resource estimation formulas thanks to the high-level programming language of our generator. Once the best suited architecture for a given user context is found, its VHDL code is

Table 5.9 *Post place-and-route synthesis results for 128-bit addition on StratixIII*

Tool	**Area**		**Latency**	
	ALUT	REG	$\delta_{cp}(ns)$	cycles
[125]	?	-	7.73	-
MegaWizard(lpm_add_sub)	270	259	3.64	2
RCA Alternative	190	129	3.18	1
AAM	376	-	3.70	-

directly produced.

Moreover, as binary adders are often subcomponents in larger designs, their integration in the sub-cycle accurate pipelining framework of FloPoCo is of primal importance. This is again possible thanks to the programming-language support of our generator.

There is still room for improvement in what concerns incorporating these architectures in coarser-grain operators. One such situation is when part of adder's inputs arrive from a shift-register. Then, the registers (or part of them) required for synchronizing the classical architecture's inputs will be absorbed by these shift-registers yielding in a real cost smaller than the one reported by our formulas. We are currently considering integrating this framework support in FloPoCo.

Other optimization possibilities can arise if one accounts that different sections of the adder's inputs are available at different clock cycles. For an example, if two adders pipelined using the classical technique are chained, all the synchronization registers between can be discarded, yielding in a more economical architecture. The `IntNAdder` component of FloPoCo accounts for this information in the case of addition. Again, we are considering adding framework support which would allow exploiting these opportunities in the general case.

All these optimizations are local, and target finding the local minima for that particular adder instance. We are still exploring whether using an adder which a shorter-latency (number of cycles) but with a higher local cost (LUT-FF) may globally reduce resource usage my minimizing synchronization cost.

Thanks

Most of the material presented in this chapter is based on collaborations with Hong Diep Nguyen at the time he was involved in his PhD at ENS de Lyon, Thomas Preußer from the Institute of Computer Engineering at TU Dresden, Germany whom I had the pleasure to meet at FPL'10 in Milano. I would like to thank them for their contributions.

CHAPTER 6
Large multipliers with fewer DSP blocks

A paper-and-pencil analysis of FPGA peak floating-point performance [145] clearly shows that DSP blocks are a relatively scarse resource when one wants to use them for accelerating double-precision (64-bit) floating-point applications.

Moreover, demand for more accuracy is growing, especially in scientific computing [61], and the IEEE-754-2008 revision of the Standard for Floating-Point Arithmetic [17] has introduced a higher precision floating-point format: quadruple precision (QP), a 128-bit format including a 112-bit mantissa. So far no general purpose processor offers hardware floating-point units supporting this format. Proprietary core generators such as LogiCore [6] from Xilinx and Megawizard [9] from Altera currently do not scale to QP either.

In this chapter we focus on techniques reducing DSP block usage for large multipliers. Here, *large* means: any multiplier that, when implemented using DSP blocks, consumes more than two of them, with special focus on the multipliers needed for single-precision (24-bit), double-precision (53-bit) and quadruple-precision(113-bit) floating-point. Although the techniques are presented here in the context of unsigned multipliers, their extension to sign multipliers is straightforward.

There are many ways of reducing DSP block usage, the simplest being to implement multiplications in logic only. However, a LUT-based large multiplier has a large LUT cost (at least n^2 LUTs for n-bit numbers, plus the flip-flops for pipelined implementations). In addition, there is also a large performance cost: a LUT-based large multiplier will either have a long latency, or a slow clock. Still, for some sizes, it makes sense to implement as LUTs some of the sub-multipliers which would use only a fraction of a DSP block.

We focus here on algorithmic reduction of the DSP cost, and specifically on approaches that consume few additional LUTs, add little to the latency (and sometime even reduce it), and operate at a frequency close to the peak DSP frequency.

The presented multipliers have been implemented as part of the FloPoCo class hierarchy and are extensively used in coarser operators, as those presented in the following chapters. All the results presented have been obtained using ISE 11.5 / LogiCore Multiplier 11, after placing and routing, unless explicitly stated otherwise.

6.1 Large multipliers using DSP blocks

Let k be an integer parameter, and let X and Y be $2k$-bit integers to multiply. We will write them in binary $X = \sum_{i=0}^{2k-1} 2^i x_i$ and $Y = \sum_{i=0}^{2k-1} 2^i y_i$.

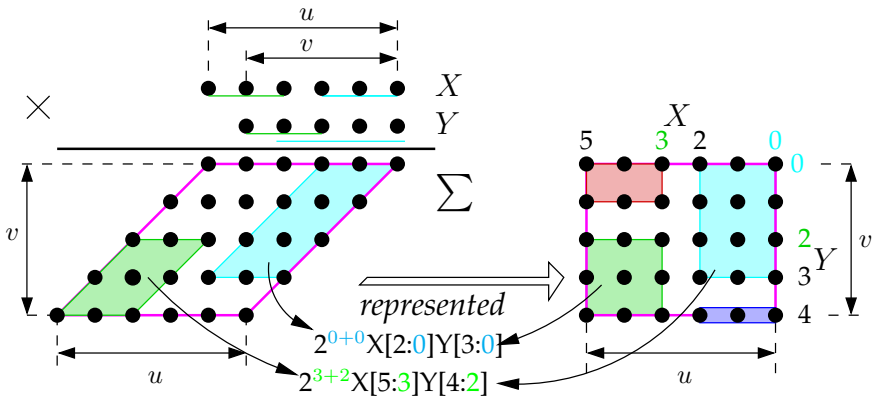


Figure 6.1 *$u \times v$-bit multiplier*

Let us now split each of X and Y into two subwords of k bit each:

$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

X_1 is the integer formed by the k most significant bits of X, and X_0 is made of the k least significant bits of X.

The product $X \times Y$ may be written

$$X \times Y = (2^k X_1 + X_0) \times (2^k Y_1 + Y_0)$$

or

$$X \times Y = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_0 + X_0 Y_1) + X_0 Y_0 \tag{6.1}$$

This product involves 4 sub-products. If k is the input size of an embedded multiplier, this defines an architecture for the $2k$ multiplication that requires 4 embedded multipliers. This architecture can also be used for any input size between $k+1$ and $2k$. Besides, it can be generalized: For any $p > 1$, numbers of size between $pk - k + 1$ and pk may be decomposed into p k-bit numbers, leading to an architecture consuming p^2 embedded multipliers.

Early FPGAs had only embedded multipliers [15], but the more recent DSP blocks [16, 21, 14, 22] also include internal adders and cascading features, designed in such a way that most of the additions in Equation (6.1) can be computed inside the DSP blocks (see page 9 for more information on the DSP block structure in modern FPGA devices). In this Chapter we also focus on effectively using these internal adder structures for minimizing global logic cost.

6.2 Visual representation of multipliers

Throughout this chapter we will make extensive use of visual representations of large multipliers. Let's consider again the multiplication $X \times Y$ with our operands on u and v bits respectively. Each line from the multiplication described in Figure 6.1 presents the operands formed by the bitwise multiplication between Y[i], $i \in [0..v-1]$ and X. In order to obtain the final multiplication result these operands need to be summed together.

The contribution of each sub-product Y[i_Y:j_Y]X[i_X:j_X] with $j_Y \leq i_Y$, $j_X \leq i_X$, $j_X \leq u-1, j_Y \leq v-1$, $i_X, i_Y \geq 0$ can be clearly be identified in the sub-product diamond. Figure 6.1 highlights two such sub-products: X[2:0]Y[3:0] and X[5:3]Y[4:2]. Their contribution to the final product is weighted by the sums of their operand magnitudes: 2^0 for X[2:0]Y[3:0] and 2^{3+2} for

X[5:3]Y[4:2]. The sum of weighted contribution of all sub-products with non-overlapping contributions is equal to the product XY.

An equivalent but more natural representation is obtained by converting the diamond into a rectangle by aligning all rows to the right (Figure 6.1). The small diamond tiles are now rectangular, and are easier to manipulate.

In this new representation building a large multiplier reduces to tiling the multiplier's rectangular board with rectangular, non-overlapping [1] tiles. Once a valid tiling is performed, it can easily be converted into an architecture. The contribution of each tile is equal to the tile's projection on the X and Y axis (in Figure 6.1 the green tile computes the product X[5:3]Y[4:2]) weighted by 2 to the sum of the tile's upper right corner coordinates (2^{3+2}). In the case of Figure 6.1, the tiling on the right computes:

$$XY = 2^{0+0}X[2:0]Y[3:0] + 2^{3+2}X[5:3]Y[4:2] + 2^{3+0}X[5:3]Y[1:0] + 2^{0+4}X[2:0]Y[4:4]$$

6.3 Karatsuba-Ofman algorithm

6.3.1 Two-part splitting

Let us now consider again our two inputs X, Y on $2k$ bits each. The classical step of Karatsuba-Ofman algorithm is the following. First compute $D_X = X_1 - X_0$ and $D_Y = Y_1 - Y_0$. The results are signed numbers that fit on $k+1$ bits [2]. Then compute the product $D_X \times D_Y$ using a DSP block. Now the middle term of equation (6.1), $X_1Y_0 + X_0Y_1$, may be computed as:

$$X_1Y_0 + X_0Y_1 = X_1Y_1 + X_0Y_0 - D_XD_Y \tag{6.2}$$

Then, the computation of XY using (6.1) only requires three multiplier blocks: one to compute X_1Y_1, one for X_0Y_0, and one for D_XD_Y.

This computation can be visualized in Figure 6.2 using the already introduced tiling representation. There, black-square tiles are products which are computed by means of direct multiplications. White-square tiles are grouped in pairs, symmetrical to the black-square diagonal. One pair of white-square tiles is computed using the two already computed black-square tiles (in the case of 2-way splitting: X_0Y_0 and X_1Y_1) and only one more sub-product: $(X_1 - X_0)(Y_1 - Y_0)$ (the dashed square groups two black-square and two white-square tiles indicating the tiles part of this computation).

There is an overhead in terms of additions. In principle, this overhead consists of two k-bit subtractions for computing D_X and D_Y, plus one $2k$-bit addition and one $2k$-bit subtraction to compute equation (6.2). There are still more additions in equation (6.1), but they also have to be computed by the classical multiplication decomposition, and are therefore not counted in the overhead.

Counting one LUT per adder bit [3], and assuming that the k-bit addition in LUTs can be performed at the DSP operating frequency, we get a theoretical overhead of $6k$ LUT. However, the actual overhead is difficult to predict exactly, as it depends on the scheduling of the various operations, and in particular in the way we are able to exploit registers and adders inside DSPs. There may also be an overhead in terms of latency, but we will see that the initial subtraction latency may be hidden, while the additional output additions use the cycles freed by the saved multiplier.

1. Overlapping parts of tiles are computed twice and need to be subtracted in order to keep the tiling valid
2. There is an alternative Karatsuba-Ofman algorithm computing $X_1 + X_0$ and $Y_1 + Y_0$. We present the subtractive version, because it uses the Xilinx 18-bit signed-only multipliers fully, while working on Altera chips as well.
3. In all the following we will no longer distinguish additions from subtractions, as they have the same LUT cost in FPGAs.

	Latency	Frequency	Slices	DSPs
LogiCore	6	447	26	4
LogiCore	3	176	34	4
K-O-2	3	317	95	3

Table 6.1 *34x34 multipliers on Virtex-4 (4vlx15sf363-12).*

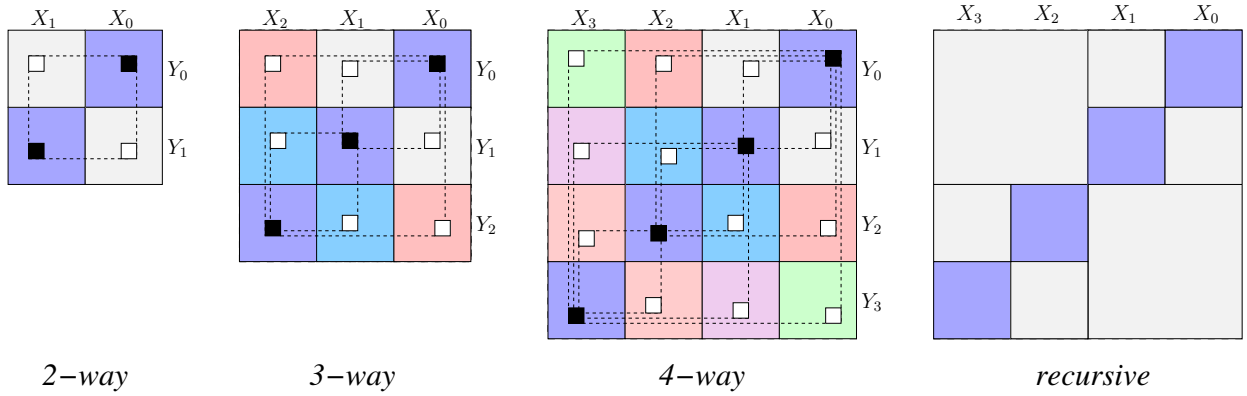


Figure 6.2 *$u \times v$-bit multiplier*

At any rate, these overheads are much smaller than the overheads of emulating one multiplier with LUTs at the peak frequency of the DSP blocks. Let us now illustrate this discussion with a practical implementation on a Virtex-4.

6.3.2 Implementation issues on Virtex-4

The fact that the differences D_X and D_Y are now signed 18-bit is actually a perfect match for a Virtex-4 DSP block.

Figure 6.3 presents the architecture chosen for implementing the previous multiplication on a Virtex-4 device. The shift-cascading feature of the DSPs allows the computation of the right-hand side of equation (6.2) inside the three DSPs at the cost of a 2k-bit subtraction needed for recovering X_1Y_1. Notice that here, the pre-subtractions do not add to the latency.

Table 6.1 presents the corresponding synthesis results for this operator, which is compared against the architecture of LogiCore multipliers. As we can see from these results, the overhead in terms of logic is minor for similar performances while our architecture consumes one DSP less. For the same latency, our architecture manages to outperform the LogiCore multipliers.

6.3.3 Three-part splitting

Now consider two numbers of size $3k$, decomposed in three subwords each:

$$X = 2^{2k}X_2 + 2^kX_1 + X_0 \quad \text{and} \quad Y = 2^{2k}Y_2 + 2^kY_1 + Y_0$$

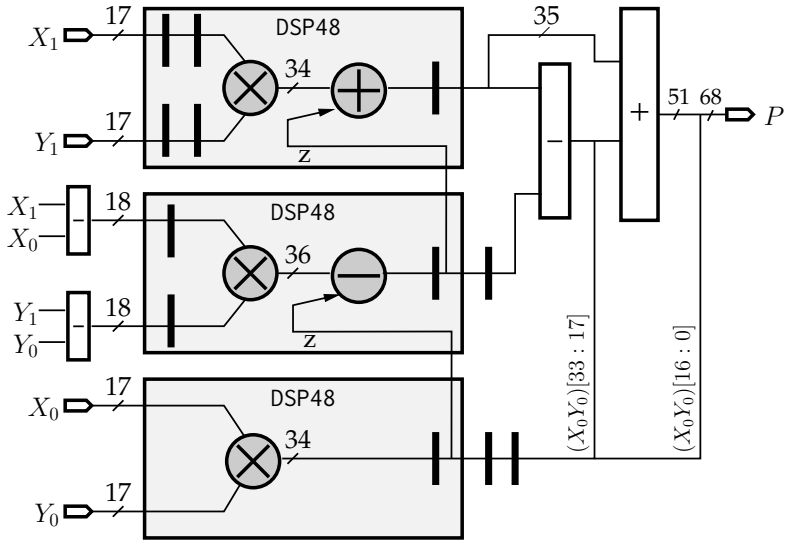


Figure 6.3 *34x34bit multiplier using Virtex-4 DSP48*

We have

$$\begin{aligned} XY &= 2^{4k}X_2Y_2 \\ &+ 2^{3k}(X_2Y_1 + X_1Y_2) \\ &+ 2^{2k}(X_2Y_0 + X_1Y_1 + X_0Y_2) \\ &+ 2^{k}(X_1Y_0 + X_0Y_1) \\ &+ X_0Y_0 \end{aligned} \tag{6.3}$$

After precomputing $X_2 - X_1$, $Y_2 - Y_1$, $X_1 - X_0$, $Y_1 - Y_0$, $X_2 - X_0$, $Y_2 - Y_0$, we compute (using DSP blocks) the six products

$$\begin{aligned} P_{22} &= X_2Y_2 & D_{21} &= (X_2 - X_1) \times (Y_2 - Y_1) \\ P_{11} &= X_1Y_1 & D_{10} &= (X_1 - X_0) \times (Y_1 - Y_0) \\ P_{00} &= X_0Y_0 & D_{20} &= (X_2 - X_0) \times (Y_2 - Y_0) \end{aligned}$$

and equation (6.3) may be rewritten as

$$\begin{aligned} XY &= 2^{4k}P_{22} \\ &+ 2^{3k}(P_{22} + P_{11} - D_{21}) \\ &+ 2^{2k}(P_{22} + P_{11} + P_{00} - D_{20}) \\ &+ 2^{k}(P_{11} + P_{00} - D_{10}) \\ &+ P_{00} \end{aligned} \tag{6.4}$$

Here we have reduced DSP usage from 9 to 6 which, according to Montgomery [121], is optimal. There is a first overhead of $6k$ LUTs for the pre-subtractions (again, each DSP is traded for $2k$ LUTs). Again, the overhead of the remaining additions is difficult to evaluate. Most may be implemented inside DSP blocks. However, as soon as we need to use the result of a multiplication twice (which is the essence of Karatsuba-Ofman algorithm), we can no longer use the internal adder behind this result, so LUT cost goes up. Table 6.2 provides some synthesis results. The implementation provides lower latency, higher frequency and reduced DSP cost from 9 to 6 at the expense of some logic.

	Latency	Frequency	Slices	DSPs
LogiCore	11	353	185	9
LogiCore	8	264	102	9
K-O-3	8	387	387	6

Table 6.2 *51x51 multipliers on Virtex-4 (4vlx15sf363-12).*

6.3.4 4-part splitting

Classically, the Karatsuba idea may be applied recursively: A 4-part splitting is obtained by two levels of 2-part splitting. However, a direct expression allows for a more straightforward implementation. From

$$\begin{aligned} X &= 2^{3k}X_3 + 2^{2k}X_2 + 2^kX_1 + X_0 \\ Y &= 2^{3k}Y_3 + 2^{2k}Y_2 + 2^kY_1 + Y_0 \end{aligned}$$

we have

$$\begin{aligned} XY &= 2^{6k}X_3Y_3 \\ &+ 2^{5k}(X_2Y_3 + X_3Y_2) \\ &+ 2^{4k}(X_3Y_1 + X_2Y_2 + X_1Y_3) \\ &+ 2^{3k}(X_3Y_0 + X_2Y_1 + X_1Y_2 + X_0Y_3) \\ &+ 2^{2k}(X_2Y_0 + X_1Y_1 + X_0Y_2) \\ &+ 2^k(X_1Y_0 + X_0Y_1) \\ &+ X_0Y_0 \end{aligned} \tag{6.5}$$

Here we compute (using DSP blocks) the products

$$\begin{aligned} P_{33} &= X_3Y_3 \\ P_{22} &= X_2Y_2 \\ P_{11} &= X_1Y_1 \\ P_{00} &= X_0Y_0 \\ D_{32} &= (X_3 - X_2) \times (Y_3 - Y_2) \\ D_{31} &= (X_3 - X_1) \times (Y_3 - Y_1) \\ D_{30} &= (X_3 - X_0) \times (Y_3 - Y_0) \\ D_{21} &= (X_2 - X_1) \times (Y_2 - Y_1) \\ D_{20} &= (X_2 - X_0) \times (Y_2 - Y_0) \\ D_{10} &= (X_1 - X_0) \times (Y_1 - Y_0) \end{aligned}$$

and equation (6.5) may be rewritten as

$$\begin{aligned} XY &= 2^{6k}P_{33} \\ &+ 2^{5k}(P_{33} + P_{22} - D_{32}) \\ &+ 2^{4k}(P_{33} + P_{22} + P_{11} - D_{31}) \\ &+ 2^{3k}(P_{33} + P_{00} - D_{30} + P_{22} + P_{11} - D_{21}) \\ &+ 2^{2k}(P_{22} + P_{11} + P_{00} - D_{20}) \\ &+ 2^k(P_{11} + P_{00} - D_{10}) \\ &+ P_{00} \end{aligned} \tag{6.6}$$

Here we have only 10 multiplications instead of 16. Note that the recursive variant saves one more multiplication: It precomputes

$$D_{3210} = (X_3 + X_2 + X_1 + X_0) \times (Y_3 + Y_2 + Y_1 + Y_0)$$

instead of P_{30} and P_{21}, and computes the middle term $X_3Y_0 + X_2Y_1 + X_1Y_2 + X_0Y_3$ of equation (6.5) as a sum of P_{3210} and the other P_{ij}. However this poses several problems. Firstly, we have to use a smaller k (splitting in smaller chunks) to ensure P_{3210} doesn't overflow from the DSP size. Secondly, we currently estimate that the saved DSP is not worth the critical path degradation. Synthesis results of this implementation can be fond in Table 6.3.

6.3.5 N-part splitting

We now try to relate our multiplier expression to the visual tiling technique previously introduced using Figure 6.2. The purpose is to find a natural form for expressing directly (without recurrences) the product XY which allows an implementation suited for modern DSP blocks. We want this technique to scale to the 7-part splitting needed for the quadruple-precision floating-point multiplier.

We start with a $(N \times k) \times (N \times k)$ board tiled using $N \times N$ tiles of size $k \times k$-bit (where k is the DSP block multiplier size), as in Figure 6.4 for $N = 7$.

The black-square diagonal tiles will each be computed using one multiplier, for a total of N embedded multipliers. Next, each pair of tiles symmetric to this diagonal will be computed using two of the already computed products and only one additional product for a total of $N(N-1)/2$ multiplications. Using this technique the full product XY requires $N(N-1)/2+N = N(N+1)/2$ embedded multipliers.

The diagonal elements involved in computing the symmetric pair of tiles are those found at the intersection of the already tiled diagonal with a square connecting the tile pair (the dashed square in Figure 6.2). For a pair of tiles meant to compute $X_iY_j + X_jY_i$ with $i > j$, their contribution is:

$$X_iY_j + X_jY_i = P_{ii} + P_{jj} - \underbrace{DX_{ij} * DY_{ij}}_{D_{ij}}$$

Expressing the full product P basically consists in first expressing the contributions for each weight of k, and then summing up these contributions. The red lines in Figure 6.4 regroup the tiles whose contribution's weight is equal. The major components of the contribution are (we use the green line in Figure 6.4 as a running example)

- the diagonal tile's sub-product if the tile is intersected by the red line (P_{22} in our working example)
- the sum of contributions of the tiles on the red line; this also has two components:
  - a **positive** component formed by the sum of diagonal tiles onto which the red line is projected in the two directions: $P_{00} + P_{11}$ for the projection on Y and $P_{33} + P_{44}$ for the projection on X in Figure 6.4
  - a **negative** component comprising of the sum of all the products of differences corresponding to these tiles $D_{40} + D_{31}$

This sums-up the contribution for weight 2^{4k} to: $P_{22} + P_{00} + P_{11} + P_{33} + P_{44} - (D_{40} + D_{31})$. The full-expansion of all these contributions for a 7-part splitting is given below:

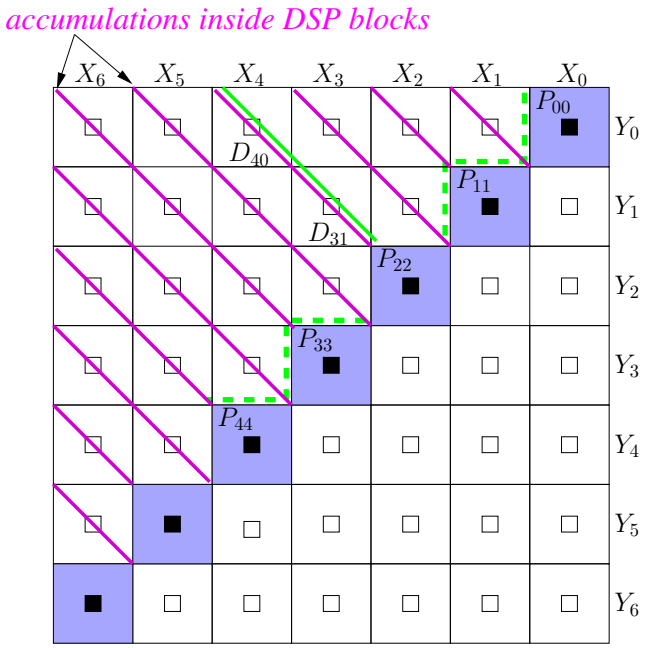
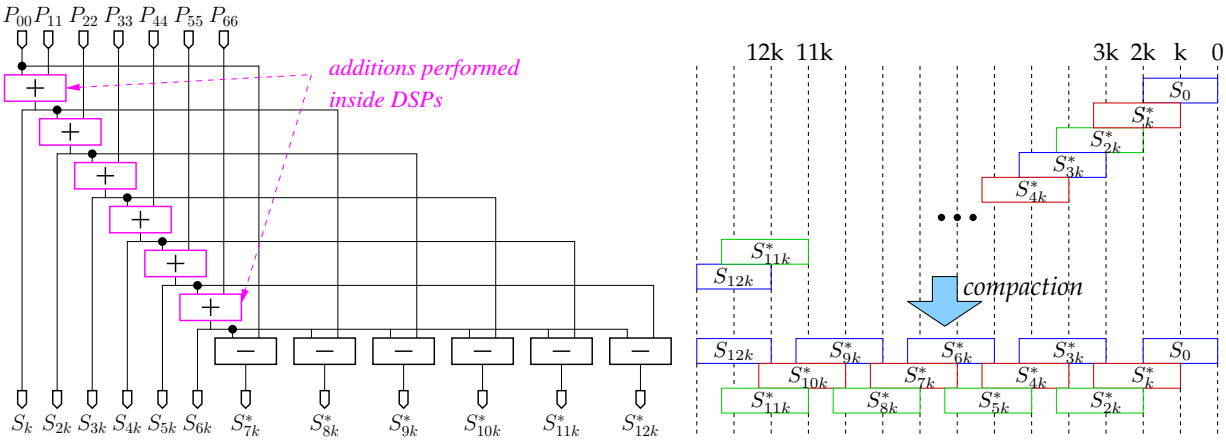


Figure 6.4 *119x119bit multiplier using Virtex-4 DSP48 for QP mantissa multiplier*

$$
\begin{aligned}
XY = \; & P_{00}+ \\
& 2^{k}(P_{00}+P_{11}-D_{10})+ \\
& 2^{2k}(\underbrace{\underbrace{P_{00}+P_{11}+P_{22}}_{S_{2k}}-D_{20}}_{S^{*}_{2k}})+ \\
& 2^{3k}(P_{00}+P_{11}+P_{22}+P_{33}-(D_{30}+D_{21}))+ \\
& 2^{4k}(P_{00}+P_{11}+P_{22}+P_{33}+P_{44}-(D_{40}+D_{31}))+ \\
& 2^{5k}(P_{00}+P_{11}+P_{22}+P_{33}+P_{44}+P_{55}-(D_{50}+D_{41}+D_{32}))+ \\
& 2^{6k}(P_{00}+P_{11}+P_{22}+P_{33}+P_{44}+P_{55}+P_{66}-(D_{60}+D_{51}+D_{42}))+ \\
& 2^{7k}(P_{11}+P_{22}+P_{33}+P_{44}+P_{55}+P_{66}-(D_{61}+D_{52}+D_{43}))+ \\
& 2^{8k}(P_{22}+P_{33}+P_{44}+P_{55}+P_{66}-(D_{62}+D_{53}))+ \\
& 2^{9k}(P_{33}+P_{44}+P_{55}+P_{66}-(D_{63}+D_{54}))+ \\
& 2^{10k}(P_{44}+P_{55}+P_{66}-D_{64})+ \\
& 2^{11k}(P_{55}+P_{66}-D_{65})+ \\
& 2^{12k}P_{66}
\end{aligned}
$$

On Virtex4 devices, the sum of negative contributions can be performed entirely inside the DSP blocks. On Stratix devices, some of these additions can as well be pushed inside the DSP blocks. The sum of positive contributions can be computed using the circuit in Figure 6.5(a). On Virtex devices, the sum $P_{00} + ... + P_{66}$ is computed constructively inside the DSP blocks, so that the positive contribution of the first N terms of the final sum brings no logic overhead. Moreover, in order to compute the positive contribution from the rest, an extra $N-1$, $2k+g$-bit subtracters are needed. The number of guard bits g is chosen such that $\sum_{i=0}^{k-1} P_{ii}$ does not overflow.

Finally, one needs to sum-up all these contributions. We can exploit the fact that these contributions each have a specific weight and a maximum length $2k+g$, smaller than $3k$. Consequently, we compact these contributions into 3 operands as presented in Figure 6.5(b). On Stratix devices, this addition can take advantage of the hardware support for 3-operand adders and can therefore



(a) Sub-product weight contributions

(b) Final weighted contributions compacted

Figure 6.5 *119x119-bit Karatsuba*

	Target	Latency	Freq.	Slices	DSPs	Bits
K-O-4	Virtex-4	15	370	918	10	68
K-O-5	Virtex-4	14	325	1272	16	85
K-O-6	Virtex-4	16	323	1655	21	102
K-O-7	Virtex-4	18	322	2053	28	119
[142]		0	76	1100	49	
K-O-7	Stratix-II	22	227	2569 ALUT, 9832 REG	56 9-bit (28 18-bit)	
lpm_mult		22	73	454 ALUT, 594 REG, 7 M4K	122 9-bit (61 18-bit)	
K-O-7	Stratix-III	16	312	2466 ALUT, 7817 REG	42 18-bit	
lpm_mult		22	136	483 ALUT, 2549 REG, 4 M9K	62 18-bit	
K-O-15	Virtex4	35	312	6624	121	255

Table 6.3 *Synthesis results of large Karatsuba multipliers. For Stratix-II/III we used the* `lpm_mult` *megafunction provided with the Megawizerd tool for generating binary multipliers*

reduce implementation cost.

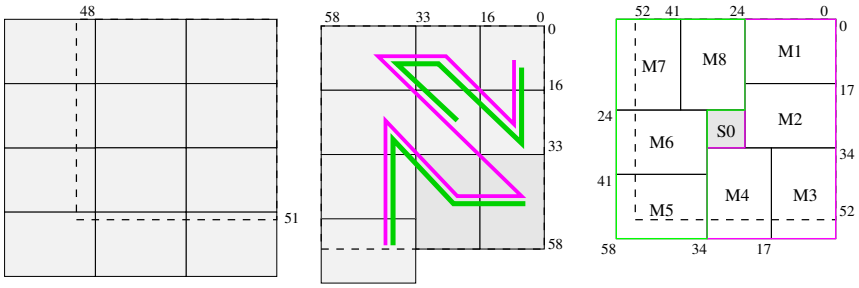
Table 6.3 presents synthesis results of large multipliers built using this technique. The highlight of this table is the 119-bit multiplier, suited for the mantissa multiplier in quadruple-precision. The number of DSPs, when compared to a standard implementation is reduced from 49 to 28. We acknowledge that Montgomery's study [121] lowers the number of multiplications to 22, however, some of them exceed the embedded multiplier's size and the circuit has much less regularity.

6.3.6 Issues with the most recent devices

The Karatsuba-Ofman algorithm is useful on Virtex-II to Virtex-4 as well as Stratix-II devices, to implement single and double precision floating-point multiplication.

The larger (36 bit) DSP block granularity (see Section 2.1.2) of Stratix-III and Stratix-IV are not as well suited to this algorithm as they prevent us from using the 18x18 bit product twice. However, for larger values of N ($N = 7$ for quadruple-precision) some of the contributions may still be pushed inside the DSPs, lowering the total multiplier count, as Table 6.3 shows.

On Virtex-5 devices, the Karatsuba-Ofman algorithm can be used if each embedded multiplier is considered as a 18x18 one, which is suboptimal. For instance, single precision K-O requires 3



(a) standard tiling (b) Logicore tiling (c) our tiling

Figure 6.6 *53-bit multiplication using Virtex-5 DSP48E. The dashed square is the 53x53 multiplication.*

DSP blocks, where the classical implementation consumes 2 blocks only. Nevertheless, as operand width increases, the DSP savings are still visible on this architecture, for example 119-bit wide multipliers can be implemented using 28 DSPs whereas the best implementation we found while maximizing DSP usage took 34 DSPs. However, we still have to find a variant of Karatsuba-Ofman that exploits the 18x25 multipliers to their full potential.

We now present an alternative multiplier design technique specific to Virtex-5/-6 devices but which can also reduce implementation cost on other platforms, such as the Virtex-4 and StratixII-IV devices.

6.4 Non-standard tilings

This section optimizes the use of the Virtex-5 25x18 signed multipliers. In this case, X has to be decomposed into 17-bit chunks, while Y is decomposed into 24-bit chunks. Indeed, in the Xilinx LogiCore Floating-Point Generator, version 3.0, a double-precision floating-point multiplier consumed 12 DSP slices (see Figure 6.6(a)): X was split into 3 24-bit subwords, while Y was split into 4 17-bit subwords. This splitting would be optimal for a 72x68 product, but quite wasteful for the 53x53 multiplication required for double-precision, as illustrated by the dashed square indicating the DP mantissa multiplier board from Figure 6.6(a).

In version Floating-Point Generator version 4.0, and in LogiCore multiplier starting with version 11.0, DSP blocks are arranged in a different way, detailed in [21, p.78], and illustrated in Figure 6.6(b). This new arrangement has the advantage that although four of multipliers are used in 17x17-bit mode, all multipliers can be cascaded as indicated by the red line in Figure 6.6(b). All the additions may be performed within the DSP blocks but some additional shift-registers are needed in order to synchronize the I/O in deeply pipelined implementations. Again, some of these shifters (at most 4 levels) can be packed inside the DSPs. This approach exploits no parallelism and therefore has a very long latency. This latency can be reduced by breaking in two the cascade chain and using a pipelined adder to sum the two contributions (Figure 6.6(b), green line).

The following equation presents an original way of implementing double-precision (actually up to 58x58) multiplication, using only eight 18x25 multipliers, whereas the Logicore version uses ten.

$$\begin{array}{rcl|l} XY & = & X_{0:23}Y_{0:16} & \text{(M1)} \\ & + & 2^{17}(X_{0:23}Y_{17:33} & \text{(M2)} \\ & + & 2^{17}(X_{0:16}Y_{34:57} & \text{(M3)} \\ & + & 2^{17}X_{17:33}Y_{34:57})) & \text{(M4)} \\ & + & 2^{24}(X_{24:40}Y_{0:23} & \text{(M8)} \\ & + & 2^{17}(X_{41:57}Y_{0:23} & \text{(M7)} \\ & + & 2^{17}(X_{34:57}Y_{24:40} & \text{(M6)} \\ & + & 2^{17}X_{34:57}Y_{41:57}))) & \text{(M5)} \\ & + & 2^{48}X_{24:33}Y_{24:33} & \end{array} \tag{6.7}$$

The reader may check that each multiplier is a 17x24 one except the last one. The proof that Equation (6.7) indeed computes $X \times Y$ consists in considering

$$X \times Y = (\sum_{i=0}^{57} 2^i x_i) \times (\sum_{j=0}^{57} 2^j y_j) \; = \sum_{i,j \in \{0 \ldots 57\}} 2^{i+j} x_i y_j$$

and checking that each partial bit product $2^{i+j}x_iy_j$ appears once and only once in the right-hand side of Equation (6.7). This is illustrated by Figure 6.6(c).

The last line of Equation (6.7) is a 10x10 multiplier (the white square at the center of Figure 6.6(c)). It could consume an embedded multiplier, but due to its small size it is probably best implemented as logic.

We have parenthesized the Equation (6.7) in order to make full use of the Virtex-5 internal DSP adders (see page 10). Due to the fixed 17-bit shifts between the operands, the sub-sums corresponding the red tiles and those corresponding to the green tiles are computed entirely using DSP block resources. This reduces the number of inputs of the final multi-operand adder to three.

Such a parenthesing involving only 17-bit shifts is graphically descried as a *super-tile*. Figure 6.7 shows some super-tiles corresponding to the DSP capabilities of Virtex-4 and Virtex-5/6. These super-tiles (and all their subsets) don't require additional hardware to perform the full product. In addition, larger super-tiles can be obtained by coupling the black and white circles of adjacent super-tiles. This corresponds to using the cascading adder input of the DSP blocks. Actually, all the possible super-tiles may be generated by the primitives shown on Figure 6.8.

On Stratix, the large adders inside the DSP block that can be used to add up to four 18x18-bit partial products having the same magnitude. This corresponds to a line of tiles parallel to the main diagonal. However, as previously stated, we are currently unable to obtain the predicted performance out of the Altera Quartus tools. This could be solved by using Alter-specific primitives, but would require much more development work and would break portability.

6.4.1 Design decisions

In the previous example, there remains an untiled 10-bit $\times$ 10-bit square. Should this be implemented as logic, or as an underutilized DSP block? This is a trade-off between logic and DSP blocks, and as such the decision should be left to the user. We have therefore decided to offer the user the possibility to select a ratio between DSP count and logic-consumption. This *ratio* is as a number in the $[0, 1]$ range. Larger values for the ratio favour DSP oriented architecture whereas lower values favor logic-oriented architectures. The total number of multipliers used is a function of the input widths, ratio and FPGA target.

In order to exploit this user-provided ratio accurately, we have modeled the logical equivalence of a DSP block for various FPGA families, inside FloPoCo's `Target` hierarchy.

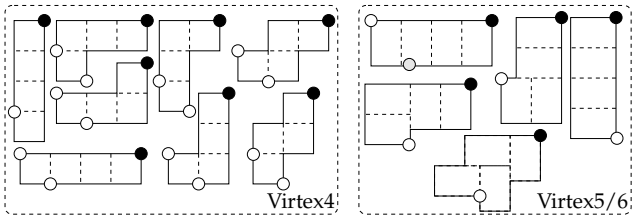


Figure 6.7 *Some super-tiles exactly matching DSP blocks*

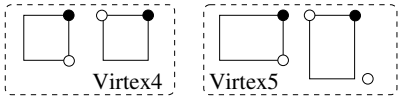


Figure 6.8 *Super-tiling primitives*

6.4.2 Algorithm

The construction of a tentative multiplier configuration consists of three steps.

1. Generate a valid partition of the large multiplication into smaller partial products or tiles.
2. Group these tiles as super-tiles in order to reduce the number of operands of the large multiplier's final adder. The super-tiles are built using the regrouping primitives presented in Figure 6.8. Two successive tiles can be regrouped if their their black and white circles correspond to one of the regrouping primitives. When building super-tiles we also balance their sizes in order to reduce operator pipeline depth and the number of synchronization registers.
3. Compute the approximate cost of the configuration. This cost includes: the DSPs, the slices needed for computing the rest of the multiplication, and the cost of the multioperand adder used to compute the final result.

Configurations may be compared according to this cost. The best one will be chosen, and its VHDL generated.

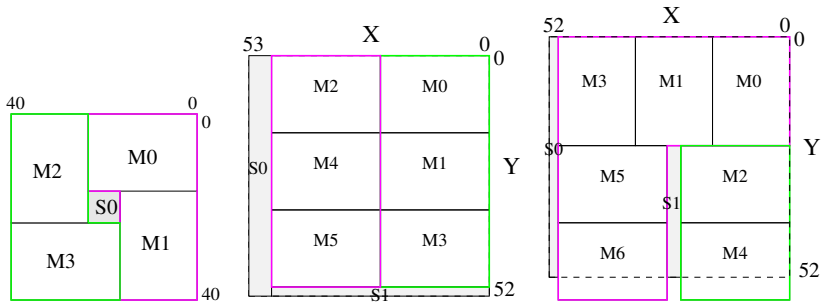
Choosing among *all* possible configurations takes an exponential number of steps with respect to the size of the multiplication board $O((u \times v)^{\delta})$, where u and v are the dimensions of the multiplication and δ is the number of DSPs. Although this would ensure we find the optimal configuration, the exponential complexity prevents from obtaining results in reasonable time. Hence, we prune exploration branches using the following criteria:

- Tiles do not overlap. In step 1, we only consider tilings which align tile edges. This reduces the number of tilings to O(2^{δ}) for Virtex4 and O(3^{δ}) for Virtex5.
- Configurations symmetrical to already existing ones are pruned.
- Configurations where large holes appear inside the tiling are also pruned.

6.4.3 Reality check

We have used the presented algorithm in order to generate mantissa multipliers for DP (53bit) and QP (113bit) floating-point. Table 6.4 presents the synthesis results obtained these mantissa multiplier on Virtex5 (xc5vfx100T-3-ff1738) FPGAs using Xilinx ISE 11.4. The results of this work are compared to Xilinx Logicore core generator and combinatorial results obtained from [142].

Our implementation offers a wide range of user-defined trade-offs between DSP, logic and latency. Our automatically generated multipliers provide better performance than the handcrafted



(a) 41-bit, 4 DSPs (b) 53-bit, 6 DSPs (c) 53-bit, 7 DSPs

Figure 6.9 *Various tilings of large multipliers*

Table 6.4 *Comparison of multiplier implementations on Virtex5 devices. All our implementations are targeted at 400MHz. Frequency is expressed in MHz*

w	Tool	**Multiplier $w \times w$-bit**			
		cycles	Freq.	Logic	DSP
41	ours Fig.6.9(a)	6	424	363LUT 342REG	4
53	ours Fig.6.9(b)	8	382	1143LUT 1052REG	6
53	ours Fig.6.9(c)	9	410	792LUT 853REG	7
58	ours Fig.6.6(c)	8	410	454LUT 751REG	8
53	[142]	0	111	200LUT	12
53	Logicore Fig.6.6(b)	12	450	229LUT 280REG	10
113	ours	13	407	2070LUT 2062REG	34
113	ours (Karatsuba)	18	367	3134LUT 2435REG	29
113	[142]	0	90	1000LUT	35

ones from [142], while also reducing DSP count. The biggest difference is for DP, where their decomposition technique infers 12 DSPs, out of which several are underutilized. With respect to Xilinx Logicore, our implementation saves DSP blocks without big penalties in logic consumption.

Unfortunately, the tiling technique proves less effective as the multiplier size increases. For these large multiplier sizes, the Karatsuba technique manages to better reduce DSP count, even thought multipliers are underutilized (used in 17x17 mode).

6.5 Squarers

The bit-complexity of squaring is roughly half of that of standard multiplication. Indeed, we have the identity:

$$X^2 = (\sum_{i=0}^{n-1} 2^i x_i)^2 = \sum_{i=0}^{n-1} 2^{2i} x_i + \sum_{0<i<j<n} 2^{i+1} x_i$$

This is is only useful if the squarer is implemented as LUTs. However, a similar property holds for a splitting of the input into several subwords:

$$(2^k X_1 + X_0)^2 = 2^{2k} X_1^2 + 2 \cdot 2^k X_1 X_0 + X_0^2 \tag{6.8}$$

$$\begin{aligned}(2^{2k}X_2 + 2^kX_1 + X_0)^2 &= 2^{4k}X_2^2 + 2^{2k}X_1^2 + X_0^2 \\ &+ 2 \cdot 2^{3k}X_2X_1 \\ &+ 2 \cdot 2^{2k}X_2X_0 \\ &+ 2^kX_1X_0\end{aligned} \tag{6.9}$$

Computing each square or product of the above equation in a DSP block, yields a reduction of the DSP count from 4 to 3, or from 9 to 6. Besides, this time, it comes at no arithmetic overhead.

6.5.1 Squarers on Virtex-4 and Stratix-II

Now consider $k = 17$ for a Virtex-4 implementation. Looking closer, it turns out that we still lose something using the above equations: The cascading input of the DSP48 and DSP48E is only able to perform a shift by 17. We may use it only to add terms whose weight differs by 17. Unfortunately, in equation (6.8) the powers are 0, 18 and 34, and in equation (6.9) they are 0, 18, 34, 35, 42, 64.

One more trick may be used for integers of at most 33 bits. Equation (6.8) is rewritten

$$(2^{17}X_1 + X_0)^2 = 2^{34}X_1^2 + 2^{17}(2X_1)X_0 + X_0^2 \tag{6.10}$$

and $2X_1$ is computed by shifting X_1 by one bit before inputting it in the corresponding DSP. We have this spare bit if the size of X_1 is at most 16, *i.e.* if the size of X is at most 33. As the main multiplier sizes concerned by such techniques are 24 bit and 32 bit, the limitation to 33 bits is not a problem in practice.

Table 6.5 provides synthesis results for 32-bit squares on a Virtex-4. Such a squarer architecture can also be fine-tuned to the Stratix II-family.

6.5.2 Squarers on Stratix-III and Stratix-IV

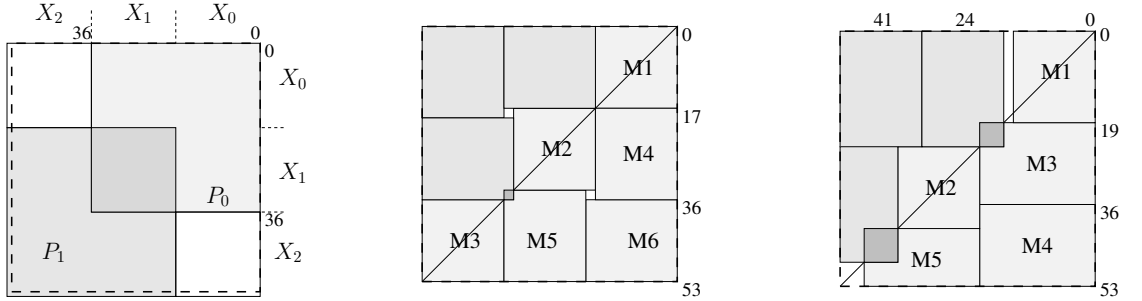
On the most recent Altera devices, the 36-bit granularity means that the previous technique begins to save DSP blocks only for very large input sizes.

We now present an alternative way of implementing a double-precision (53-bit) squarer on such devices using only two 36x36 half-DSPs, where a standard multiplier requires four on a Stratix-III and two and a half on a Stratix-IV. It exploits the fact that, although the addition structure of the four 18x18 sub-multipliers is fixed, their inputs are independent.

The two 36x36 multipliers we need are illustrated on Figure 6.10(a) (P_0 and P_1). P_0 is completely standard and computes the sub-square $X_{35:0}X_{35:0}$. The bottom-left one (labeled P_1) is configured as a multiplier, too, but it doesn't need to recompute and add the sub-product $X_{35:18}X_{35:18}$ (the dark square in the center), which was already computed by the previous multiplier. Instead, this sub-multiplier will complete the 53-bit square by computing $2X_{17:0}X_{52:36}$ (the sum of the two

	Latency	Frequency	Slices	DSPs	bits
LogiCore	6	489	59	4	32
LogiCore	3	176	34	4	
Squarer	3	317	18	3	
LogiCore	18	380	279	16	53
LogiCore	7	176	207	16	
Squarer	7	317	332	6	

Table 6.5 *32-bit and 53-bit squarers on Virtex-4 (4vlx15sf676-12)*



(a) DP Squarer for StratixIII/IV (b) DP Squarer Virtex-5/6 (i) (c) DP Squarer Virtex-5/6 (ii)

Figure 6.10 *Double-precision squaring. Tilings for StratixIII/IV and Virtex-5/6 devices*

white squares), which has the same weight 2^{36}. To this purpose, the inputs of the corresponding 18x18 sub-multiplier have to be set as $X_{0:17}$ and $2X_{52:36}$. The latter will not overflow, because a double-precision significand product is 53x53 and not 54x54, therefore we have $X_{53} = 0$. The corresponding squarer equation is given below:

$$X^2 = \underbrace{X_{10}X_{10}}_{P_0} + 2^{36} \underbrace{(2X_2X_0 + 2^{18}(X_2X_1 + X_1X_2) + 2^{36}X_2X_2)}_{P_1}$$

Applied to a single 36x36 block, a similar technique allows us to compute squares up to 35x35 using only three of the four 18x18 blocks. The fourth block is unusable, but this may reduce power consumption.

We were not able to verify these designs experimentally. Low level access to the DSP blocks is possible only through Altera Megafunctions which currently don't implement our desired functional mode.

6.5.3 Non-standard tilings on Virtex-5/6

Figures 6.10(b),6.10(c) illustrate non-standard tilings for double-precision square using six or five 24x17 multiplier blocks. These tilings are symmetrical with respect to the diagonal, so that each symmetrical multiplication may be computed only once. However, there are slight overlaps on the diagonal: the darker squares are computed twice, and therefore the corresponding subproduct must be removed. These tilings are designed in such a way that all the smaller subproducts may be computed in LUTs at the peak DSP frequency.

Note that a square multiplication on the diagonal of size n, implemented as LUT, should consume only $n(n+1)/2$ LUTs instead of n^2 thanks to symmetry.

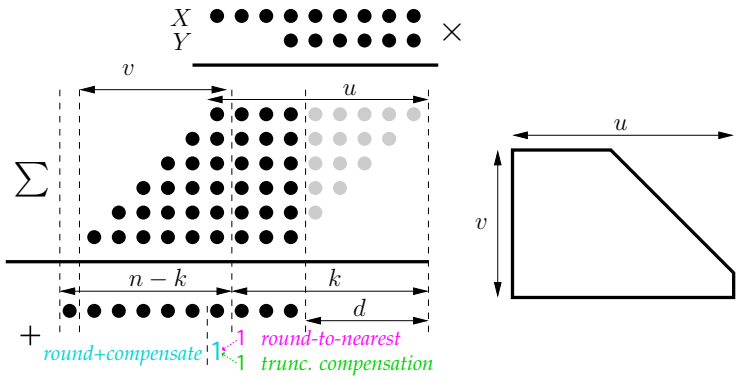
We currently do not have implementation results. It is expected that implementing such equations will lead to a large LUT cost, partly due to the many sub-multipliers, and partly due to the irregular weights of each line (no 17-bit shifts) which may prevent optimal use of the internal adders of the DSP48E blocks.

6.6 Truncated multipliers

Truncated multipliers reduce resources, delay, or power consumption [157, 138] for a well-controlled accuracy degradation. Let us consider two integers X and Y on u and v bits respectively with XY on $n = u + v$ bits. The idea is to save the computation of some of the less significant columns in the multiplication array (see the greyed-out rows in Figure 6.11(a)) so that the error of the integer multiplication remains small enough. More precisely, given an integer k, we build

Table 6.6 *Truncated multipliers providing faithful rounding for common floating point formats*

Precision	k	**Discarded** d
Single	23	18
Double	52	46
Quadruple	112	105



(a) truncated multiplication (b) truncated board

Figure 6.11 *Truncated multiplication and the corresponding tiling multiplication board*

a multiplier that returns a result faithfully rounded on $n-k$ bits, meaning that the final error between the result of the full multiplier and the truncated multiplier is smaller than 2^k.

6.6.1 Faithfully accurate multipliers

Let us first determine the maximum number of columns, denoted by d, that may be removed (see Figure 6.11(a)).

The error E_{total} has two components:

$$E_{\text{total}} = E_{\text{approx}} + E_{\text{round}},$$

where E_{approx} is the approximation error introduced by the truncation of the d columns, and E_{round} is the error of rounding the $n-d$-bit intermediate result to $n-k$ bits.

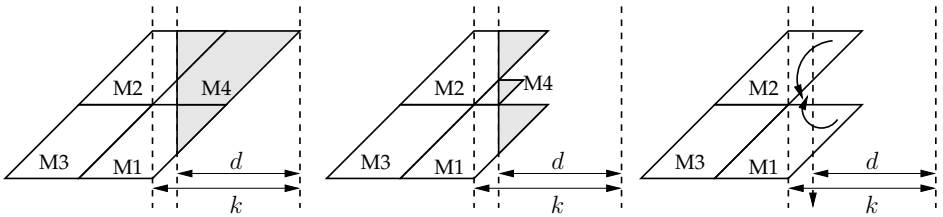
To ensure that $E_{\text{total}} \leq 2^k$, we need to distribute our 2^k error budget between the two error sources. Firstly, the E_{round} can be bounded by 2^{k-1} when implementing round-to-nearest. From the implementation perspective, this reduces to adding a 1 in position 2^{k-1} (highlighted in Figure 6.11(a))

Secondly, the error budget for E_{approx} is now 2^{k-1}. The sum of the first d discarded columns is in the interval:

$$0 \leq E_{\text{approx}} \leq \sum_{i=1}^{d} i2^{i-1} = (d-1)2^d + 1$$

An offset correction bit in position 2^{k-1} can reduce this error by almost half by centering it [157] (the green '1' in Figure 6.11(a)). Combined with the previous constraint $E_{\text{approx}} < 2^{k-1}$, this provides us with a relation of the form $d = f(k)$. Table 6.6 shows how the number of discarded columns varies for common floating point formats.

Truncated multipliers can effectively be used for implementing the mantissa multiplication in floating-point multipliers: if no IEEE-754 compliance is mandatory, or data-paths includes the evaluation of elementary functions. Moreover, fixed-point pipelines can also take benefit from this



(a) wasteful (b) better (c) compensated

Figure 6.12 *Truncation applied to multipliers. Left: Classical truncation technique applied to DSPs. Center: Improved truncation technique; M4 is computed using logic. Right: FPGA optimized compensation technique; M4 is not computed.*

techniques. For example, truncated multiplications together with the proper approximation-error estimation techniques will be used to reduce the number of DSP blocks for large precision Horner polynomial evaluation scheme in Chapter 7.

6.6.2 FPGA fitting

The theoretical saves in complexity entailed by truncated multiplications approaches 50%. The entailed savings have two components: the size of the sub-products non computed and the size reduction of the operands in the multioperand reduction scheme. The truncation technique applied to a multiplication performed using DSP blocks is presented in Figure 6.12(a). The architecture consumes 4 DSPs to compute the sub-products M1-M4. The grayed out parts of these sub-products are then discarded before performing the final addition. Out of the 4 DSPs used, 2 are softly underutilized (M1 and M2) and one is greatly underutilized (M4). A better architecture that performs M4 in logic is presented in figure 6.12(b). This architecture saves one DSP block at the expense of the logic used to perform M4, which can be itself truncated.

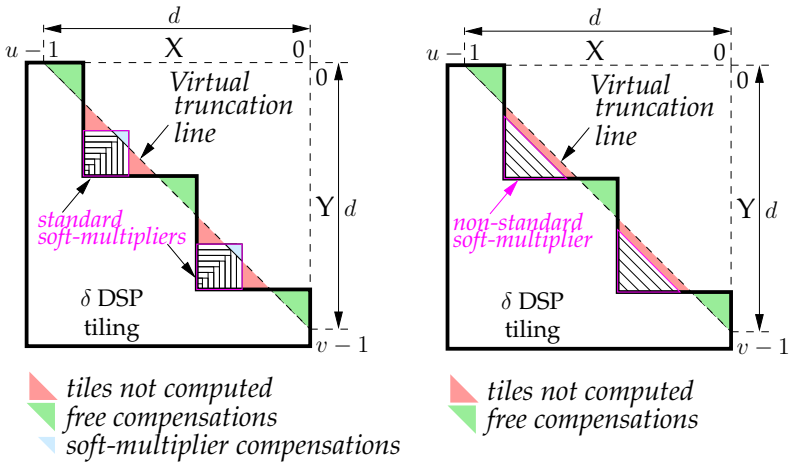
However, on both Figure 6.12(a) and 6.12(b), the monolithic DSP blocks compute all the bits of M1 and M2. As these bits come for free, we may take them into account, as it will reduce E_{approx} and possibly allow us to increase d. This requires adders extending beyond $n - d$, but those come for free if they are inside the DSP blocks.

We therefore want to tile the truncated multiplier such that the error entailed by discarding the untiled part meets the previously defined error budget. In this way, the bits not computed at the left of k will be compensated by the ones computed at the right, as illustrated on Figure 6.12(c).

6.6.3 Architecture generation algorithm

A two phase algorithm was implemented in order to generate truncated multiplier using the previously presented tiling technique. The first phase tiles the multiplication board starting from bottom left using $\delta = \lfloor \text{Area}_{\text{board}}/\text{Area}_{\text{tile}} \rfloor$ DSPs where $\text{Area}_{\text{board}}$ is the area of a multiplication board similar in shape to that in Figure 6.11(b) (size is dependent on k) and $\text{Area}_{\text{tile}} = \alpha \times \beta$. By construction, the approximation error of this tiling, $E_{\text{approx}} \geq 2^{k-1}$.

The second phase reduces E_{approx} so that it becomes smaller than 2^{k-1}. In order to do this, we rely on pipelined soft-core multipliers (pipelined multipliers using logic-only). E_{approx} can be reduced by tiling some high-weighted yet untiled bits. Taking Figure 6.13(a) as running example, these are the untiled bits situated further away (Euclidean distance) from the origin (top right corner). This is an iterative process which ends when the error bound is reached. Each iteration consists in finding the furthest untiled point from the origin: if this point is adjacent to an existing soft-core multiplier, it increases the respective dimension of this multiplier by 1 (illustrated in



(a) our implementation (b) ideal implementation

Figure 6.13 *Tiling truncated multiplier using DSPs and soft-core multipliers*

Table 6.7 *Truncated multiplier results*

FPGA	Prec.	Latency, Freq.	Resources
Virtex5	DP	6 cycles @ 414MHz	320LUT 302REG 5DSP
	QP	20 cycles @ 334MHz	2497LUT 2321REG 19DSP
	QP	14 cycles @ 245MHz	2249LUT 1576REG 19DSP
Virtex4	DP	11 cycles @ 368MHz	358sl. 7DSP
	QP	21 cycles @ 368MHz	1735sl. 26DSP

Figure 6.13(a)); otherwise, an 1×1 bit soft-core multiplier is instantiated at that point. There is a small inefficiency here in using standard multipliers as they would eventually end-up tiling some of the lower-weighted points (top-right corner) whereas some higher-weighted points are still untiled. One would better use in this situation non-standard multipliers 6.13(b). These can eventually decrease the soft-multiplier size but require more work to implement and integrate so it is left for future work. Now back to soft-multiplier tiling, once the process finishes, a post-processing phase replaces large soft-core multipliers by DSP blocks.

Figure 6.13(a) shows how the size these soft-core multipliers increases. When a valid configuration is met, its hardware cost is evaluated, and stored if minimal. If possible, a new tiling is explored and cost is re-evaluated.

We remark that with respect to the classical truncation algorithm, not all the bits at the left of the virtual truncation line are computed. In fact, the bits computed for free at the right of this line compensate them. The extra cost of this architecture comes from the few extra bits of the operands in the final multi-operand addition.

Figure 6.14 shows some possible tilings for high precision truncated multipliers. Table 6.7 presents synthesis results for DP and QP. Using our improved truncated multiplier technique we are able to significantly reduce the number of DSPs with respect to classical multiplications. For example, on Virtex4 for DP we are able to reduce DSP count from 10 to 7 DSPs while also reducing slice count, and for QP we reduce from 49 to 26 at without any slice penalty. On Virtex5, the reductions are from 6 to 5 for and roughly half the LUTs and REGs for DP and from 34 to 19 at a small increase in logic resources for QP.

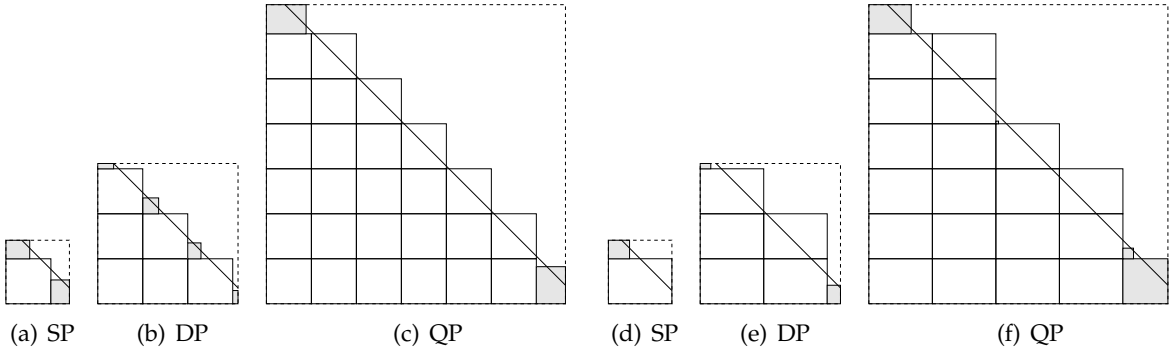


Figure 6.14 *Mantissa multipliers for SP,DP,QP, Virtex4 (left) and Virtex5 (right) ensuring faithful rounding. The gray tiles represent soft-core multipliers*

6.7 Conclusion

We have presented in this chapter several situations where the DSP resources can be saved by by exploiting the flexibility of the FPGA target. An original family of multipliers for Virtex-5/6 is also introduced, along with original squarer architectures. FPGA-specific implementations of truncated multipliers are also presented. These manage to substantially reduce DSP cost and help reduce the cost of implementing high-performance polynomial evaluators, as those presented in Chapter 7.

Most of the presented architectures, aside from DSP reduction, also enable a possible reduction in latency. In the case of multipliers implemented on Xilinx devices, one can trade a longer latency for significantly less resources, mostly due to the cascading opportunities of the DSP-blocks combined with the SRL slice configurations. We believe that the best decision to be made is dependent on the context, and therefore this should be left as a user knob.

Moreover, we believe that the place of some of these algorithms is in vendor core generators and synthesis tools, where they will widen the space of implementation trade-off offered to a designer.

Thanks

Part of the material presented in this chapter is based on collaborations with Sebastian Banescu and Radu Tudoran, students at the Technical University of Cluj-Napoca, whom I had the pleasure to work with during their summer internship in the Arenaire team. I gratefully acknowledge their contributions.

CHAPTER 7

Polynomial-based architectures for function evaluation

Function evaluation is the implementation bottleneck of computational-bound scientific computations: over 60% of time is time is spent evaluating functions for a jet-engine simulation in [127], SPICE circuit simulations based on electronic component modeling make extensive use of functions [94] (see Figure 3.4 for arithmetic operation distribution in modeling the basic circuit components).

In this chapter, we consider real functions $f(x)$ of one real variable x, and we are interested in a fixed-point implementation of this function over some interval. We will implement composed functions as a fused operator, rather than a composition of successive operators. We assume that f is continuously differentiable over this interval up to a certain order. There are many examples where the hardware implementation of such functions is required. The following list should not, in any case, be considered exhaustive:

- Fixed-point sine, cosine, exponential and logarithms are routinely used in signal processing algorithms.
- Random number generators with a Gaussian distribution may be built using the Box-Muller method, which requires logarithm, square root, sine and cosine [106]. Arbitrary distributions may be obtained by the inversion method, in which case one needs a fixed-point evaluator for the inverse cumulative distribution function (ICDF) of the required distribution [52]. There are as many ICDF as there are statistical distributions.
- Approximations of the inverse $1/x$ and inverse square root $1/\sqrt{x}$ functions are used in recent floating-point units to bootstrap division and square root computation [119].
- $f_{\log}(x) = \log(x + 1/2)/(x - 1/2)$ over $[0, 1]$, and $f_{\exp}(x) = e^x - 1 - x$ over $[0, 2^{-k}]$ for some small k, are used to build hardware floating-point logarithm and exponential in [72].
- $f_{\cos}(x) = 1 - \cos\left(\frac{\pi}{4}x\right)$, and $f_{\sin}(x) = \frac{\pi}{4} - \frac{\sin\left(\frac{\pi}{4}x\right)}{x}$ over $[0, 1]$, are used to build hardware floating-point trigonometric functions in [71].
- $s_2(x) = \log_2(1 + 2^x)$ and $d_2(x) = \log_2(1 + 2^x)$ are used to build adders and subtracters in the Logarithm Number System (LNS), and many more functions are needed for Complex LNS [37].

Many function-specific algorithms exist, for example variations of the CORDIC algorithm provide low-area, long-latency evaluation of most elementary functions [123]. However, these implementations lead to substantial non-reusable work for obtaining a functional implementation. The work presented in this chapter focuses on a *generic* implementation method which not only works well for a large class of functions, but is also well suited to the architecture of modern FPGAs containing many embedded multipliers. The generic operator for this implementation has been developed and integrated in the FloPoCo framework, as Figure 7.1 presents. This operator

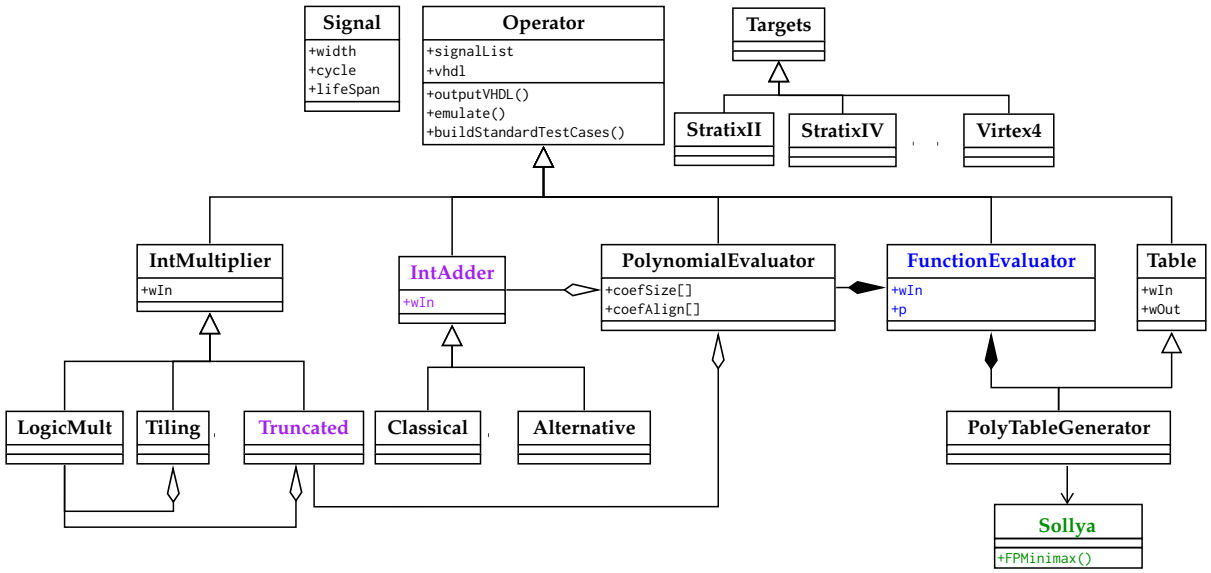


Figure 7.1 *FloPoCo class structure integrating the generic fixed-point FunctionEvaluator*

makes extensive use of other FloPoCo components such as pipelined binary adders and truncated multipliers whose architecture was described in the previous chapters.

The work presented in this chapter will facilitate the implementation of a full hardware mathematical library (libm) in FloPoCo. Next chapters will illustrate the first steps in this direction: designing of the floating-point $\sqrt{x}$ and e^x operators. Although some specific-FPGA optimizations are presented here, most of the methodology is independent of the FPGA target and could apply to other hardware targets such as ASIC circuits. This would need adding an ASIC target to the FloPoCo target hierarchy.

7.1 Related work

There have been an important numbers of articles published on polynomial evaluators. We focus here on those which describe *generic* methods as a fair comparison with work described next.

Several table-based, multiplier-less methods for linear (or degree-1) approximation have evolved from the original paper by Sunderland et al [147]. See [60] or [123] for a review. These methods have very low latency but do not scale well beyond 20 bits: the table sizes scale exponentially, and so does the design-space exploration time.

The High-Order Table-Based Method (HOTBM) by Detrey and Dinechin [69] extended the previous methods to higher-degree polynomial approximation. An open-source implementation is available in FloPoCo. In its current version, the generated architectures fit poorly recent FPGAs with powerful DSP blocks, and don't scale beyond 32 bits. Retargetting it would require considerable effort. Additionally, HOTBM focuses on parallel polynomial evaluation whereas, in this work, we use a sequential evaluation which reduces implementation cost at the expense of latency.

Lee et al [105] have published many variations on a generic datapath optimization tool called MiniBit to optimize polynomial approximation. They use ad-hoc mixes of analytical techniques such as interval analysis, and heuristics such as simulated annealing to explore the design space. However, the design space explored in these articles does not include the architectures we describe in this work: All the multipliers in these papers are larger than strictly needed, therefore they miss

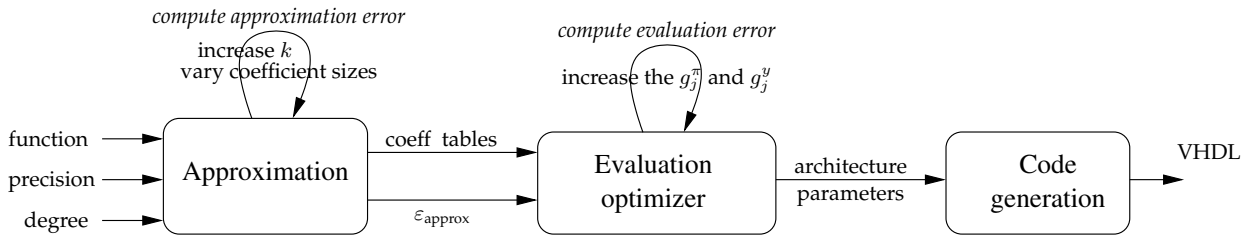


Figure 7.2 *Automated implementation flow*

the optimal. In addition, this tool is closed-source and difficult to evaluate from the publications, in particular it is unclear if it scales beyond 32 bits.

Tisserand studied the optimization of low-precision (less than 10 bits) polynomial evaluators [149]. He finetunes a rounded minimax approximation using an exhaustive exploration of neighboring polynomials. He also uses other tricks on smaller (5-bit or less) coefficients to replace the multiplication with such a coefficient by very few additions. Such tricks do not scale to larger precisions.

Compared to these publications, the work presented next has the following distinctive features:

- it *scales* to precisions of 64 bits or more, while being equivalent or better than the previous approaches for smaller precisions.
- it uses *minimax polynomials* provided by the Sollya tool [1] for polynomial approximation, which is the state-of-the-art for this application, as detailed in Section 7.2.2.
- it attempts to *reduce the number of embedded-multipliers* used. On one hand we attempt to minimize coefficient sizes (as others in the literature do as well). On the other hand, we trim the evaluation data-path to the bare minimum of bits that are needed at each step. We integrate the truncated multipliers introduced in Chapter 6 to additionally save multiplier resources.
- it is *fully automated*, from the parsing of an expression describing the function to VHDL generation. It is integrated in the FloPoCo class hierarchy under the name FunctionEvaluator and allows for the generated code to be optimized for a wide range of target FPGAs and operating frequencies.
- it is fully pipelined to a user-specified frequency.
- the resulting architecture *evaluates the function with last-bit accuracy*. The associated `emulate()` function and the integration in the FloPoCo framework allows generating test-benches for operator testing.

7.2 Function evaluation by polynomial approximation

Polynomial approximation is the generic mathematical tool that reduces the evaluation of a function to additions and multiplications. For these operations, we can either build architectures (in FPGAs or ASICs), or use built-in operators (in processors or DSP-enabled FPGAs). A good primer on polynomial approximation for function evaluation is Muller's book [123].

Building a polynomial evaluator for a function may be decomposed into two subproblems: 1/ *approximation*: finding a good approximation polynomial, and 2/ *evaluation*: evaluating it using adders and multipliers. The smaller the input argument, the better these two steps will behave, therefore a *range reduction* may be applied first if the input interval is large.

1. http://sollya.gforge.inria.fr/

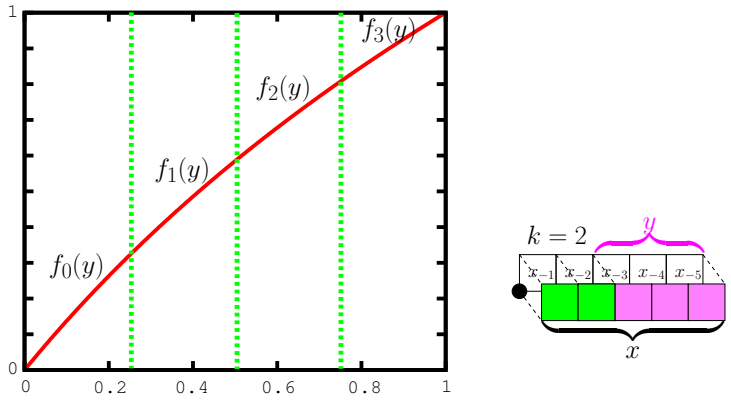


Figure 7.3 *Range reduction example for the $f(x) = \log_2(x)$, for $x \in [0, 1)$ where the input interval is split into four sub-intervals*

We now discuss each of these steps in more detail, to build the implementation flow depicted on Figure 7.2. In this chapter, without loss of generality we consider a function f over the input interval $x \in [0, 1)$.

In our implementation, the user inputs a function (assumed on $[0, 1)$), the input and output precisions (both expressed as LSB weight), and the degree d of the polynomials used. This last parameter could be determined heuristically, but we leave it as a user-defined parameter which allows to trade-off multipliers and latency for memory size.

7.2.1 Range reduction

In this work, we use the simple range reduction that consists in splitting the input interval in 2^k sub-intervals, indexed by $i \in \{0, 1, ..., 2^k - 1\}$. The index i may be obtained as the leading bits of the binary representation of the input: $x = 2^{-k}i + y$ with $y \in [0, 2^{-k})$. This decomposition comes at no hardware cost. We now have $\forall i \in \{0, \ldots, 2^k - 1\} \quad f(x) = f_i(y)$, and we may approximate each f_i by a polynomial p_i. This simple range reduction is illustrated in Figure 7.3. A table will hold the coefficients of all these polynomials, and the evaluation of each polynomial will share the same hardware (adders and multipliers), which therefore has to be built to accommodate the worst-case among these polynomials. Figure 7.5 describes the resulting architecture.

Compared to using a single polynomial on the interval, this range reduction increases the storage space required, but decreases the cost of the evaluation hardware for two reasons. First, for a given target accuracy $\varepsilon_{\text{total}}$, the degree of each of the p_i decreases with increasing k. There is a strong threshold effect here, and for a given degree there is a minimal k that allows to achieve the accuracy.

Second, the reduced argument y has k bits less than the input argument x, which will reduce the input size of the corresponding multipliers. If we target an FPGA with DSP blocks, there will also be a threshold effect here on the number of DSP blocks used.

Many other range reductions are possible, most related to a given function or class of functions, like the logarithmic segmentation used in [52]. For an overview, see Muller [123]. Most of our contributions are independent of the range reduction used.

7.2.2 Polynomial approximation

One may use the well-known Taylor or Chebyshev approximation polynomials of arbitrary degree d [123]. These polynomials can be obtained analytically, or using computer algebra systems.

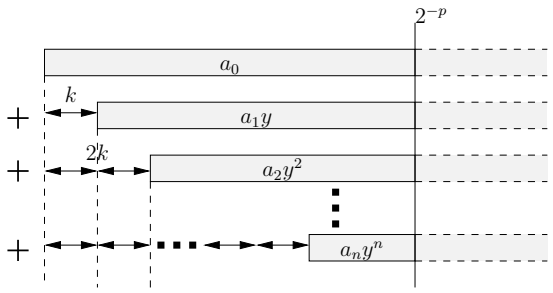


Figure 7.4 *Alignment of the monomials*

A third method of polynomial approximation is Remez' algorithm, a numerical process that, under some conditions, converges to the minimax approximation: the polynomial of degree d that minimizes the maximal difference between the polynomial and the function. We denote $\varepsilon_{\text{approx}}$ the approximation error, defined as the maximum absolute difference between the polynomial and the function.

Between approximation and evaluation, for an efficient machine implementation, one has to round the coefficients of the minimax polynomial (which are real numbers in theory, and are computed with large precision in practice) to smaller-precision numbers suitable for efficient evaluation. On a processor, one will typically try to round to single- or double-precision numbers. On an FPGA, we may build adders and multipliers of arbitrary size, so we have one more question to answer: what is the optimal size of these coefficients?

Lee et al. [105] use an error analysis that considers separately the error of rounding each coefficient of the minimax polynomial (considered as a real-coefficient one) and tries to minimize the bit-width of the rounded coefficients while remaining within acceptable error bounds. However, there is no guarantee that the polynomial obtained by rounding the coefficients of the real minimax polynomial is the minimax among the polynomials with coefficients constrained to these bit-width. Indeed, this assumption is generally wrong.

Finer quality polynomials are obtained using the analysis by Tisserand [149] for low-degree polynomials targeting very low precisions. There, after rounding the minimax coefficients to the target precisions, several other polynomials derived from the initial one are explored. The new polynomials are obtained by allowing each coefficient to "move" in a tight interval around this initial value. The polynomial with the smallest approximation error is then returned. While testing all these polynomials is possible for low polynomial degrees, this method doesnt's scale to larger degree polynomials. Moreover the method finds the local-minimum in the neighborhood of the rounded minimax solution but this may not be the global optimum.

One may obtain much more accurate polynomials for the same coefficient bit-width using a modified Remez algorithm due to Brisebarre and Chevillard [48] and implemented as the `fpminimax` command of the Sollya tool. This command inputs a function, an interval and a list of constraints on the coefficients (e.g. constraints on bitwidths), and returns a polynomial that is very close to the best minimax approximation polynomial among those with such constrained coefficients.

Since the approximation polynomial now has constrained coefficients, we will not round these coefficients anymore. In other words, we have merged the approximation error and the coefficient truncation error of Lee et al. [105] into a single error, which we still denote $\varepsilon_{\text{approx}}$. The only remaining rounding or truncation errors to consider are those that happen during the evaluation of the polynomial.

Let us now provide a good heuristic for determining the coefficient constraints. Let $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$ be the polynomial on one of the sub-intervals (for clarity, we remove the

indices corresponding to the sub-interval). The constraints taken by `fpminimax` are the minimal weights of the least significant bit (LSB) of each coefficient. To reach some target accuracy 2^{-p}, we need the LSB of a_0 to be of weight at most 2^{-p}. As $P(y)$ is the sum of $d+1$ terms, a few guard bits are additionally needed for a_0 such that the summation accuracy will be of the order 2^{-p}. This provides the constraint on a_0.

Now consider the developed form of the polynomial, with the terms illustrated in Figure 7.4. Coefficient a_j is multiplied by y^j ($y < 2^{-k}$) which is smaller than 2^{-kj}. The accuracy of the monomial $a_j y^j$ will be aligned on that of the monomial a_0 if its LSB is of weight $2^{-p+kj+g}$. This provides a constraint on a_j.

The heuristic used is therefore the following (remember that the degree d is provided by the user):

- the constraints on the $d+1$ coefficients are set as previously explained. Moreover, for a given k we also explore neighboring coefficient constraints. From this new set of constraints, some may reduce block memory cost by sufficiently reducing the coefficient table width so that it falls into a memory sweet-spot (a memory whose width is a multiple of the embedded memory width, for instance the memory 512x72=(512x32)x2 falls in such a sweet-spot on a Virtex-4).
- for increasing k, we try to find 2^k approximation polynomials p_i of degree d respecting the constraints, and fulfilling the target approximation error (which will be defined in Section 7.2.4). The smallest k might not be the best from the implementation point of view: larger k can fill-up better the used block memories and reduce evaluation cost. For instance $k = 7$ on a Virtex4 fills 128 memory addresses out of the minimum 256 of the half-BRAM, thus wasting half the resources. If $k = 8$, the size of y is reduced by one bit and we may also gain a few bits on the coefficients, as this time the same degree polynomials are used on half the interval size.
- the best value of k which meets all the requirements is then returned. The maximum magnitude of all the coefficients of degree j (the largest MSB) together with the constraints on their LSB give the width that each coefficient occupies in the final coefficient table. These values are constantly used in order to target memory sweet-spots. All this information must then be passed towards the polynomial evaluator.

7.2.3 Polynomial evaluation

Given a polynomial, there are many possible ways to evaluate it. The HOTBM method [69] uses the developed form $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$ and attempts to tabulate as much of the computation as possible. This leads to short-latency architecture since each of the $a_i y^i$ may be evaluated in parallel and added thanks to an adder tree, but at a high hardware cost.

In this work, we chose a more classical Horner evaluation scheme, which minimizes the number of operations, at the expense of latency: $p(y) = a_0 + y \times (a_1 + y \times (a_2 + ... + y \times a_d)...)$. Our contribution is essentially a fine error analysis that allows us to minimize the size of each of the operations. This analysis is presented in Section 7.2.4.

There are intermediate schemes that could be explored. For large degrees, the polynomial may be decomposed into an odd and an even part: $p(y) = p_e(y^2) + y \times p_o(y^2)$. The two sub-polynomial may be evaluated in parallel, so this scheme has a shorter latency than Horner, at the expense of the precomputation of x^2 and a slightly degraded accuracy. Many variations on this idea, e.g. the Estrin scheme, exist [123], and this should be the subject of future work. A polynomial may also be refactored to trade multiplications for more additions [96], but this idea is mostly incompatible with range reduction.

7.2.4 Accuracy and error analysis

The maximal error target $\varepsilon_{\text{total}}$ is an input to the algorithm. Typically, we aim at *faithful rounding*, which means that $\varepsilon_{\text{total}}$ must be smaller than the weight of the LSB of the result, noted u. In other words, all the bits returned hold useful information. This error is decomposed as follows: $\varepsilon_{\text{total}} = \varepsilon_{\text{approx}} + \varepsilon_{\text{eval}} + \varepsilon_{\text{finalround}}$ where

- $\varepsilon_{\text{approx}}$ is the approximation error, the maximum absolute difference between any of the p_i and the corresponding f_i over their respective intervals. This computation belongs to the approximation step and is also performed using Sollya [53].
- $\varepsilon_{\text{eval}}$ is the total of all rounding and truncation errors committed during the evaluation;
- $\varepsilon_{\text{finalround}}$ is the error corresponding to the final rounding of the evaluated polynomial to the target format. It is bounded by $u/2$.

We therefore need to ensure $\varepsilon_{\text{approx}} + \varepsilon_{\text{eval}} < u/2$. The polynomial approximation algorithm iterates until $\varepsilon_{\text{approx}} < u/4$, then reports $\varepsilon_{\text{approx}}$. The error budget that remains for the evaluation is therefore $\varepsilon_{\text{eval}} < u/2 - \varepsilon_{\text{approx}}$ and is between $u/4$ and $u/2$.

In $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$, the input y is considered exact, so $p(y)$ is the value of the polynomial if evaluated in infinite precision. What the architecture evaluates is $p^*(y)$, and our purpose here is to compute a bound on $\varepsilon_{\text{eval}}(y) = p^*(y) - p(y)$.

Let us decompose the Horner evaluation of p as a recurrence:

$$\begin{cases} \sigma_0 = a_d \\ \pi_j = y \times \sigma_{j-1} & \forall j \in \{1...d\} \\ \sigma_j = a_{d-j} + \pi_j & \forall j \in \{1...d\} \\ p(y) = \sigma_d \end{cases}$$

This would compute the exact value of the polynomial, but at each evaluation step, we may perform two truncations, one on y, and one on π_j. As a rule of thumb, each step should balance the effect of these two truncations on the final error. For instance, in an addition, if one of the addends is much more accurate than the other one, it probably means that it was computed too accurately, wasting resources.

To understand what is going on, consider step j. In the addition $\sigma_j = a_{d-j} + \pi_j$, the π_j should be at least as accurate as a_{d-j}, but not much more accurate: let us keep g_j^π bits to the right of the LSB of a_{d-j}, where g_j^π is a small positive integer ($0 \leq g_j^\pi < 5$ in our experiments). The parameter g_j^π defines the truncation of π_j, and also the size of σ_j (which also depends on the weight of the MSB of a_{d-j}).

Now, since we are going to truncate $\pi_j = y \times \sigma_{j-1}$, there is no need to input to this computation a fully accurate y. Instead, y should be truncated to the size of the truncated π_j, plus a small number g_j^y of guard bits.

The computation actually performed is therefore the following:

$$\begin{cases} \sigma_0^* = a_d \\ \pi_j^* = \tilde{y}_j \times \sigma_{j-1}^* & \forall j \in \{1...d\} \\ \sigma_j^* = a_{d-j} + \tilde{\pi}_j^* & \forall j \in \{1...d\} \\ p^*(y) = \sigma_d^* \end{cases}$$

In both previous equations, the additions and multiplications should be viewed as exact: the truncations are explicited by the tilded variables, e.g. $\tilde{\pi}_j^*$ is the truncation of π_j^* to g_j^π bits beyond the LSB of a_{d-j}. There is no need to truncate the result of the addition, as the truncation of π_j^* serves this purpose already.

We may now compute the rounding error:

$$\varepsilon_{\text{eval}} = p^*(y) - p(y) = \sigma_d^* - \sigma_d$$

where

$$\begin{aligned}\sigma_j^* - \sigma_j &= \tilde{\pi}_j^* - \pi_j \\ &= (\tilde{\pi}_j^* - \pi_j^*) + (\pi_j^* - \pi_j)\end{aligned}$$

Here we have a sum of two errors. The first, $\tilde{\pi}_j^* - \pi_j^*$, is the truncation error on π^* and is bounded by a power of two depending on the parameter g_j^π. The second is computed as

$$\begin{aligned}\pi_j^* - \pi_j &= \tilde{y}_j \times \sigma_{j-1}^* - y \times \sigma_{j-1} \\ &= (\tilde{y}_j \sigma_{j-1}^* - y\sigma_{j-1}^*) + (y\sigma_{j-1}^* - y\sigma_{j-1}) \\ &= (\tilde{y}_j - y)\sigma_{j-1}^* \quad + \quad y \times (\sigma_{j-1}^* - \sigma_{j-1})\end{aligned}$$

Again, we have two error terms which we may bound separately. The first bound is the truncation error on y, which depends on the parameter g_j^y, and is multiplied by a bound on σ_{j-1}^* which has to be computed recursively itself. The second term recursively uses the computation of $\sigma_j^* - \sigma_j$, and the bound $y < 2^{-k}$.

The previous error computation is implemented in C++. From the values of the parameters g_j^π and g_j^y, it decides if the architecture defined by these parameters is accurate enough.

7.2.5 Parameter space exploration for the FPGA target

The architecture of our implementation is depicted in Figure 7.5. It consists of a table storing the coefficients of the polynomials $p_0 \dots p_{2^k-1}$ approximating $f(x)$ on $x \in [0..1)$ and one polynomial evaluator using the Horner scheme. As briefly introduced in Section 7.2.1 various optimizations are applied for generating this coefficient table. Their purpose is to minimize the block-memory count and at the same time, if possible, reduce the multiplier cost. We will next present in some more details these optimizations, and we will use the Virtex4 FPGA to illustrate our examples (the same type of optimizations are preformed for all FPGAs).

First, k directly influences block memory count. The BRAM configuration allowing for the widest output is 512x36 bits for Virtex4. Moreover, this configuration has two independent ports, therefore its granularity can be seen as 2x(256x36) bits. In the following we consider the case for which the function evaluators are needed every clock cycle and the frequency is high-enough that no time-multiplexing schemes can be applied [2]. In such a case, if the number of polynomials used for one function is less than 256, the rest of the half memory block will remain unused. The remaining half-memory block can be efficiently used to wrap in the same block memory coefficient tables wider than 36 bits. Therefore, our heuristic will try to fill these half-tables. By doing so we minimize y which reduces the overall evaluation cost. Additionally, this may also gain a few bits on the coefficients (the same-degree polynomials are used on half the interval) which reduces evaluation cost and may allow exploiting a memory sweet-spot.

Secondly, the total width of the coefficient table is computed as the maximum width needed to store each of the $d+1$ coefficients. As the same evaluation unit is used for all the 2^k polynomials, these are needed to be stored in memory in the same format. Therefore, once a set of 2^k approximation polynomials have been found, the main task is to compute the magnitude of their coefficients in order to determine their width. For $2^k = 512$, the best use of Xilinx memory blocks of Virtex4 devices is the 512x36 bits configuration. If the coefficient table would have 74-bits for instance, three blocks would be needed for storage. Our heuristic tries several tighter coefficient constraints in order to reduce this number to 72, a sweet-spot for this device. In the case when $2^k = 1024$, the configuration 1024x18 bits would be better used. In this case our heuristic will try to reduce the coefficient size to a multiple of 18, and so on.

2. If time-multiplexing can be effectively applied when the target circuit's frequency is significantly lower than the nominal speed of embedded memories such that multiple reads/writes can be accomplished during one system cycle.

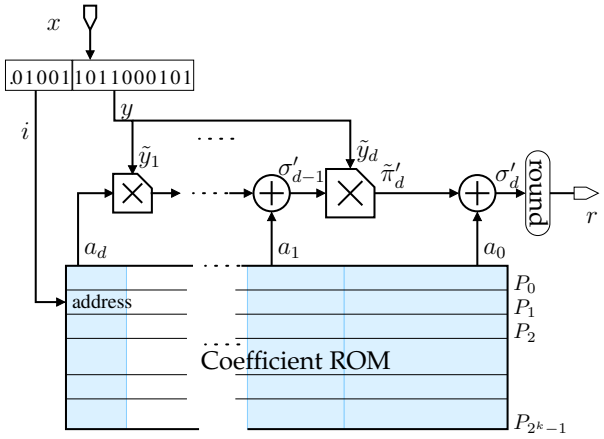


Figure 7.5 *The function evaluation architecture*

Next, moving to the polynomial evaluator component of our implementation, we need to find the truncation sizes which minimize the number of required embedded multipliers. Let us first consider the g_j^y parameter. The size of this truncation directly influences the DSP count. Here, we observe that once a DSP block is used, it saves us almost nothing to under-use it. We therefore favor truncations which reduce the size of y to the smallest multiple of a multiplier input size that allows us to reach the target accuracy. For Virtex4 and StratixII, the size of y should target a multiple of 17 and 18 respectively. On Virtex5 and Virtex6, multiples of 17 or 24 should be investigated. Consequently, each g_j^y can take a maximum of three possible values: 0, corresponding to no truncation, and one or two soft spots corresponding to multiples of multiplier input size.

The determination of the possible values of g_j^π also depends on the DSP multiplier size, as the truncation of π_j^* defines the size of the sum σ_j^*, which inputs a multiplier. There are two considerations to be made: First, it makes no sense to keep guard bits to the right of the LSB of $\tilde{\pi}_j^*$. This gives us an upper bound on g_j^π. Secondly, as we are trying to reduce DSP count, we should not allow a number of guard bits that increases the size of σ_j^* over a multiple of the multiplier input size. This gives us a second upper bound on g_j^π. The real upper-bound in computed as a minimum of the two precomputed upper-bounds.

These upper bounds define the parameter space to explore. We also observe that the size of the multiplications increases with j in our Horner evaluation scheme. We therefore favor truncations in the last Horner steps, as these truncations can save more DSP blocks. This defines the order of exploration of the parameter space. The parameters g_j^π and g_j^y are explored using the above rules until the error $\varepsilon_{\text{eval}}$ satisfies the bound $\varepsilon_{\text{eval}} < u/2 - \varepsilon_{\text{approx}}$.

Finally, we use truncated multipliers in order to additionally reduce multiplier usage. Consider g_j^π which determines the number of guard bits for a_{d-j}, before its addition to π_j^*. At the same time, g_j^π determines the truncation size of π_j^*. If part of this result is not used, then we may use truncated multipliers to compute this value, accurate to this truncation size. This does not change the presented error analysis, as this truncation is already accounted for. However, truncated multipliers have an important impact on the number of multipliers used.

All in all, this is a fairly small parameter space exploration, and its execution time is negligible with respect to the few seconds it may take to compute all the constrained minimax approximations.

Standard datapath			Truncated datapath			Trunc. datapath + mult		
π_1	43×15	3 DSPs	π_1^*	$\mathbf{18} \times 15$	1 DSPs	π_1^*	35×19 trunc. to 24	2 DSPs
π_2	43×26	6 DSPs	π_2^*	$\mathbf{35} \times 26$	4 DSPs	π_2^*	35×30 trunc. to 32	3 DSPs
π_3	43×37	9 DSPs	π_3^*	43×39	9 DSPs	π_3^*	44×41 trunc. to 40	4 DSPs
π_4	43×48	9 DSPs	π_4^*	43×52	9 DSPs	π_4^*	44×52 trunc. to 50	5 DSPs
		27 DSPs			23 DSPs			14 DSPs

Table 7.1 *The decrease in internal datapath truncations allows reducing DSP count*

7.3 Reality check

This section will first try to show the effect of the several optimizations we apply in the process of generating the implementation. Then, we will provide general data for the implementation of several common fixed-point functions for several accuracies, together with synthesis results for these functions on a Virtex4 FPGA. Finally, we compare our generic approach against the CORDIC implementation of $\sin$ and $\cos$ of Logicore.

7.3.1 Optimization effect

Let us take as a running example the function $\log_2(1 + x)$. For an implementation accurate to 23-bits (single-precision equivalent) this function requires $2^k = 128$ subintervals. The coefficients sizes are 26, 20, 12 plus the 3 more sign bits as we store our coefficients signed in memory: these total 61 bits. A direct implementation on our Virtex4 FPGA would require 2 BRAMs with configuration 512x36. Due to the dual-port nature of the BRAM, the 25-bits exceeding the 36-bit capacity are packed in the BRAM starting with address 256. This optimization saves one memory blocks, reduces the latency by one cycle (from 9 to 8 in our case) and reduces slice count (from 134 to 121 for our case). For the 36-bit version of the same operator this optimization reduces the BRAM count from 4 to 2.

We next investigate the savings on the total DSP count by truncating the operators to the minimum width. For this we use $\log_2(1 + x)$ for 52-bit accuracy (double-precision equivalent) for which the internal multiplication sizes are shown in the left column of Table 7.1. This standard implementation would require 27 DSP blocks. The middle column shows the effects of the datapath trimming on the multiplier inputs: a total reduction of four DSP blocks is accomplished by finding the datapath's sweet-spot (one multiplier input is reduced from 43 to 18, another from 43 to 35. The third column shows the reduction in DSP count caused by the introduction of truncated multipliers in the datapath. This entails additional savings, roughly reducing the initial count by half.

All in all, the presented optimizations significantly reduce both BRAM and DSP count. Let us now give more general results and see how well these perform against some examples from the literature.

7.3.2 Examples and comparisons

Table 7.2 presents the input and output parameters for obtaining the approximation polynomials for several representative functions mentioned in the introduction. The functions f are all considered over $[0, 1]$, with identical input and output precision. Three precisions are given in Table 7.2. Table 7.3 provides synthesis results for the same experiments.

It is difficult to compare to previous works, especially as none of them scales to the large precisions we do. Our approach brings no savings in terms of DSP blocks for precisions below 17 bits.

Table 7.2 *Examples of polynomial approximations obtained for several functions. S represents the scaling factor so that the function image is in [0,1]*

$f(x)$	S	23 bits (single prec.)			36 bits			52 bits (double prec.)		
		d	k	Coeffs size	d	k	Coeffs size	d	k	Coeffs size
$\sqrt{1+x}$	$\frac{1}{2}$	2	64	$26, 20, 14$	3	128	$39, 32, 25, 18$	4	512	$55, 46, 37, 28, 19$
		1	2048	$26, 15$	2	2048	$39, 28, 17$	3	2048	$55, 44, 33, 22$
$\frac{\pi}{4} - \frac{\sin(\frac{\pi}{4}x)}{x}$	2^3	2	128	$26, 19, 12$	3	128	$39, 32, 25, 18$	4	256	$55, 47, 39, 31, 23$
		1	4096	$26, 14$	2	2048	$39, 28, 17$	3	2048	$55, 44, 33, 22$
$1 - \cos(\frac{\pi}{4}x)$	2	2	128	$26, 19, 12$	3	256	$39, 31, 23, 15$	4	256	$55, 47, 39, 31, 23$
		1	4096	$26, 14$	2	2048	$39, 28, 17$	3	4096	$55, 43, 31, 19$
$\log_2(1+x)$	1	2	128	$26, 19, 12$	3	256	$39, 31, 23, 15$	4	256	$55, 45, 35, 25, 15$
		1	4096	$26, 14$	2	4096	$39, 27, 15$	3	4096	$55, 43, 31, 19$
$\frac{\log(x+1/2)}{x-1/2}$	$\frac{1}{2}$	2	256	$26, 18, 10$	3	512	$39, 30, 21, 12$	4	1024	$55, 45, 35, 25, 15$
		1	4096	$26, 14$	2	4096	$39, 27, 15$	3	8192	$55, 42, 29, 16$

Table 7.3 *Synthesis Results using ISE 11.1 on VirtexIV xc4vfx100-12. l is the latency of the operator in cycles. All the operators operate at a frequency close to 320 MHz. The grayed rows represent results without coefficient table BRAM compaction and the use of truncated multipliers*

$f(x)$	23 bits (single prec.)					36 bits					52 bits (double prec.)				
	d	l	slices	DSP	BRAM	$d.$	l	slices	DSP	BRAM	d	l	slices	DSP	BRAM
$\sqrt{1+x}$	2	9	118	3	2	3	18	351	9	3	4	32	893	21	5
	1	5	62	1	5	2	12	231	5	9	3	24	668	15	17
$\sqrt{1+x}$	2	8	92	2	1	3	17	672	3	2	4	31	1313	11	6
	1	4	37	1	3	2	11	373	3	5	3	23	819	9	18
$\frac{\pi}{4} - \frac{\sin(\frac{\pi}{4}x)}{x}$	2	9	120	2	1	3	19	1039	4	2	4	34	1172	14	3
	1	4	36	1	11	2	13	412	3	11	3	25	1029	10	19
$1 - \cos(\frac{\pi}{4}x)$	2	9	120	2	1	3	19	1039	4	2	4	34	1773	14	3
	1	4	36	1	11	2	13	412	3	11	3	22	790	9	40
$\log_2(1+x)$	2	9	120	2	1	3	21	1066	4	2	4	33	1569	14	6
	1	4	36	1	11	2	11	320	3	22	3	24	933	9	40
$\frac{\log(x+1/2)}{x-1/2}$	2	8	103	2	1	3	17	779	4	4	4	32	1584	12	11
	1	4	36	1	11	2	11	314	3	22	3	23	999	8	78

LogiCore CORDIC 4.0 sin+cos 32 cyles@296MHz, 3812 LUT, 3812 FF
This work, sin alone 16 cycles@353MHz, 2 BlockRam, 3 DSP48E, 575 FF, 770 LUT
This work, cos alone 16 cycles@390MHz, 2 BlockRam, 3 DSP48E, 609 FF, 832 LUT

Table 7.4 *Comparison with CORDIC for 32-bit sine/cosine functions on Virtex5*

We may compare to the logarithm unit by Lee et al. [106] which computes $\log(1+x)$ on 27 bits using a degree-2 approximation. Our tool instantly finds the similar coefficient sizes 30, 22 and 12 (13 in [106]). However, our implementation uses 2 DSP blocks where [106] uses 6: one multiplier is saved thanks to the truncation of y and others thanks to truncated multipliers. For larger precisions, the savings would also be larger.

We should compare the polynomial approach to the CORDIC family of algorithms which can be used for many elementary functions [123, 26]. Table 7.4 compares implementations for 32-bit sine and cosine, using for CORDIC the implementation from Xilinx LogiCore [26]. This table illustrates that these two approaches address different ends of the implementation spectrum. The polynomial approach provides smaller latency, higher frequency and low logic consumption (hence predictability in performance independently of routing pressure). The CORDIC approach consumes no DSP nor memory block. Variations on CORDIC using higher radices could improve frequency and reduce latency, but at the expense of an even higher logic cost. A deeper comparison remains to be done.

7.4 Conclusion, open issues and future work

Application-specific systems sometimes need application-specific operators, and this includes operators for function evaluation. This work has presented a fully automatic design tool that allows one to quickly obtain architectures for the evaluation of a polynomial approximation with a uniform range reduction for large precisions, up to 64 bits. The resulting architectures are better optimized than what the literature offers, firstly thanks to state-of-the-art polynomial approximation tools, secondly thanks to a finer error analysis that allows for truncating the evaluation datapath and thirdly thanks to the state-of-the-art truncated multipliers available in the FloPoCo framework which were integrated in the polynomial evaluator. The architectures presented here benefit from the FloPoCo framework integration and may therefore be fully pipelined to a frequency close to the nominal frequency of current FPGAs.

This work will enable the design, in the near future, of elementary function libraries for reconfigurable computing that scale to double precision. However, we also wish to offer the designer a tool that goes beyond a library: a generator that produces carefully optimized hardware for his very function. Such application-specific hardware may be more efficient than the composition of library components.

Towards this goal, this work can be extended in several directions.

- Non-uniform range reduction schemes should be explored. The power-of-two segmentation of the input interval used in [52] has a fairly simple hardware implementation using a leading zero or one counter. This will enable more efficient implementation of some functions.
- More parallel versions of the Horner scheme should be explored to reduce the latency.
- Our tools could attempt to detect if the function is odd or even [103], and consider only odd or even polynomials for such case [123, 103]. Whether this works along with range reduction remains to be explored.

– We currently only consider a constant target error corresponding to faithful rounding, but a target error function could also be input.
– Designing a pleasant and universal interface for such a tool is a surprisingly difficult task. Currently, we require the user to input a function on $[0, 1)$, and the input and output LSB weight. Most functions can be trivially scaled to fit in this framework, but many other specific situations exist.

Thanks

The material presented in this chapter is based on collaborations with Mioara Joldes, a PhD student at ENS de Lyon and one of the main developers of the Sollya tool. I would like to kindly thank Mioara and her supervisors Nicolas Brisebare and Jean-Michel Muller for making this collaboration possible.

CHAPTER 8
Multiplicative square root algorithms

Most current square root implementations for FPGAs use a digit recurrence algorithm which is well suited to their LUT structure. However, recent computing-oriented FPGAs include embedded multipliers and memory blocks which can also be used to implement quadratic convergence algorithms, very high radix digit recurrences, or polynomial approximation algorithms. In this chapter we compare the classical digit-recurrence implementation of FloPoCo [1] to a new polynomial approximation implementation based on the *FunctionEvaluator* operator described in the previous chapter. We prove that polynomial approximation implementations manage to achieve shorter latencies than the classical approach for faithful (last bit accurate) results. Moreover, we show that the cost of IEEE-compliant correct rounding using such approximation algorithms is very high, and faithful operators are advocated in this case.

8.1 Algorithms for floating-point square root

There are two main families of algorithms that can be used to extract square roots.

The first family is that of *digit recurrences*, which provides one digit (often one bit) of the result at each iteration. Each iteration consists of additions and digit-by-number multiplications (which have comparable cost) [81]. Such algorithms have been widely used in microprocessors that didn't include hardware multipliers. Most FPGA implementations in vendor tools or in the literature [113, 104, 73] use this approach, which was the obvious choice for early FPGAs, which did not yet include embedded multipliers. Probably this is also the approach which minimizes the complexity in terms of logical operations for computing the square root.

The second family of algorithms is *multiplication based*, and was studied as soon as processors included hardware multipliers. It includes quadratic convergence recurrences derived from the Newton-Raphson iteration, used in AMD IA32 processors starting with the K5 [137], in more recent instruction sets such as Power/PowerPC and IA64 (Itanium) whose floating-point unit is built around the fused multiply-and-add [119, 56], and in the INV_SQRT core from the Altera MegaWizard. Other variations involve piecewise polynomial approximations [93, 130]. On FPGAs, the VFLOAT project [153] uses an argument reduction based on tables and multipliers, followed by a polynomial evaluation of the reduced argument.

To sum up, digit recurrence approaches allow one to build minimal hardware, while multiplicative approaches allow one to make the best use of available resources when these include multipliers. As a bridge between both approaches, a very high radix algorithm introduced for the

1. This implementation is based on the FPLibrary FPSqrt operator

Cyrix processors [47] is a digit-recurrence approach where the digit is 17-bit wide, and digit-by-number multiplication uses the 17x69-bit multiplier designed for floating-point multiplication.

Now that high-end FPGAs embed several thousands small multipliers, the purpose of this work is to study how this resource may be best used for computing square root [102]. To this purpose we provide an implementation of a multiplier-based square root based on polynomial evaluation, which is, to our knowledge, original in the context of FPGAs.

The conclusion is that it is surprisingly difficult to really benefit from the embedded multipliers as precision increases from single to double-precision. One problem is correct rounding (mandated by the IEEE-754 standard) which is shown to require a large final squaring of size $2 + w_F$ bits.

Even if correct rounding is relaxed to save this final operation (which is perfectly acceptable if the square root is used to build coarser atomic operators such as $\sqrt{x^2 + y^2}$), the logic consumption of a double-precision multiplicative square root surpasses that of a digit-recurrence one, of comparable performance and which doesn't consume any multiplier. For this case, the saving of using the polynomial approximation version are in terms of latency: the operator's pipeline depth is usually reduced to half, but also in terms of performance predictability, due to the lower routing pressure.

8.1.1 Notations and terminology

In all this chapter, x, the input, is a floating-point number on w_F bits of mantissa and w_E bits of exponent. IEEE-754 single precision is $(w_E, w_F) = (8, 23)$ and double-precision is $(w_E, w_F) = (11, 52)$.

Given a floating-point format with w_F bits of mantissa, it makes no sense to build an operator which is accurate to less than w_F bits: it would mean wasting storage bits, especially on an FPGA where it is possible to use a smaller w_F instead. However, the literature distinguishes two levels of accuracy, as previously presented in Chapter 3: correct and faithful rounding. Consider the round-to-nearest rounding more:

- with correct rounding (CR): the operator returns the FP number nearest to $\sqrt{x}$. This correspond to a maximum error of $0.5\texttt{ulp}$ with respect to the exact mathematical result. Noting the (normalized) mantissa $1.F$ with F a w_F-bit number, the ulp value is 2^{-w_F}. Correct rounding is the best that the format allows.
- with faithful rounding (FR): the operator returns one of the two FP numbers closest to $\sqrt{x}$, but not necessarily the nearest. This corresponds to a maximum error strictly smaller than $1\texttt{ulp}$.

In general, to obtain a faithful evaluation of a function such as $\sqrt{x}$ to w_F bits, one needs to first approximate it to a precision higher than that of the result (we denote this intermediate precision w_F+g where g is a number of guard bits), then round this approximation to the target format. This final rounding performs an incompressible error of almost $0.5\texttt{ulp}$ in the worst case, therefore it is difficult to directly obtain a correctly rounded result: one needs a very large g, typically $g \approx w_F$ [123]. It is much less expensive to obtain a faithful result: a small g (typically less than 5 bits) is enough to obtain an approximation on $w_F + g$ bits with a total error smaller than $0.5\texttt{ulp}$, to which we then add the final rounding error of another $0.5\texttt{ulp}$.

However, in the specific case of the square root, the overhead of obtaining correct rounding is lower than in the general case. Section 8.1.2 shows a general technique to convert a faithful square root on $w_F + 1$ bits to a correctly rounded one on w_F bits. This technique is, to our knowledge, due to [93], and its use in the context of a hardware operator is novel.

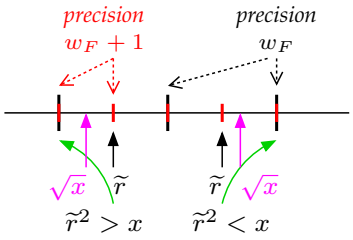


Figure 8.1 *Deducing the correctly rounded value of $\sqrt{x}$ on w_F bits from a faithfully rounded result on w_{F+1} bits*

8.1.2 The cost of correct rounding

For square root, correct rounding may be deduced from faithful rounding thanks the following technique, used in [93]. We first compute a value of the square root $\widetilde{r}$ on $w_F + 1$ bits, faithfully rounded to that format (total error smaller than 2^{-w_F-1}). This is relatively cheap. Now, with respect to the w_F-bit target format, $\widetilde{r}$ is either a floating-point number, or the exact middle between two consecutive floating-point numbers. In the first case, the total error bound of 2^{-w_F-1} on $\widetilde{r}$ entails that it is the correctly rounded square root. In the second case, squaring $\widetilde{r}$ and comparing it to x tells us (thanks to the monotonicity of the square root) if $\widetilde{r} < \sqrt{x}$ or $\widetilde{r} > \sqrt{x}$. What needs to be proved is that $\sqrt{x}$ cannot be exactly equal to the middle of the floating-point numbers at precision w_F, that is, it cannot have the form:

$$\sqrt{x} = \underbrace{1.XXXXXXX}_{w_F+1}1.$$

If $\sqrt{x}$ would indeed have this form, its square would have a length of at least $2(w_F + 2) - 1$ bits, which is impossible as x has at most $w_F + 2$ bits ($x \in [1, 4[$). The possible cases for correct rounding are illustrated in Figure 8.1.

This is enough to conclude which of its two neighboring floating-point numbers is the correctly rounded square root on w_F bits.

We use in this work the following algorithm, which is a simple rewriting of the previous idea.

$$\circ(\sqrt{x}) = \begin{cases} \widetilde{r} \text{ truncated to } w_F \text{ bits} & \text{if } \widetilde{r}^2 \geq x, \\ \widetilde{r} + 2^{-w_F-1} \text{ truncated to } w_F \text{ bits} & \text{otherwise.} \end{cases} \tag{8.1}$$

With respect to performance and cost, one may observe that the overhead of correct rounding over faithful rounding on w_F bits is

- a faithful evaluation on $w_F + 1$ bits – this is only marginally more expensive than on w_F bits;
- a square on $w_F + 1$ bits – even with state-of-the-art dedicated squarers presented in Chapter 6, this is expensive. The cost of this operation can be reduced if we consider that we are actually interested only in the lower squarer bits (see Figure 8.2). The largest difference between x and $\widetilde{r}^2$ is bounded by one `ulp`, or 2^{-w_F}. Therefore, by keeping the bits up to weight 2^{-w_F+1} out of $\widetilde{r}^2$, which we denote by $\overline{r}^2$, suffices find whether $\widetilde{r}^2 \geq x$. Indeed, if $|\overline{r}^2 - x| \leq 2^{-w_F}$ then:

$$\circ(\sqrt{x}) = \begin{cases} \overline{r} \text{ truncated to } w_F \text{ bits} & \text{if } \overline{r}^2 \geq x, \\ \overline{r} + 2^{-w_F-1} \text{ truncated to } w_F \text{ bits} & \text{otherwise.} \end{cases} \tag{8.2}$$

  Otherwise, if $|\overline{r}^2 - x| > 2^{-w_F}$:

$$\circ(\sqrt{x}) = \begin{cases} \overline{r} \text{ truncated to } w_F \text{ bits} & \text{if } \overline{r}^2 < x, \\ \overline{r} + 2^{-w_F-1} \text{ truncated to } w_F \text{ bits} & \text{otherwise.} \end{cases} \tag{8.3}$$

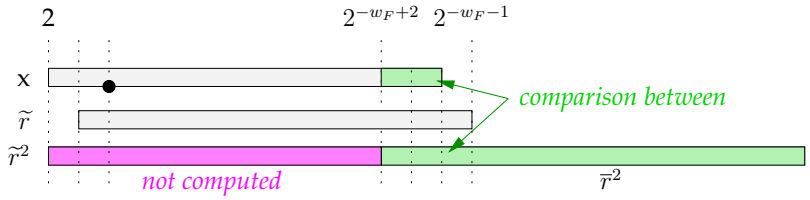


Figure 8.2 *Bits involved in the comparison of $\widetilde{x}^2 \geq x$ are highlighted*

The highlighted bits from Figure 8.2 suggest that a the squaring computation can indeed save multipliers. The savings will be larger as precision grow, and an example of a squaring architecture for double precision is depicted in Figure 8.3.

This overhead (both in area and in latency) may be considered a lot for an accuracy improvement of one half-ulp. Indeed, on an FPGA, it will make sense in most applications to favor faithful rounding on $w_F + 1$ bits over correct rounding on w_F bits (for the same relative accuracy bound).

The FloPoCo implementation offers both alternatives, but in the following, we only consider faithful implementations for approximation algorithms.

8.2 Square root by polynomial approximation

We compute the square root of a floating-point number X in a format similar to IEEE-754:

$$X = 2^E \times 1.F$$

where E is an integer (coded on w_E bits with a bias of $2^{w_E-1} - 1$), and F is the fraction part of the mantissa, written in binary on w_F bits: $1.F = 1.f_{-1}f_{-2}\cdots f_{-w_F}$ (the indices denote the bit weights).

There are classically two cases to consider.

– If E is even, the square root is

$$\sqrt{X} = 2^{E/2} \times \sqrt{1.F}\,.$$

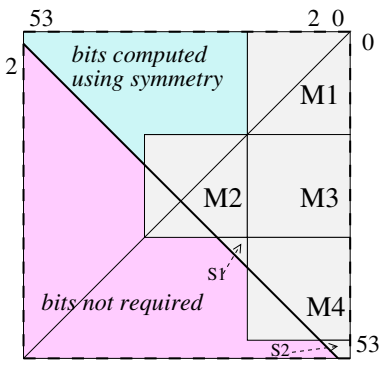


Figure 8.3 *The multipliers required for the squaring operation operation $\widetilde{r}^2$ for double-precision on Virtex4*

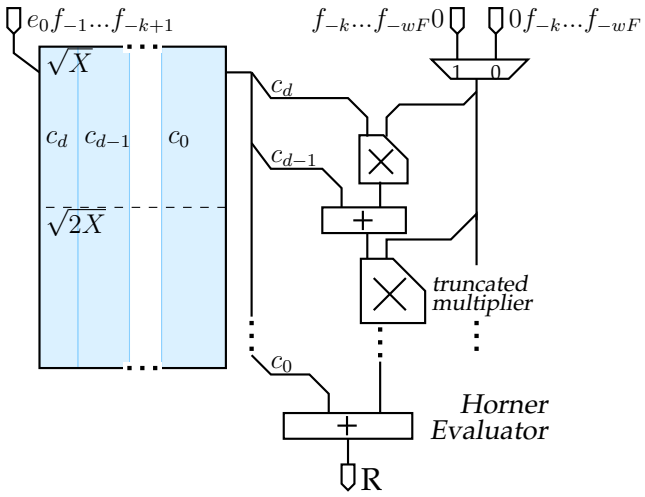


Figure 8.4 *Generic polynomial evaluator for the square root*

– If e is odd, the square root is

$$\sqrt{X} = 2^{(E-1)/2} \times \sqrt{2 \times 1.F} \; .$$

In both cases the computation of the result exponent is straightforward, and we will not detail it further. The computation of the square root is reduced to computing $\sqrt{Z}$ for $Z \in [1, 4[$.

We are classically [105] splitting the interval $[1, 4[$ into sub-intervals, and using for each sub-interval an approximation polynomial whose coefficients are read from a table. The state of the art for obtaining such polynomials is the `fpminimax` command of the Sollya tool [54] whose advantages have been stated in Chapter 7.

The polynomial evaluation hardware is shared by all the polynomials, therefore they must be of same degree d and have coefficients of the same format (here a fixed-point format). We evaluate the polynomial in Horner form, computing just right at each step by truncating all intermediate results to the bare minimum and making use of truncated multipliers.

A first idea is to address the coefficient table is to use the most significant bits of Z. However, as $Z \in [1, 4[$, the address range `00xxx...xxx` is unused, which would mean that one quarter of the table is never addressed. A good compiler might save the resources needed to implement this quarter-table for some very specific cases. For example, if the table size is 2048×36 bits (4 BRAMs on Virtex-4 using the configuration 512×36) one BRAM, used to implement one quarter of the table, could be saved.

A better idea is to take advantage that the function $\sqrt{Z}$ varies more for Z on the left of interval $[1, 4[$. For a given degree d, the polynomials on the left of $[1, 4[$ will be less accurate than those on the right. Our improved strategy consists in splitting the computation $\sqrt{Z}$ over $[1, 4[$ into two cases, according to the exponent parity: $[1, 2[$ for even exponents and $[2, 4[$ for odd exponents. We now split both $[1, 2[$ and $[2, 4[$ into an equal number of sub-intervals, with the sub-interval size for $[1, 2[$ being twice as small than that for $[2, 4[$. This technique will balance the approximation errors between the two cases and can save, in most cases, one quarter of the coefficient table.

Next we give the details of the algorithm. Let k be an integer parameter that defines the number of sub-intervals (2^k in total). The coefficient table has 2^k entries.

– If E is even, let $\tau_{\text{even}}(x) = \sqrt{1+x}$ for $x \in [0, 1)$: we need a piecewise polynomial approximation for τ_{even}.
The interval $[0, 1[$ is split into 2^{k-1} sub-intervals $[\frac{i}{2^{k-1}}, \frac{i+1}{2^{k-1}}[$ for i from 0 to $2^{k-1} - 1$. The index (and table address) i consists of the bits $f_{-1} f_{-2} \cdots f_{-k+1}$ of the mantissa $1.F$. On each

Table 8.1 *FloPoCo polynomial square root for Virtex-4 4vfx100ff1152-12 and Virtex5 xc5vlx30-3-ff324. The command line used is* `flopoco -target=Virtex4|Virtex5 -frequency=`f `FPSqrtPoly` w_E w_F `0 degree`

	(w_E, w_F)	Acc.	Degree	cycles	Synthesis results			
					Freq.	Slices	BRAM	DSP
handcrafted	$(8,23)$	FR	2	4	339	79	2	2
handcrafted		CR	2	13	303	245	2	5
		FR	2	10	318	165	2	2
		CR	2	18	334	297	2	5
	$(9,36)$	FR	3	22	319	473	4	6
		CR	3	34	270	840	4	12
	$(10,42)$	FR	3	23	311	876	8	6
Virtex4, 400 MHz		CR	3	35	311	1148	15	12
	$(11,52)$	FR	3	29	319	1192	76	8
		FR	4	35	319	1797	11	12
		FR	5	40	318	1944	7	15
2010 version:		FR	4	33	318	1145	11	26
digit-rec. FloPoCo:		CR	-	53	307	1770	-	-
digit-rec. CoreGen:		CR	-	57	265	1820	-	-
VFLOAT [153]		2.39 ulp	-	17	*>200*	*1572*	*116*	*24*
Altera $(1/\sqrt{x})$ [102]		1 ulp?	-	32	?	900 ALM	*32 M9K*	27 (18-bit)
Altera $(1/\sqrt{x})$ [31]		1 ulp?	-	36	192	1200 ALM	-	78 (18-bit)
Virtex5, 400 MHz	$(8,23)$	FR	2	7	415	177LUT 132REG	2	2
	$(9,36)$	FR	3	22	330	760 LUT, 665 REG	4	4
	$(10,42)$	FR	3	25	320	1735 LUT, 1289 REG	4	4
	$(11,52)$	FR	4	35	336	2533 LUT, 2146REG	6	9

of these sub-intervals, $\tau_{\text{even}}(1 + \frac{i}{2^{k-1}} + y)$ is approximated by a polynomial of degree d: $p_i(y) = c_{0,i} + c_{1,i}y + \cdots + c_{d,i}y^d$.

– If E is odd, we need to compute $\sqrt{2 \times 1.F}$. Let $\tau_{\text{odd}}(x) = \sqrt{2+x}$ for $x \in [0,2[$. The interval $[0,2[$ is also split into 2^{k-1} sub-intervals $[\frac{j}{2^{k-2}}, \frac{j+1}{2^{k-2}}[$ for j from 0 to $2^{k-1}-1$.
The reader may check that the index j consists of the same bits $f_{-1}f_{-2}\cdots f_{-k+1}$ as in the even case. On each of these sub-intervals, $\tau_{\text{odd}}(1 + \frac{j}{2^{k-2}} + y)$ is approximated by a polynomial q_j of same degree d.

We will use the FunctionEvaluator component presented in the previous chapter to implement these two functions. Once d is chosen by the user we need to determine the minimum number of intervals, given by k, such that $max(|\tau(y) - p_j(y)|) < 2^{-w_F-2}$ holds for both our cases $[1,2[$ and $[2,4[$. Due to the splitting strategy previously described k_{odd} should equal k_{even} in most cases. Nevertheless, we cannot guarantee this without loosing efficiency. Once the smallest number of intervals meeting the accuracy constraint has been found for the even case k_{even}, it might not be possible to meet the same accuracy constraint with $k_{\text{odd}} = k_{\text{even}}$. In such a case the table will not be filled entirely and we must rely on the compilation tool to save the memory blocks associated with the empty part of the table.

This way we obtain 2^k polynomials, whose coefficients are stored in a ROM with 2^k entries addressed by $A = e_0 f_{-1} f_{-2} \cdots f_{-k+1}$. Here e_0 is the exponent parity, and the remaining bits are i or j as above.

We choose to share the same hardware polynomial evaluator between the to implemented functions, based on exponent parity. Nevertheless, if $k_{\text{odd}} = k_{\text{even}} = k-1$ then the reduced argument Y will not have the same weight in the two cases and will have to be tweaked before feeding it to the evaluator:

– In the even case we have $1.f_{-1}\cdots f_{-w_F}$
$= 1 + 0.f_{-1}\cdots f_{-k+1} + 2^{-k+1}0, f_{-k}\cdots f_{-w_F}$.

Listing 8.1 *Emulate function for the polynomial approximating square root*

```
1 void FPSqrtPoly::emulate(TestCase * tc){
2        mpz_class svX = tc->getInputValue("X"); /* Get I/O values */
3        FPNumber fpx(wE, wF);
4        fpx = svX;
5        mpfr_t x, r;
6        mpfr_init2(x, 1+wF);
7        mpfr_init2(r, 1+wF);
8        fpx.getMPFR(x);
9
10       if(correctRounding) {
11              mpfr_sqrt(r, x, GMP_RNDN);
12              FPNumber fpr(wE, wF, r);
13              /* Set outputs */
14              mpz_class svr= fpr.getSignalValue();
15              tc->addExpectedOutput("R", svr);
16       }else { // faithful rounding
17              mpfr_sqrt(r, x, GMP_RNDU);
18              FPNumber fpru(wE, wF, r);
19              mpz_class svru = fpru.getSignalValue();
20              tc->addExpectedOutput("R", svru);
21
22              mpfr_sqrt(r, x, GMP_RNDD);
23              FPNumber fprd(wE, wF, r);
24              mpz_class svrd = fprd.getSignalValue();
25              /* Set outputs */
26              tc->addExpectedOutput("R", svrd);
27       }
28       mpfr_clears(x, r, NULL);
29 }
```

– In the odd case, we need the square root of $2 \times 1.F$
$= 1f_{-1}.f_{-2} \cdots f_{-w_F}$
$= 1 + f_{-1}.f_{-2} \cdots f_{-k+1} + 2^{-k+2}0, f_{-k} \cdots f_{-w_F}$.
As we want to build a single fixed-point architecture for both cases, we align both cases:
$y = 2^{-k+2} \times 0, 0f_{-k} \cdots f_{-w_F}$ in the even case, and
$y = 2^{-k+2} \times 0, f_{-k} \cdots f_{-w_F}0$ in the odd case.

Figure 8.4 presents the generic architecture used for the polynomial evaluation. The internal datapath of the Horner evaluator is trimmed to the bare minimum such that the error budget of $2^{-w_F-1} - \delta$ is still met. For a more detailed insight of corresponding techniques used to minimize hardware evaluation please consult chapter 7.

8.3 Results, comparisons, and some handcrafting

Table 8.1 summarizes the actual performance obtained from the polynomial square root from the FloPoCo 2.2.2. All these operators have been tested for faithful and correct rounding, using FloPoCo's testbench generation framework. The code of the `emulate()` function for the square root operator is given in Listing 8.1. We can observe the simplicity of this function describing the functionality of the proposed architectures. The same correctly rounded branch is also used to test the FloPoCo digit-recurrence version.

The polynomials are obtained completely automatically using FunctionEvaluator operator. When compared with our results previously published in 2010 [64], we have considerably improved the heuristics that define the coefficient sizes and and we now also use truncated multipliers within our generic polynomial approximator which severely reduce DSP usage. The line for double-precision *2010 version* in table Table 8.1 uses 26 DSP blocks whereas the actual version available with FloPoCo 2.2.2 requires just 10 DSPs for roughly 1K slices more. Conservatively estimating that implementing the functionality of the DSP block requires 350 slices (DSP blocks also

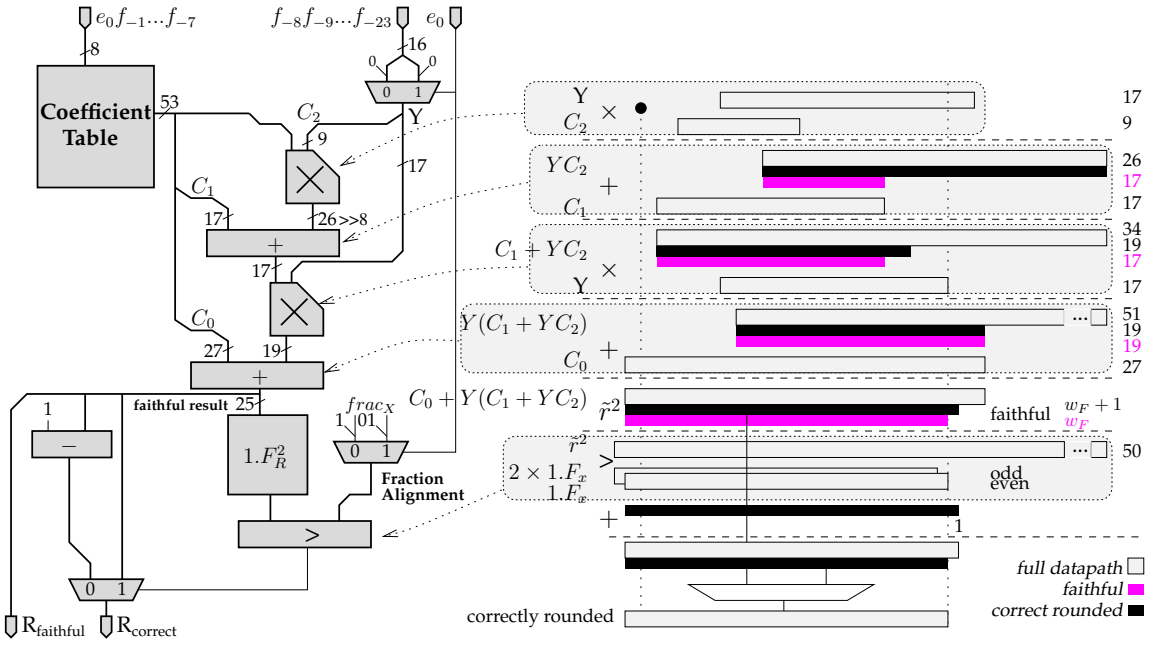


Figure 8.5 *Handcrafted architecture for single precision*

count adders and pipeline registers not accounted for here) we can state that we manage to save 13 DSPs for double precision.

Even so, we still think that there is place for improvement. For illustration, compare the two first lines of Table 8.1. The first was obtained one year ago, as we started the work by designing by hand a single-precision square root using a degree-2 polynomial (Figure 8.5 presents the architecture and the corresponding operand alignments of the datapath). In this context, it was an obvious design choice to ensure that both multiplications were smaller than 17×17 bits. Our current heuristic manages to obtain the same numbers as our handcrafted version for BRAM and DSP count, but it is a bit larger and takes more cycles to compute.

We also hand-crafted a correctly rounded version of the single-precision square root, Figure 8.5, adding the squarer and correction logic described in Section 8.1.2. One observes that it more than doubles the DSP count and latency for single precision (we were not able to attain the same frequency but we trust it should be possible). For larger precisions, the overhead will be proportionally smaller, but disproportionate nevertheless. Consider also that the correctly rounded multiplicative version even consumes more slices than the iterative one. Indeed, it only has the advantage of latency.

8.4 Conclusion and future work

In this chapter we have investigated the best way to compute a square root on a recent FPGA by comparing a state-of-the-art pipelining of the classical digit recurrence, and an original polynomial evaluation algorithm. For large precisions, the latter has the best latency, at the expense of an increase of resource usage. We also observe that the cost of correct rounding with respect to faithful rounding is quite large, and therefore suggest sticking to faithful rounding. In the wider context of FloPoCo, a faithful square root is a useful building block for coarser operators, for instance an operator for $\sqrt{x^2+y^2+z^2}$ (based on the sum of square presented in chapter 4) that would be faithful itself.

Considering the computing power they bring, we found it surprisingly difficult to exploit the embedded multipliers to surpass the classical digit recurrence in terms of latency, performance and resource usage. However, as stated by Langhammer [102], embedded multipliers also bring in other benefits such as predictability in performance and power consumption.

Future works include a careful implementation of a high-radix algorithm, and a similar study around division. The polynomial evaluator that was refined along this work can be used as a building block for many other elementary functions as Chapter 9 will show for the exponential function.

Stepping back, this work asks a wider-ranging question: does it make any sense to invest in function-specific multiplicative algorithms such as the high-radix square root (or the iterative exp and log of [74], or the high-radix versions of CORDIC [123], etc)? Or won't a finely tuned polynomial evaluator, computing just right at each step, be just as efficient in all cases? It seems to be a good idea to invest in function-specific multiplicative algorithms for software implementations of elementary functions [119, 56]. However, for FPGAs which have smaller multiplier granularity, and logic, a finely tuned polynomial evaluator, computing just right at each step, might be as performant, while being generic.

Thanks

The work presented in this chapter is based on a collaboration with Mioara Joldes and Guillaume Revy. I would also like to thank Claude-Pierre Jeannerod for the insightful discussions on the published article on this subject, and also Marc Daumas for pointing to us the high-radix recurrence algorithm. I would also like to thank Jérémie Detrey for this implementation of the digit-recurrence algorithm from FPLibrary, which was recoded in FloPoCo for comparison purposes.

CHAPTER 9
Floating-point exponential

The exponential function is, after the basic arithmetic operators, one of the next most useful building block for floating-point applications. On FPGAs, it has been used for scientific or financial Monte-Carlo simulations [79], for SPICE simulation [94], in phylogenetic tree reconstruction, in quantum chemistry simulations, and in the implementation of the power function [78] among others [156].

9.1 Related work

Several publications have described exponential implementations. Earlier works targeted single precision, first by adapting to FPGAs a software algorithm based on floating-point operations [76], then by using a more efficient fixed-point architecture [70]. This architecture was later improved [79], however the table-based method used there doesn't scale up to double-precision, as the size of the tables grows exponentially with the mantissa size.

As FPGAs are increasingly being used for double-precision floating-point, iterative architectures that scale better [80, 158, 129] were adapted for FPGAs [74]. The architecture in [74] was designed with 5-input LUTs in mind, but is poorly suited to DSP-enabled FPGAs, as Section 9.4.2 will show. It was parameterized in precision, but to our knowledge was never pipelined. Another pipelined, but double-precision only implementation was proposed in [154, 155].

In [132], a CORDIC-based approach using several parallel CORDIC cores was proposed. It has a complex control logic including input and output FIFOs. Being radix-2 CORDIC, it computes one digit per iteration and thus has a very long latency. Moreover, it is based on a floating-point adder, whereas CORDIC is inherently a fixed-point computation, so there is probably room for improvement there.

From a user point of view, the current state of the art is probably the floating-point exponential function `ALTFP_EXP` provided with Altera Megawizard since 2008 [102]. This implementation is parameterized in exponent and mantissa size and fully pipelined. Being included in the standard Quartus releases, it is widely available, although only for Altera targets.

Many other publications have addressed the computation of exponential function in ASIC, e.g. [80, 158, 152, 129]. However, it is difficult to evaluate the relevance of such works on FPGAs.

In the present chapter, we propose yet another architecture for the floating-point evaluation of the exponential function, and its implementation in the open-source FloPoCo project. Its main specificities are the following.

– The algorithm, based on the usual multiplicative range reduction followed by a polynomial approximation, was chosen with DSP blocks and embedded memories in mind, so it makes

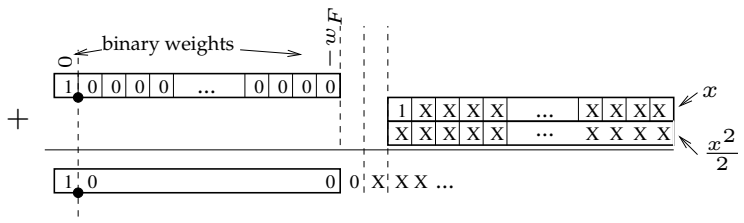


Figure 9.1 *Operand alignment for* $1 + x + x^2/2$ *for* $x < 2^{-w_F-2}$

efficient use of these resources. For instance, the single-precision version now involves just one 17x17-bit multiplier and 18Kbits of dual-port memory, and runs at 375MHz on a Virtex-4, which is a large improvement in all respects over the state of the art [79].
- As we believe that floating-point on FPGA should exploit the flexibility of the target and therefore not be limited to IEEE single and double precision, the algorithm and implementation proposed here are fully parametrized in exponent and mantissa size. They scale to double-precision and beyond.
- The implementation is pipelined to a user-specified frequency. It is last-bit accurate for all supported mantissa sizes.
- The architectures are generated as synthesizable VHDL portable to any FPGA target. In addition, many target-specific optimizations are performed by the FloPoCo framework.
- A novel variation of the KCM algorithm (which initially multiplies and integer by an integer constant) was developed for multiplying an integer by a real constant.
- All this work is freely available as the `FPExp` operator of the FloPoCo project, since version 2.1.0. It comes with test vector generation. In general, it should be immediately usable for application designers.

Section 9.2 gives an overview of the algorithm used, and Section 9.3 discusses some implementation choices. Section 9.4 compares implementation results with the literature, and Section 9.5 concludes.

9.2 Algorithm and architecture

The exponential function is defined on the set of the reals. However, in this floating-point format, the smallest and largest representable numbers are:

	exponent |	1. fraction	value
$X_{\min}$	000...000 |	1.000...000	2^{-E_0}
$X_{\max}$	111...111 |	1.111...111	$(2 - 2^{-w_F}) \cdot 2^{2^{w_E}-1-E_0}$

The exponential should return zero for all input numbers smaller than $\log(X_{\min})$, and should return $+\infty$ for all input numbers larger than $\log(X_{\max})$. In single precision ($w_E = 8$, $w_F = 23$), for instance, the set of input numbers on which a computation will take place is $[-88.03, 89.42]$. In addition, for small x we can use the Taylor series expansion of e^x so we have $e^x \approx 1 + x + x^2/2$. As soon as x is smaller than 2^{-w_F-2} the exponential will return 1. The operand alignment in Figure 9.1 makes this clear.

The reduced exponent range of our implementation is presented in Figure 9.2. One consequence is that testing a floating-point exponential operator should focus on numbers between $X_{\min}$ and $X_{\max}$. In FloPoCo's testbench generator for `FPExp`, the exponent of the random inputs is restricted to $[-w_F - 3, w_E - 2]$.

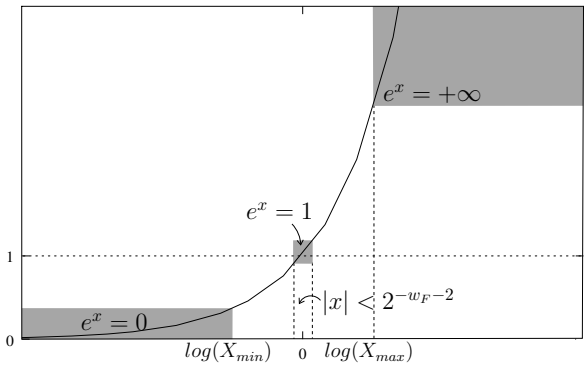


Figure 9.2 *The ranges of the input where the exponential takes specific values*

9.2.1 Algorithm overview

The algorithm used is similar to what is typically used in software [112].
The main idea is to reduce X to an integer E and a fixed-point number Y such as:

$$X \approx E \cdot \log 2 + Y \tag{9.1}$$

where $Y \in [-1/2, 1/2)$ – we will show in section 9.2.2 how to ensure this enclosure.
Then, we may then use the identity

$$e^X \approx 2^E \cdot e^Y \tag{9.2}$$

so E is almost the exponent of the result, and e^Y almost the mantissa. Indeed, if $Y \in [-1/2, 1/2)$, we have $e^Y \in [0.6, 1.7]$, and a mantissa must be $1.F \in [1, 2)$. Thus the exponent and mantissa of the result may be obtained as

$$\begin{cases} R = 2^E \cdot e^Y & \text{if } e^Y \geq 1) \\ R = 2^{E-1} \cdot (2e^Y) & \text{if } e^Y < 1) \end{cases} \tag{9.3}$$

This test boils-down to testing the most significant bit of e^Y, and the multiplication by 2 is just a shift.

The architecture of this operator is given on Figure 9.3. This figure also explicits the alignment of the fixed-point data.

9.2.2 Range reduction

To implement equation (9.1), we have to implement an approximation of

$$E = \left\lfloor \frac{X}{\log 2} \right\rceil \tag{9.4}$$

where $\lfloor x \rceil$ denotes the rounding of x to the nearest integer. Then,

$$Y = X - E \times \log 2. \tag{9.5}$$

If computed infinitely accurately, this would ensure $Y \in [-\frac{\log 2}{2}, \frac{\log 2}{2}]$. On one hand, this is not ideal from an architectural point of view, as Y will later be input to a table and $\frac{\log 2}{2}$ is not a power of two (as $\log 2 \approx 0.34$, the next power of 2 is $1/2$, so only 69% of the table would be used). On the other hand, implementing (9.4) and (9.5) accurately enough would be expensive. A solution

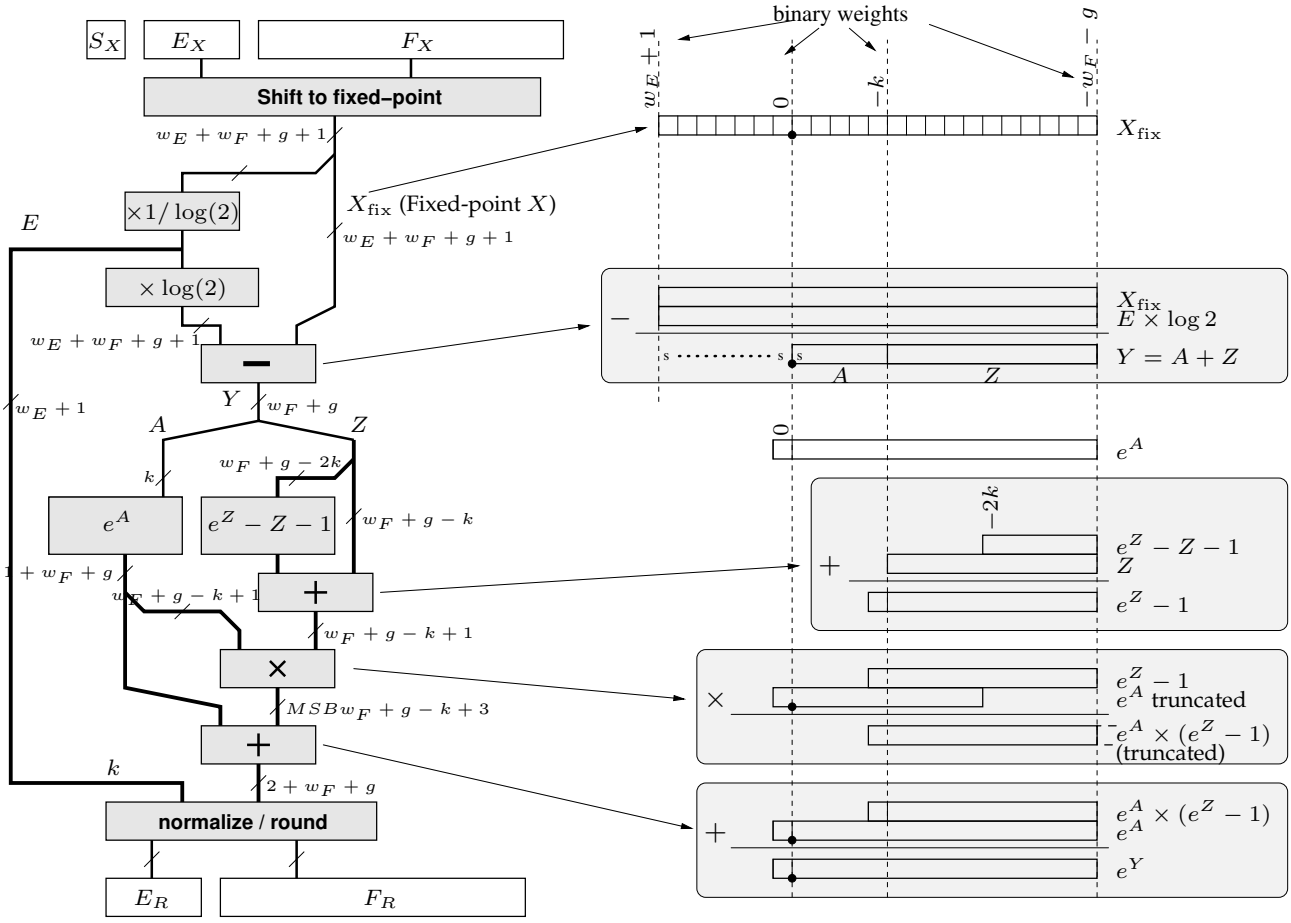


Figure 9.3 *Architecture and fixed-point data alignment*

to both problems is therefore a relaxed implementation of (9.4) that will save on the computation of (9.4) and (9.5) while ensuring $Y \in [-1/2, 1/2)$. The idea is that the computation of E can be grossly approximate, as long as (9.5) is accurately implemented. The normalization process (9.3) will take care of the cases where E was not directly computed as the exact result exponent.

As (9.4) and (9.5) are inherently fixed-point computations, the first task is to build a fixed-point representation X_{fix} of the input X. The most significant bit (MSB) of this representation is provided by the condition $X > \log(X_{\max}) \Rightarrow \exp(X) = +\infty$, from which we deduce $X > 2^{w_E+1} \Rightarrow \exp(X) = +\infty$. The MSB of X_{fix} should therefore have weight w_E. The least significant bit is provided by the condition $X < 2^{-w_F-2} \Rightarrow \exp(x) = 1$, which defines a LSB of weight $-w_F - 2$. Actually, we will improve this accuracy to $-w_F - g$ with $g = 3$ (see below in 9.3.2) to allow for rounding error accumulation in these g guard bits.

Thus the *shift to fixed point* box on Figure 9.3 shifts the mantissa by the value of the exponent. More specifically, if the exponent is positive, it shifts to the left by up to w_E positions (more means overflow). If the exponent is negative, it shifts to the right by up to $w_F + g$ positions. This box also generates out-of-range signals (not shown on the figure).

Let us now turn to the relaxed computation of E which is an integer. Since it is almost the result's exponent (of size w_E), its size in bits will be $w_E + 1$, including one sign bit, the $+1$ preventing overflow in the second case of (9.3).

Let us first determine the error we are allowed to perform in the relaxed computation of E. We will denote this value by E'. This value is directly influenced by the relaxed version of Y,

$Y' \in \left[-\frac{1}{2}, \frac{1}{2}\right]$.

We have $Y' - Y \in \left[-\frac{1}{2} + \frac{\log 2}{2}, \frac{1}{2} - \frac{\log 2}{2}\right] = \left[\frac{-1+\log 2}{2}, \frac{1-\log 2}{2}\right]$. But $E' - E$, which defines the maximum miss-computation of E is:

$$E' - E = \frac{Y - Y'}{\log 2} \in \left[\frac{-1 + \log 2}{2 \log 2}, \frac{1 - \log 2}{2 \log 2}\right] \approx [-0.22, 0.22]$$

This gives us a bound on the error when computing E'. Next, $E = X_{\text{fix}} \frac{1}{\log 2}$. For simplicity we denote $C = \frac{1}{\log 2}$ which gives the value of E, if computed accurately $E = X_{\text{fix}} C$. The miss-computed $E' = X'_{\text{fix}} C'$ where X'_{fix} is X_{fix} truncated to $-w_x$ precision, and C' is the constant rounded-to-nearest on $-w_c$ bits of precision.

$$\begin{aligned} E' - E = &\ X'_{\text{fix}} C' - X_{\text{fix}} C \\ = &\ (X'_{\text{fix}} C' - X'_{\text{fix}} C) + (X'_{\text{fix}} C - X_{\text{fix}} C) \\ = &\ X'_{\text{fix}} (C' - C) + C(X'_{\text{fix}} - X_{\text{fix}}) \end{aligned}$$

where $C' - C$ represents the rounding error on C and is lower than $2^{-w_c - 1}$, $X'_{\text{fix}} - X_{\text{fix}}$ is truncation error on X_{fix} bounded by 2^{-w_x}. and X'_{fix} and C are the magnitudes of the operands. The inequation:

$$(2^{wE} - 1) 2^{-w_c} + 1.4442^{-w_x} < 0.22$$

has a solution $w_x = 3$ and $w_c = 8$ for single precision. Which yields:

$$E' = \left\lfloor \lfloor X_{\text{fix}} \rfloor_{-3} \times \left\lfloor \frac{1}{\log 2} \right\rfloor_{-8} \right\rfloor \tag{9.6}$$

Then, (9.5) may be implemented as

$$Y' = X_{\text{fix}} - E' \times \log 2. \tag{9.7}$$

This fixed-point subtraction cancels the integer part and the first bit of the fractional part.

In this work, we have also considered reducing to $Y \in [0, 1)$ instead of $Y \in [-1/2, 1/2)$. It turns out that guaranteeing this enclosure, especially $Y \geq 0$, is more expensive.

9.2.3 Computation of e^Y

Let us now turn to the computation of e^Y. From here on $Y \in [-\frac{1}{2}, \frac{1}{2}]$. We use a second range reduction, splitting Y as:

$$Y = A + Z \tag{9.8}$$

where A consists of the k most significant bits of Y, and Z consists of the $w_F + g - k$ least significant bits. Then we have

$$e^Y = e^{A+Z} = e^A \cdot e^Z. \tag{9.9}$$

Here e^A will be tabulated in a table indexed by A, and Z is small enough to enable us to use the Taylor formula

$$e^Z \approx 1 + Z + Z^2/2 + ... \tag{9.10}$$

This formula has the advantage that the three first coefficients are powers of two, therefore the corresponding multiplications can be mere shifts. Actually we define

$$f(Z) = e^Z - Z - 1 \tag{9.11}$$

From $0 \leq Z < 2^{-k}$ and $e^Z - Z - 1 \approx Z^2/2 + ...$, we know that the MSB of $f(Z)$ has weight $-2k-1$. As $f(Z)$ will be added to Z, its LSB should have the same weight $-w_F - g$. The useful size of $f(Z)$ is therefore $w_F + g - 2k$. As a consequence, we do not need to compute it out of all the bits of Z. Truncating Z to its $w_F + g - 2k$ MSBs will entail an error of roughly the same weight as the error entailed by the fixed-point format of $f(Z)$.

Out of Z and $f(Z)$, we compute $e^Z - 1 = f(Z) + Z$. This addition may overflow, so the result is on $w_F + g - k + 1$ bits, one more bit than Z.

If $1 + w_F + g < 17$, the final multiplication $e^Y = e^A \cdot e^Z$ may be computed directly as a single DSP block. For larger precisions, the cost of this multiplication is reduced by implementing it as

$$e^A \cdot (1 + Z + f(Z)) = e^A + e^A \cdot (Z + f(Z)) \tag{9.12}$$

Again, the two addends have LSB weight $-w_F - g$. Again, the multiplier inputs need not be more accurate than their output, so we truncate e^A to its LSB $w_F + g - k + 1$ bits.

As we need to truncate the result of this multiplier, we may as well use, for large precisions, truncated multipliers to save DSPs and possibly reduce latency.

A final normalization step possibly shifts left the mantissa by one bit, then performs the final rounding. The rounding consists in possibly adding one bit, then truncating. The IEEE-754 format has the nice property that we may use an adder of size $w_E + w_F + 1$ to add the rounding bit to the concatenated exponent and mantissa: carry propagation from mantissa to exponent will handle the possible exponent change due to rounding up.

9.3 Implementation issues

This computation involves several approximation and rounding errors. The purpose of this section is to guarantee faithful rounding, *ie.* an error of less than one *unit in the last place* (ulp) of the result. Here the ulp has the value 2^{-w_F}, the weight of the last bit of the mantissa $1.F$ of the result.

9.3.1 Constant multiplications

As both constant multiplications (by $1/\log 2$ and $\log 2$) multiply a large constant by a small input, it is natural to use the KCM algorithm [51]. For the larger multiplication by the real value $\log 2$, we actually use a variation that is original to our knowledge and that we briefly present now.

Assume we need to multiply a n-bit integer E by a *real* constant K (here $K = \log 2$), and we want an m-bit result with $m \geq n$. The usual techique is to first round the constant to precision m, then use a fixed-point multiplier (that returns an $n + m$-bit result), then again round the result to m bits. We have two roundings to m bits that each introduces one half-ulp of error on the result, so the final result is accurate to 1 ulp. This accuracy can be improved by rounding the constant to more than m bits. On the implementation side, the multiplication by a constant can use the KCM algorithm [51], and the final rounding costs one addition (truncation is also possible, but then the total error is above 1 ulp). The following technique attains the same accuracy, saving hardware in the KCM, and without needing this final adder.

Let α be the LUT input size of the target FPGA. The input E is split into chunks of size α:

$$E = \sum_{i=0}^{p} 2^{i\alpha} E_i$$

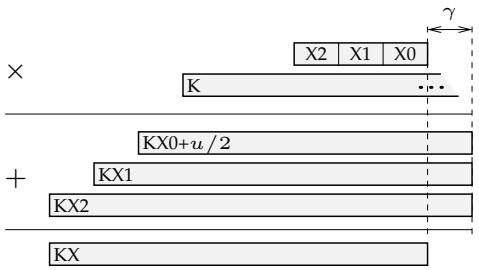


Figure 9.4 *Improved accuracy constant multiplication*

therefore

$$KE = \sum_{i=0}^{p} 2^{i\alpha} KE_i \ .$$

We tabulate in LUTs each product $2^{i\alpha}KE_i$ on just the required precision, so that its LSB has value $2^{-\gamma}u$ where u is the ulp of the result. Here γ is again a number of guard bits. Each table may hold the correctly rounded value of the product of E_i by the *real* value $\log 2$ to this precision, so entails an error of $2^{-\gamma-1}$ ulp. Finally, the first table actually holds $KE_0+u/2$, so that the truncation of the sum will correspond to a rounding of the product. The value of γ is chosen to ensure 1-ulp accuracy. Figure 9.4 presents the operator alignment for this operator when multiplying a n-bit variable by a real constant.

This operator is implemented generically as the `FixRealKCM` operator in FloPoCo. Back to the exponential, as $\alpha \in \{4..6\}$ for current FPGAs, and practical values of E are smaller than 15, the value $\gamma = 2$ is usually enough to ensure that this multiplier returns a faithful multiplication by $\log 2$. For the multiplier by $1/\log 2$ we manually set $\gamma = 3$ to mimic (9.6).

9.3.2 Overall error analysis

In the following, all the errors will be expressed in terms of unit in the last place of Y, which has the value 2^{-w_F-g}. Thus errors expressed this way can be made as small as required by increasing g.

First, note that the argument reduction is not exact. As already stated, numerical errors in the computation (9.6) of E mostly impact the range of Y. Concerning the computation of Y (9.1), there are two exclusive cases:

- If X is large (its exponent is larger than -2), its mantissa is shifted without loss of information, then the computation of $E \times \log 2$ introduces at most one ulp of error in Y as seen in 9.3.1.
- Or, X is small, its mantissa is shifted right beyond the ulp, so its LSBs are lost, which also entails an error of one ulp in Y. However, in this case $E = 0$, so the computation of $E \times \log 2$ is exact.

In both cases we may thus have an error of at most one ulp on Y. Let us now see how it propagates to e^Y.

e^A is tabulated rounded to the nearest, thus with an error of 1/2 ulp.

$e^Z - Z - 1$ is either tabulated (1/2 ulp) or evaluated through polynomial approximation (1 ulp). As the higher order bits of Z are used, the error on Y (which is the error on Z) is scaled down and becomes negligible.

Then $e^Y - 1$ adds the error on Z and the error on $e^Z - Z - 1$, and thus holds an error of 1.5 or 2 ulps.

The error on the other input to the multiplier (e^A truncated) is of one ulp. The product adds these error as $(a + \epsilon) \times (b + \epsilon') = ab + b\epsilon + a\epsilon' + \epsilon\epsilon'$. Here is another subtlety. This formula shows that the error on $e^Z - Z - 1$ is scaled by the value of e^A. Fortunately, the worst case error will occur for $e^A < 1$, since in this case the result will be shifted left by one bit. In the case $e^A > 1$ the error on $e^Z - Z - 1$ may be scaled up (by up to 1.6) but we will have in this case the extra bit of precision needed for the other case, so it doesn't matter.

Truncating the multiplier result would yield another error of one ulp, however we may instead round it (1/2 ulp only) at very little cost by adding its round bit to the right of e^A, so the addition of e^A will also compute the rounding of the product.

Finally the product holds an error of 3 or 3.5 ulps.

Adding the error on e^A, we deduce that the error on e^Y may be up to 3.5 ulp in the dual table case, and 4 ulp in the polynomial case.

If $e^Y < 1$ the final 1-bit shift will multiply this error by 2, so we need $g = 3$ guard bits.

Previous works need more guard bits for the same final accuracy (5 guard bits in [70], 8 in [132] for instance), hence a wider datapath. This improvement in the present work is partly due to a finer error analysis, partly to a refined implementation, in particular of the multiplication by $\log 2$. It is proportionally more important for lower precisions.

However, our implementation also allows increasing the parameter g beyond this minimal value of 3. More guard bits will mean a larger percentage of correctly rounded results. This possibility is also useful when building larger faithful operator based on the exponential, for instance the power function [78] (under development in FloPoCo).

9.3.3 The case study of single precision

Setting $w_F = 23$ and $g = 3$ in the previous architecture, it turns out that $k = 9$ allows for a highly efficient architecture on recent FPGAs.

Firstly, we need altogether $2^9 \times 27$ bits of RAM for e^A and $2^9 \times 9$ bits for $e^Z - Z - 1$. We can group both tables in a single $2^9 \times 36$ table with dual-port access. This perfectly matches one Xilinx BlockRAM, or two Altera M9K.

Secondly, the multiplication is now 18x18 bits, unsigned. This perfectly matches the DSP blocks of Altera chips. On Xilinx chips up to Virtex-4, the multipliers are able of 17x17 unsigned, so the cost is one DSP block plus two 18-bit additions. On Virtex-5 the DSP block is able of 17x24 unsigned, so we only need one addition. One more trick allows us to hide the latency of this addition. We choose to input e^A on 17 bits only instead of 18. To keep the same error bound of one ulp, we now need to round it to 17bits. This rounding requires an addition (so there is no saving compared to extending the multiplier input to 18 bit), but this addition is now before the multiplier, in parallel to the addition of Z to $e^Z - Z - 1$.

9.3.4 Polynomial approximation for large precisions

For larger values of w_F, the generic polynomial evaluator presented in chapter 7 is used as a black box. It inputs a function of $[0, 1] \to [0, 1]$ (here $e^{2^{-k}x} - 2^{-k}x - 1$) with its input and output precisions (given on Figure 9.3) and a degree, and implements a piecewise polynomial approximation. The input interval is decomposed into smaller intervals, and the number of such intervals is computed so that the generated architecture returns a faithfully rounded result. The architectures are optimized for the target FPGA (currently Xilinx Virtex-4, Virtex-5 and Virtex-6, and Altera Stratix II to IV), making efficient use of the DSP blocks to attain high frequencies.

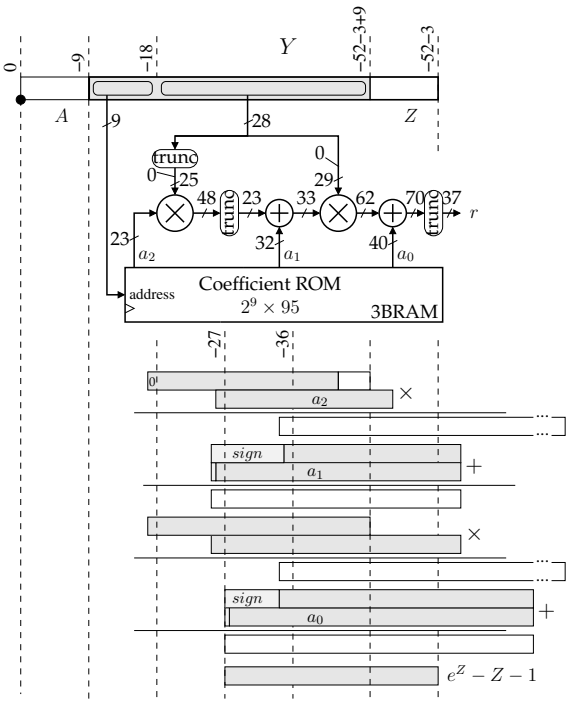


Figure 9.5 *The architecture evaluating $e^Z - Z - 1$ for Virtex-5/Virtex-6*

One advantage of this approach is that it is DSP- and memory-based. Another one is its genericity, as future improvements to the polynomial evaluator will immediately benefit the exponential. This includes the adaptation of the polynomial evaluator to newer FPGAs, but also performance improvements. For instance, we have improved the polynomial evaluator so that it can make use of truncated multipliers to reduce the DSP count, and this has improved FPExp.

More specifically, the function evaluated here is easy to approximate by a low-degree polynomial approximations. It turns out that degree 2 is enough for precision up to double-extended precision.

9.3.5 Parameter selection

We now have two parameters to set: k, that fixes the input to the e^A table, and the degree d of the polynomial, that fixes the trade-off between area of the coefficient table and DSP count/latency. We have varied these parameters to obtain the best trade-offs, that is a an architecture well balanced between DSP and memory consumption, with memories as full as possible and multipliers used as fully as possible. For instance, for double precision, on all targets the best choice is $k = 9$ and a degree-2 approximation on 512 intervals. The FPExp operator provides a good default choice of these parameters, and an expert mode allows the user to set them manually for a different trade-off.

Figure 9.5 details one instance of this architecture for Virtex-5/6.

Table 9.1 *Synthesis results of the various instances of the floating-point exponential operator. We used QuartusII v9.0 for StratixIII EPSL50F484C2 and ISE 11.5 for VirtexIV XC4VFX100-12-ff1152, Virtex5 XC5VFX100T-3-ff1738 and Virtex6 XC6VHX380T-3-ff1923*

Precision	FPGA	Tool	Performance		Resource Usage				
			f(MHz)	Latency	Logic Usage (A)LUTs	Logic Usage Reg.	Logic Usage Slice	DSPs	Memory
(8,23)	StratixIII	Altera MegaWizard	274	17	527	900	-	**19** 18-bit elem.	0
		ours	391	6	832	374	-	**2** 18-bit elem.	0
			405	7	519	382	-		**2** M9K
	VirtexII 1000	[74]	1/123ns	0			728	0	0
	VirtexIV	*ours*	313	14	613	469	338	**1** DSP48	**1** BRAM
			208	7	584	245	318		
	Virtex5	*ours*	349	12	520	436	-	**1** DSP48E	**1** BRAM
			197	6	488	215	-		
	Virtex6	*ours*	401	10	507	458	-	**1** DSP48E1	**1** BRAM
			265	5	445	169	-		
(10,40)	Virtex4	ours* (k=5,d=2)	320	29	1798	1529	1067	**11** DSP48	**3** BRAM
	Virtex5	ours (k=5,d=2)	310	26	1192	1035	-	**8** DSP48E	**3** BRAM
	Virtex5	ours* (k=5,d=2)	216	16	1003	586	-	**8** DSP48E	**3** BRAM
	Virtex6	ours (k=5,d=2)	396	23	1182	1008	-	**8** DSP48E1	**3** BRAM
	Virtex6	ours* (k=5,d=2)	225	14	1058	475	-	**8** DSP48E1	**3** BRAM
(11,52)	StratixIII	Altera MegaWizard	213	25	2941	1476	-	**58** 18-bit elem.	0
		ours	327	29	1307	3757	-	**22** 18-bit elem.	**10** M9K
			256	15	1437	1984	-		
	VirtexII 1000	[74]	1/229ns	0			2045	0	0
	VirtexIV	[154]	?	0	1293	105		**71** DSP48	**6** BRAM
		[155]	200	30	13614	19704		0	**29** BRAM
		[132] (CORDIC)	5.25clk@100Mhz	>61			23455	**36** DSP48	
		ours	319	37	2460	2336	1596		
			178	23	2128	1361	1154	**14** DSP48	**5** BRAM
	Virtex5	*ours*	334	32	1930	1792	-		
			165	18	1647	917	-	**9** DSP48E	**5** BRAM
	Virtex6	*ours*	407	27	1846	1693	-	**9** DSP48E1	**5** BRAM
			225	15	1748	738	-		
(15,64)	Virtex6	ours (k=11, d=2)	353	40	4410	3352	-	**17** DSP48E1	**7** BRAM
(15,112)	Virtex6	ours (k=11, d=4)	360	63	12467	9859	-	**61** DSP48E1	**17** BRAM

9.4 Results

9.4.1 Synthesis results

Table 9.1 provides synthesis results for several precisions and several FPGA targets, and compares with results from previous works. Our approach is clearly the most efficient of the literature for all the precisions. It combines very high frequency (close to the nominal DSP block frequency), the lowest DSP and memory consumption, portability to both Xilinx and Altera targets, last-bit accuracy, flexibility in precision, and also flexibility in terms of latency versus frequency.

Note that the synthesis on Stratix III reports 2 DSP blocks for single precision. One is actually unused. The coarse-grain DSP block structure of Altera chips since Stratix III prevent using the 18×18-bit multipliers completely independently.

Of special interest is the last line of this table, which shows that even a quadruple-precision exponential function will consume only one tenth of the resources of a high-end FPGA while still running at a very high frequency.

9.4.2 Comparison with other works

In [74], a double-precision combinatorial operator consumes, on VirtexII, 2045 slices for a delay of 229 ns. To our knowledge, it was never pipelined, but we estimate that a high-frequency pipelined would require a doubling of the area and roughly 40 cycles.

In addition, this architecture was based on tables inputting α bits and rectangular multipliers where one dimension was also α (an integer parameter) and the other dimension varied from α to the mantissa size. This was a good design choice for LUT-based FPGAs, but it poorly matches the capabilities of the DSP blocks and embedded memories of modern FPGAs. For a short latency, and to use the DSP blocks optimally, one should choose $\alpha = 17$, but then the tables would be much too large (2^{17} entries). Or, one should chose $\alpha \approx 10$, but then the DSPs would be underutilized.

As Altera Megawizard produces readable source files, we analyzed the algorithm used for double precision. The range reduction is the usual one, and the architecture diverges only for the computation of e^Y. Altera's architecture is based on a decomposition of the input as $Y = Y_0 + Y_1 + Y_2 + Y_L$ where Y_0 consists of the 9 leading bits, Y_1 and Y_2 consist of the two following 9-bit chunks, and Y_L consists of the remaining lower bits. The exponential is computed as $e^Y = (e^{y_0} \times e^{y_1}) \times (e^{y_2} e^{y_L})$, where the three first terms are simply read from tables with 2^9 entries, and e^{y_L} is approximated as the Taylor polynomial $e^{Y_L} \approx 1 + Y_L$. This is very similar to the method proposed by Wielgosz et al [154, 155], and both were probably designed independently. However the Altera implementation is generic in precision.

This approach has a potential of lower latency, as the multipliers are organized in tree, and not in sequence as in our proposal. Its drawback is that it doesn't exploit the structure of the numbers. Indeed, the three multiplications are of size roughly 60×60 bits. However, e^{y_1}, e^{y_2}, and e^{y_L} are all of the form $1 + \epsilon$, so at the bit level, we have a lot of predictable multiplications by 0, for which the hardware could be saved. Table 9.1 illustrates this waste of resource compared to our approach.

We also remark in Table 9.1 that the Altera ALTFP_EXP operators do not use 9Kbit embedded memories, although this design would be a perfect match for them (it should consume $(61 + 51 + 42)/18 = 9$ of them, with a corresponding huge reduction in logic resources).

A final remark is that the two references by Wielgosz et al. [154, 155] seem to use the same architecture, however the first one reports results using DSP blocks, while the second one replaces all the DSPs with logic. This actually makes sense, since in this case the parts of the large multipliers that multiply by zero will indeed be optimized out by the synthesizer.

9.4.3 Comparison with microprocessors

This table allows us to compare the theoretical peak performance, in terms of floating-point exponentials, of a large FPGA and a high-end processor. These numbers, of course, should be taken for what they are, as they ignore the critical issue of data movements [155].

The largest Virtex-6 FPGA (XC6VSX475T) could accommodate 168 double-precision exponential cores running above 400 MHz, thus providing a theoretical peak performance over 60 giga double-precision exponentials per second (GDPexp/s).

For a fair comparison, we have to compare to the highest performance software implementation currently available, one which was tuned with comparable effort. To our knowledge, it is the Intel Vector Math Library (VML), which can achieve a peak of 6 cycles/DPExp on Itanium-2 or Core i7. On an 8-core processor running at 3GHz, we obtain a peak performance of 4 GFPExp/s, with a speed-up of 15 in favor of the FPGA. On single precision, the numbers are in excess of 400GSPExp/s for the FPGA while the performance of VML is only improved to 6GSPExp/s. The FPGA speed-up is now above 60.

9.5 Conclusion and future work

We have presented in this chapter a state-of-the-art floating-point exponential operator generator. It produces last-bit accurate architectures for a wide range of FPGA targets, for a wide

range of precisions up to IEEE-754-2008 quadruple precision, and for a wide range of latency/frequency trade-offs. It is designed to make good use of the DSP blocks and embedded memories of high-end FPGAs, and outperforms previous works in performance and resources consumption.

Hopefully, other elementary function of the same quality will join the exponential, forming a complete open-source mathematical library for FPGAs. To this purpose, the case study of the exponential has already lead to improvements in the pipeline framework and the generic polynomial approximator. These will be improved further. This work also suggests that the FloPoCo framework could be enhanced by attaching an optional fixed-point semantics to the signals, which is being investigated.

CHAPTER 10

Floating-point accumulation and sum-of-products

Summing many independent terms is a very common operation. Scalar product, matrix-vector and matrix-matrix products are defined as sums of products. Numerical integration usually consists in adding many elementary contributions. Monte-Carlo simulations also involve sums of many independent terms. Many other applications involve accumulations of floating-point numbers, and some related work will be surveyed in section 10.4.

If the number of summands is small and constant, one may build trees of adders, but to accommodate the general case, it is necessary to design an iterative accumulator, illustrated by Figure 10.1.

It is a common situation that the error due to the computation of one summand is independent of the other summands and of the sum, while the error due to the summation grows with the number of terms to sum. This happens in integration and sum of products, for instance. In this case, it makes sense to have more accuracy in the accumulation than in the summands.

A first idea is to use a standard floating-point adder, possibly with a larger significand than the summands. The problem is that FP adders have long latencies: typically $l = 3$ cycles in a processor, up to tens of cycles in an FPGA (see Table 10.1). This is explained by the complexity of their architecture, illustrated on Figure 10.2. This long latency means that an accumulator based on an FP adder will either add one number every l cycles, or compute l independent sub-sums which then have to be added together somehow. This will add to the complexity and cost of the application, unless at least l accumulations can be interleaved, which is the case of large matrix operations [162, 44]. In addition, an accumulator built out of a floating-point adder is inefficient, because the significand of the accumulator has to be shifted, sometimes twice (first to align both operands and then to normalise the result, see Figure 10.2). These shifts are in the critical path of the loop of Figure 10.1.

In this chapter, we suggest building an accumulator of floating-point numbers which is *tailored to the numerics of each application* in order to ensure that (1) its significand never needs to

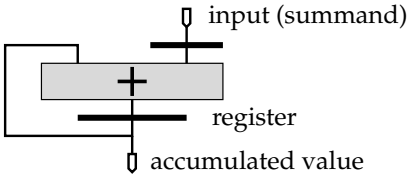


Figure 10.1 *Iterative accumulator*

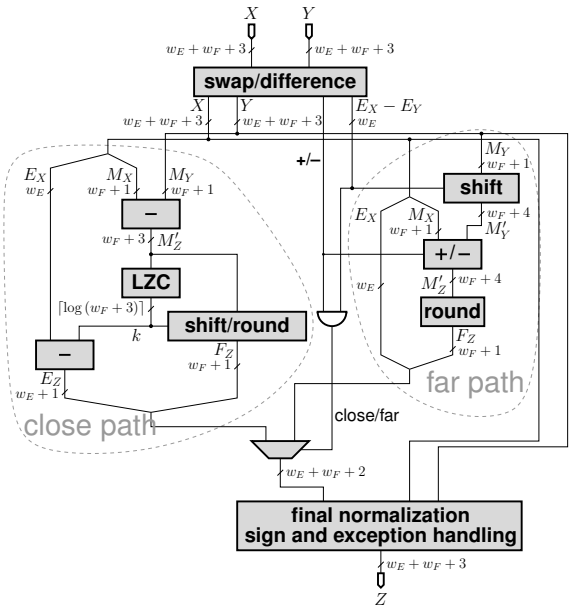


Figure 10.2 *A typical floating-point adder (w_E and w_F are the exponent and significand sizes)*

be shifted, (2) it never overflows and (3) it eventually provides a result that is as accurate as the application requires. We also show that it can be clocked to any frequency that the FPGA supports. We show that, for many applications, the determination of operator parameters ensuring the required accuracy is easy, and that the area can be much smaller for a better overall accuracy. Finally, we combine the proposed accumulator with a modified, errorless FP multiplier to obtain an accurate application-specific dot-product operator.

10.1 A fast and accurate accumulator

This section presents the architecture of the proposed accumulator. Section 10.2 will discuss the determination of its many parameters in an application-specific way.

10.1.1 Overall architecture

The proposed accumulator architecture, depicted on Figure 10.3, removes all the shifts from the critical path of the loop by keeping the current sum as a large fixed-point number. In this figure only the registers on the accumulator itself are shown. The rest of the design is combinatorial and can be pipelined arbitrarily. There is still a loop, but it is now a fixed-point addition for which current FPGAs are highly efficient. Specifically, the loop involves only the most local routing, and the dedicated carry logic of current FPGAs provides good performance up to 64 bits. For instance, a Virtex-4 with speed grade -12 runs such a 64-bit accumulator at more than 220MHz, while consuming only 64 LUTs. Section 10.1.3 will show how to reach even larger frequencies and/or accumulator sizes.

For clarity, some details are not shown on this figure. In particular, `LongAcc` also outputs three sticky bits (input overflow, input underflow, and accumulator overflow), and manages exceptional cases (infinities and Not-a-Number).

Figure 10.4 illustrates the accumulation of several floating-point numbers (represented by their significands shifted by their exponent) into such an accumulator.

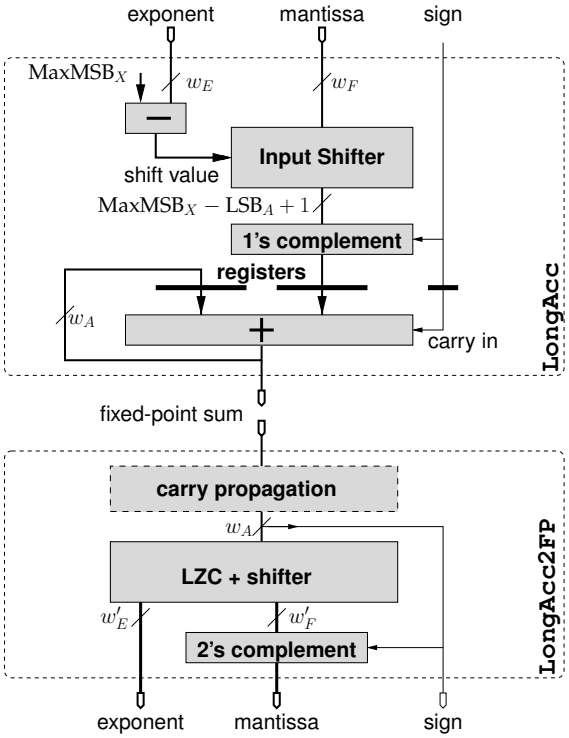


Figure 10.3 *The proposed accumulator (top) and post-normalisation unit (bottom).*

The shifters now only concern the summand (see Figure 10.3), and, being combinatorial, can be pipelined as deep as required by the target frequency.

As seen on Figure 10.3, the accumulator stores a two's complement number while the summands use a sign/magnitude representation, and thus need to be converted to two's complement. This can be performed without carry propagation: If the input is negative, it is first complemented (fully in parallel), then a 1 is added as carry-in to the accumulator. All this is out of the loop's critical path, too.

10.1.2 Parameterisation of the accumulator

Let us now introduce, with the help of Figure 10.4, the parameters of this architecture.

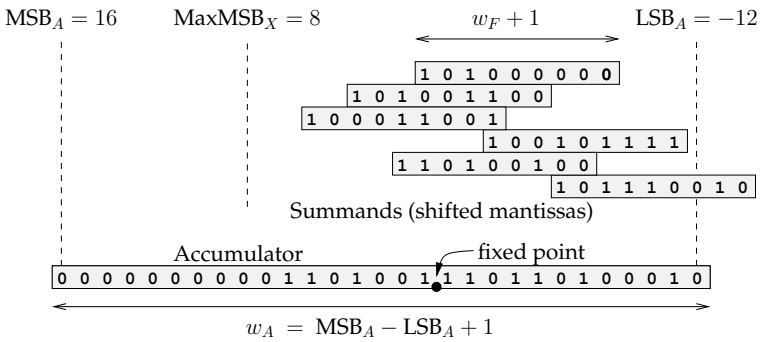


Figure 10.4 *Accumulation of floating-point numbers into a large fixed-point accumulator*

- w_E and w_F are the exponent size and significand size of the summands
- MSB_A is the position of the most-significant bit (MSB) of the accumulator. If the maximal expected sum is smaller than 2^{MSB_A}, no overflow ever occurs.
- LSB_A is the position of the least-significant bit of the accumulator. It will determine the final accuracy as Section 10.2 will show.
- For simplicity we note $w_A = \text{MSB}_A - \text{LSB}_A$ the width of the accumulator.
- MaxMSB_X is the maximum expected position of the MSB of a summand. MaxMSB_X may be equal to MSB_A, but very often one is able to tell that each summand is much smaller in magnitude than the final sum. In this case, providing $\text{MaxMSB}_X < \text{MSB}_A$ will save hardware in the input shifter.

We strongly believe that for most applications accelerated using an FPGA, values of MaxMSB_X, MSB_A and LSB_A can be determined a priori, using a rough error analysis or software profiling, that will lead to an accumulator smaller and more accurate than the one based on an FP adder. This claim will be justified in section 10.2.

This claim sums up the essence of the advantage of FPGAs over the fixed FP units available in processors, GPUs or dedicated floating-point accelerators: We advocate an accumulator specifically tailored for the application to be accelerated, something that would not be possible or economical in a general-purpose FPU.

10.1.3 Fast accumulator design using partial carry-save

If the dedicated carry logic of the FPGA is not enough to reach the target frequency, a partial carry-save representation allows to reach any arbitrary frequency supported by the FPGA. As illustrated by Figure 10.5, the idea is to cut the large carry propagation into smaller chunks of k bits ($k = 4$ on the figure), simply by inserting $\lfloor(\text{MSB}_A - \text{LSB}_A)/k\rfloor$ registers. The critical path is now that of a k-bit addition, and the value of k can therefore be chosen to match the target frequency. This is a classical technique which was in particular suggested by Hossam, Fahmy and Flynn [82] for use as an internal representation in processor FPUs. For $k = 1$ one obtains a standard carry-save representation, but larger values of k are prefered as they take advantage of dedicated carry logic while reducing the register overhead. The FloPoCo implementation computes k out of the target frequency. For illustration, $k = 32$ allows to reach 400MHz on Virtex-4 and StratixII. The additional hardware cost is just the few additional registers – $1/4$ more in our figure, and $1/32$ more for 400MHz accumulation on current FPGAs.

Of course, a drawback of the partial carry-save accumulator is that it holds its value in a non-standard redundant format. To convert to standard notation, there are two options. One is to dedicate $\lfloor(\text{MSB}_A - \text{LSB}_A)/k\rfloor$ cycles at the end of the accumulation to add enough zeroes into the accumulator to allow for carry propagation to terminate. This comes at no hardware cost. The other option, if the running value of the accumulator is needed, is to perform this carry propagation in a pipelined way before the normalisation – this is the carry propagation box on Figure 10.3. The important fact is again that this carry propagation is outside of the critical loop.

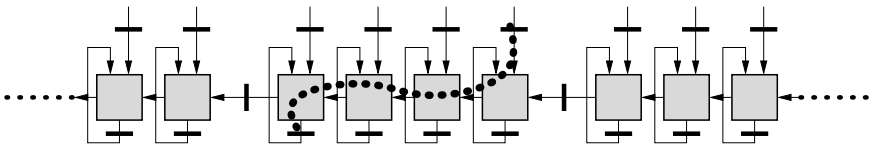


Figure 10.5 *Accumulator with 4-bit partial carry-save. The boxes are full adders, bold dashes are 1-bit registers, and the dots show the critical path.*

10.1.4 Post-normalisation unit, or not

Figure 10.3 also shows the FloPoCo `LongAcc2FP` post-normalisation unit, which performs the conversion of the long accumulator result to floating-point.

Let us first remark, using a few examples, that this component is probably much less useful than the accumulator itself.

In [57], the FPGA computes a very large integration – several hours – and only the final result is relevant. In such applications, it makes no sense to dedicate hardware to the conversion of the accumulator back to floating-point. FPGA resources will be better exploited at speeding up the computation as much as possible, and FloPoCo provides a small helper program to perform this conversion in software.

Another common case is that one needs one normalisation every N accumulations. For instance, a dot product of vectors of size N accumulates N numbers before needing to convert the result back to floating-point. Therefore, in matrix operations, one pipelined `LongAcc2FP` may be shared between N dot product operators [162], at the cost of some multiplexers and routing. Alternatively, one may use N instances of `LongAcc2FP` running at $1/N$ the frequency of the accumulator – they will be smaller. In both cases, it makes sense to provide `LongAcc2FP` as a separate component, as on Figure 10.3. In the following we give separate synthesis results for the accumulators themselves and the post-normalisation unit.

Note that the same discussion holds for an accumulator based on an FP adder of latency l (that actually computes l intermediate subsums). If only the final sum is needed, it may be computed in software at no extra hardware cost. However, if the running sum is needed at each cycle, it will take $l-1$ additions to get it [162, 44].

Back to `LongAcc2FP`, it mostly consists in leading-zero/one counting and shifting, followed by conversion from 2's complement to sign/magnitude, and rounding. If the accumulator holds a partial carry-save value, the carries need to be propagated. This simply requires $\lceil w_A/k \rceil$ pipeline levels, each consisting of one k-bit adder and $\lceil w_A/k \rceil - 1$ registers of k bits, and it can actually be merged with the 2's complement conversion. Again, all this may be performed at each cycle and pipelined arbitrarily.

10.1.5 Synthesis results

All the results presented here are synthesis results obtained for Virtex-4, speedgrade -12, using ISE 11.5 (before place-and-route) and for Stratix-III, fastest speedgrade, using Quartus 10.1 (post place-and-route on an empty FPGA). Post place-and-route results will depend on the FPGA occupation and floorplanning.

Table 10.1 illustrates the performance of the proposed accumulator compared to one built using a floating-point adder from the Xilinx CoreGen tool. These operators are not functionally equivalent. The FP adder-based accumulator either computes an accumulation every l-clock cycles, either needs a supplementary reduction circuit to summing-up the l partial sub-sums. The proposed accumulator is more accurate (Section 10.2.3 will study this quantitatively), but does not return a normalized result as the accumulator based on an FP adder.

For each summand size, we build accumulators of twice the size of the input significand ($\text{MSB}_A = w_F$, $\text{LSB}_A = -w_F$) for two configurations: a small one where $\text{MaxMSB}_X = 1$, and a larger one where $\text{MaxMSB}_X = \text{MSB}_A = w_F$. For single and double precision we additionally list one more configuration $\text{MSB}_A = w^{w_E-1} + 22$, $\text{LSB}_A = -(2^{wE-1} - 1) - w_F$ where $\text{MaxMSB}_X = 2^{w_E-1}$. This configuration allows error-free accumulation of at least 2^{22} floating point numbers on the entire floating-point range of SP ($w_E = 8, w_F = 23$) and DP ($w_E = 11, w_F = 52$). Again, these results are for illustration only: an accumulator should be built in an application-specific way. As section 10.2 will show, a typical accumulator will be between these configurations.

Summand (w_E, w_F)	FPGA	Accumulator	Synthesis Results		
(7,16)	Virtex-4	CoreGen FP adder (w_E, w_F)	317 slices	12 cycles	358 MHz
		$2w_F$ accumulator MaxMSB$_X = 1$	81 slices	3 cycles	451 MHz
		$2w_F$ accumulator MaxMSB$_X$ = MSB$_A$	110 slices	3 cycles	443 MHz
(8,23)	Virtex-4	CoreGen FP adder (w_E, w_F)	482 slices	13 cycles	486 MHz
		$2w_F$ accumulator MaxMSB$_X = 1$	110 slices	3 cycles	391 MHz
		$2w_F$ accumulator MaxMSB$_X$ = MSB$_A$	143 slices	3 cycles	385 MHz
		$22 + 2^{wE}$ acumulator MaxMSB$_X = 2^{wE-1}$	537 slices	4 cycles	335 MHz
	Stratix-III	$2w_F$ accumulator MaxMSB$_X = 1$	164 ALUT 75 REG	2 cycles	491 MHz
		$2w_F$ accumulator MaxMSB$_X$ = MSB$_A$	189 ALUT 97 REG	2 cycles	495 MHz
(10,37)	Virtex-4	CoreGen FP adder (w_E, w_F)	633 slices	14 cycles	421 MHz
		$2w_F$ accumulator MaxMSB$_X = 1$	208 slices	3 cycles	363 MHz
		$2w_F$ accumulator MaxMSB$_X$ = MSB$_A$	271 slices	4 cycles	396 MHz
	Stratix-III	$2w_F$ accumulator MaxMSB$_X = 1$	257 ALUT 117 REG	2 cycles	454 MHz
		$2w_F$ accumulator MaxMSB$_X$ = MSB$_A$	312 ALUT 153 REG	2 cycles	442 MHz
(11,52)	Virtex-4	CoreGen FP adder (w_E, w_F)	839 slices	14 cycles	354 MHz
		$2w_F$ accumulator MaxMSB$_X = 1$	268 slices	3 cycles	350 MHz
		$2w_F$ accumulator MaxMSB$_X$ = MSB$_A$	361 slices	4 cycles	381 MHz
		$22 + 2^{wE}$ acumulator MaxMSB$_X = 2^{wE-1}$	5496 slices	6 cycles	371 MHz
	Stratix-III	$2w_F$ accumulator MaxMSB$_X = 1$	384 ALUT 163 REG	2 cycles	451 MHz
		$2w_F$ accumulator MaxMSB$_X$ = MSB$_A$	463 ALUT 216 REG	2 cycles	460 MHz
	Virtex-II	MPFA [92]	4991 slices 2 BRAM		207 MHz
		AeMPFA [92]	3130 slices 14 BRAM		204 MHz
		FAAC [146]	6252 slices		162 MHz

Table 10.1 *Compared synthesis results for an accumulator based on FP adder, versus proposed accumulator with various combinations of parameters, for Virtex-4 and Stratix-III devices targeting 400 MHz.*

We also give results for DP for three recently published architectures MPFA, AeMPFA from [92] and FAAC from [146] which are complete solutions based on floating-point adders. All these results are given for Virtex-II which essentially has the same architecture as Virtex-4 in what concerns our accumulator but has lower frequencies.

Table 10.2 provides results for the `LongAcc2FP` post-normalization unit. The results prove the portability of our proposed operator, providing good results on both Virtex-4 and Stratix-III FPGAs. Moreover, the integration of this operator into the FloPoCo framework allows exploring a wide range of frequencies, which has in immediate impact on its area and latency, for selecting the best suited operator for a given design.

10.2 Application-specific accumulator design

Let us now justify the claim, made in 10.1.2, that the few parameters of the proposed accumulator are easy to determine on a per-application basis. We acknowledge that the main purpose of floating-point is to free the designer from the painful task of converting a computation on real numbers to fixed-point. Indeed, the proposed accumulator is definitely a floating-point operator, and we hope to convince the reader that the effort it requires to set up is minimal.

10.2.1 A performance vs. accuracy tradeoff

First note that a designer has to provide a value for MSB$_A$ and MaxMSB$_X$, but these values do not have to be accurate. For instance, adding 10 bits of safety margin to MSB$_A$ has no impact on the latency and very little impact on area. Now, from the application point of view, 10 bits means

(w_E, w_F)	FPGA	Freq.(MHz)	LongAcc2FP, $2w_F \to w_F$		
(7,16)	Virtex-4	400	178 slices	9 cycles	445 MHz
		200	116 slices	3 cycles	247 MHz
		100	98 slices	1 cycles	85 MHz
	Stratix-III	400	87 ALUT, 212 REG	7 cycles	461 MHz
		200	151 ALUT, 91 REG	3 cycles	345 MHz
(8,23)	Virtex-4	400	234 slices	10 cycles	411 MHz
		200	153 slices	3 cycles	195 MHz
		100	136 slices	1 cycles	83 MHz
	Stratix-III	400	120 ALUT, 275 REG	7 cycles	444 MHz
		200	193 ALUT, 119 REG	3 cycles	361 MHz
(10,37)	Virtex-4	400	486 slices	13 cycles	364 MHz
		200	282 slices	4 cycles	186 MHz
		100	261 slices	2 cycles	101 MHz
	Stratix-III	400	294 ALUT, 494REG	9 cycles	447 MHz
		200	366 ALUT, 235REG	4 cycles	262 MHz
(11,52)	Virtex-4	400	659 slices	14 cycles	364 MHz
		200	371 slices	4 cycles	182 MHz
		100	386 slices	2 cycles	88 MHz
	Stratix-III	400	370 ALUT, 779 REG	10 cycles	414 MHz
		200	487 ALUT, 324 REG	4 cycles	254 MHz

Table 10.2 *Synthesis results for a LongAcc2FP compatible with Table 10.1, rounding an accumulator of size $2w_F$ to an FP number of size w_F. Virtex-4 results are obtained using ISE 11.5 and for Stratix-III using Quartus 10.1 (after place and route)*

3 orders of magnitude. For most applications, it is huge. A designer in charge of implementing a given computation on FPGA is expected to understand it well enough to bound the expected result with a margin of 3 orders of magnitude. An actual example is detailed below in 10.2.2. As another example, consider a Monte Carlo simulation where the accumulation computes an estimate of the value of a share. No share will go beyond, say, $100,000 before something happens that makes the simulation invalid anyway.

It may be more difficult to evaluate MaxMSB_X. In doubt, $\text{MaxMSB}_X = \text{MSB}_A$ will do, but in many cases application knowledge will help reduce it, hence reducing the input shifter size. For instance, in Monte Carlo simulations, probabilities are smaller than 1. Another option is profiling. A typical instance of the problem may be run in software, instrumented to output the max and min of the absolute values of summands. Again, the trust in such an approach comes from the possibility of adding 20 bits of margin for safety.

In some cases, the application will dictate MaxMSB_X but not MSB_A. In this case, one has to consider the number n of terms to add. Again, one will usually be able to provide an upper bound, be it the extreme case of 1 year running at 500MHz, or 2^{53} cycles. In a worst-case scenario on such simulation times, this suggests the relationship $\text{MSB}_A = \text{MaxMSB}_X + 53$ to avoid overflows. For comparison, 53 is the precision of a DP number, so the cost of this worst case scenario is simply a doubling of the accumulator itself, *but not of the input shifter* which shifts up to MaxMSB_X only. It will cost just slightly more than 53 LUTs in the accumulator (although much more in the post-normalisation unit if one is needed).

The last parameter, LSB_A, allows a designer to manage the tradeoff between precision and performance. First, remark that if a summand has its LSB higher than LSB_A (case of the 5 topmost summands on Figure 10.4), it is added exactly, entailing no rounding error. Therefore, the proposed accumulator will compute exactly if the accumulator size is large enough so that its LSB is smaller than those of all the inputs. Conversely, if a summand has an LSB smaller than LSB_A (case of the bottommost summand on Figure 10.4), adding it to the accumulator entails a rounding error

of at most 2^{LSB_A-1}. In the worst case, when adding n numbers, this error will be multiplied by n and invalidate the $\log_2 n$ lower bits of the accumulator. A designer may lower LSB_A to absorb such errors, an example is given below in 10.2.2. A practical maximum is again an increase of 53 bits for 1 year of computation at 500MHz.

Here we have only discussed the errors due to the accumulation process. In practice, even when a summand is added exactly, it is usually the result of some rounding, so it carries an error of the order of its LSB, which it adds to the accumulator. These summand errors, which are outside of the scope of this work (they can be reduced by increasing w_F), will typically dwarf the rounding errors due to the accumulator. This suggests that the previous worst-case analysis will typically lead to an accumulator that is much more accurate (and bulky) than the application actually requires.

All considered, it is expected that an accumulator will rarely need to be designed larger than 100 bits. Note that the fast carry chain of the smallest Virtex-4 already extends to 128-bit.

Finally, thanks to the sticky output bits for overflows in the summands and in the accumulator, the validity of the result can be checked a posteriori.

10.2.2 A case study

In the inductance computation of [57], physical expertise tells that the sum will be less than 10^5 (using arbitrary units due to factoring out some physical constants), while profiling showed that the absolute value of a summand was always between 10^{-2} and 2.

Converting to bit positions, and adding two orders of magnitude (or 7 bits) for safety in all directions, this defines $\text{MSB}_A = \lceil \log_2(10^2 \times 10^5) \rceil = 24$, $\text{MaxMSB}_X = 8$ and $\text{LSB}_A = -w_F - 15$ where w_F is the significand width of the summands. For $w_F = 23$ (SP), we conclude that an accumulator stretching from $\text{LSB}_A = -23 - 15 = -38$ (least significant bit) to $\text{MSB}_A = 24$ (most significant bit) will be able to absorb all the additions without any rounding error: No summand will add bits lower than 2^{-38}, and the accumulator is large enough to ensure it never overflows. The accumulator size is therefore $w_A = 24 + 38 + 1 = 63$ bits.

Remark that only LSB_A depends on w_F, since the other parameters (MSB_A and MaxMSB_X) are related to physical quantities, regardless of the precision used to simulate them. This illustrates that LSB_A is the parameter that allows one to manage the accuracy/area tradeoff for an accumulator.

10.2.3 Accuracy measurements

Table 10.3 compares for accuracy and performance the proposed accumulator to one built using Xilinx CoreGen in the context of the previous case study. To evaluate the accuracies, we computed the exact sum using multiple-precision software on a small run (20,000,000 summands), and the accuracy of the different accumulators was computed with respect to this exact sum. The proposed accumulator is both smaller, faster and more accurate than the ones based on FP adders. This table also shows that for production runs, which are 1000 times larger, a single-precision FP accumulator will not offer sufficient accuracy.

Table 10.4 provides other examples of the final relative accuracy, with respect to the exact sum, obtained by using an FP adder, and using the proposed accumulator with twice as large a significand. In the first column, we are adding n numbers uniformly distributed in [0,1]. The sum is expected to be roughly equal to $n/2$, which explains that the result becomes very inaccurate for n=1,000,000: As soon as the sum gets larger than 2^{17}, any new summand in [0,1] is simply shifted out and counted for zero. This problem can be anticipated by using a larger significand, or a larger MSBA in the accumulator as we do.

	accuracy	area	latency
SP FP adder acc	$1.2 \cdot 10^{-3}$	482 slices	13 cycles @ 486 MHz
DP FP adder acc	$2.8 \cdot 10^{-15}$	839 slices	14 cycles @ 354 MHz
proposed acc	$2.0 \cdot 10^{-16}$	182 slices	3 cycles @ 451 MHz

Table 10.3 *Compared performance and accuracy of different accumulators for SP summands from [57].*

sum size	rel. error for unif[0, 1]		rel. error for unif[-1, 1]	
	FP adder	long acc.	FP adder	long acc.
1000	-5.76e-05	1.05e-07	-1.59e-05	1.40e-04
10,000	-2.74e-04	1.07e-08	-3.04e-04	2.36e-04
100,000	-4.31e-04	1.07e-09	2.54e-03	-2.73e-04
1,000,000	-0.738	-3.57e-09	3.18e-03	-4.47e-05

Table 10.4 *Accuracy of accumulation of FP(7,16) numbers, using an FP(7,16) adder, compared to using the proposed accumulator with 32 bits ($MSBA = 20$, $LSBA = -11$).*

In the second column, numbers are uniformly distributed in [-1,1]. The sum grows as well (it is a random walk) but much more slowly. As we have taken a fairly small accumulator ($\text{LSB}_A = -11$), for the first sums floating-point addition is more accurate: While the sum is smaller than 1, its LSB is smaller than -16. However, as more numbers are added, the sum grows. More and more of the bits of a summand are shifted out in the FP adder, but kept in the long accumulator, which becomes more accurate. Note that by adding only 5 bits to it ($\text{LSB}_A = -16$ instead of -11), the relative error becomes smaller than 10^{-10} in all cases depicted in Table 10.4: Again, LSB_A is the parameter allowing to manage the accuracy/area tradeoff.

We have discussed in this section only the error of the long fixed-point accumulator itself (the upper part of Fig. 10.3). If its result is to be rounded to an FP(7,16) number using the post-normalisation unit of Figure 10.3, there will be a relative rounding error of at most $2^{-17} \approx 0.76 \cdot 10^{-5}$. Comparing this value with the relative errors given in Table 10.4, one concludes that the proposed accumulator, with the given parameters, always leads to a result accurate to the two last bits of an FP(7,16) number.

10.3 Accurate Sum-of-Products

We now extend the previous accumulator to a highly accurate sum-of-product operator. The idea is simply to accumulate the exact results of all the multiplications. To this purpose, instead of standard multipliers, we use *exact* multipliers which return all the bits of the exact product: For $1 + w_F$-bit input significand, they return a FP number with a $2 + 2w_F$-bit significand. Such multipliers incur no rounding error, and are actually *cheaper* to build than the standard (w_E, w_F) ones. Indeed, the latter also have to compute $2w_F + 2$ bits of the result, and in addition have to round it. In the exact FP multiplier, results do not need to be rounded, and do not even need to be normalised, as they will be immediately sent to the fixed-point accumulator. There is an additional cost, however, in the accumulator, whose input shifter is twice as large.

This idea was advocated by Kulisch [99, 98] for inclusion in microprocessors, but a generic DP version requires a 4288 bits accumulator, which manufacturers always considered too costly to implement. On an FPGA, one may design an application-specific version with an accumulator of 100-200 bits only. This was implemented in FloPoCo, and Table 10.5 provides synthesis results for the DotProduct operator, compared to units built using standard floating-point operators. The

Sum-of-product			Synthesis Results			
			Slices	DSPs	Cycles	Freq.
CoreGen	SP ×	SP +	587	4	23	482 MHz
CoreGen	SP ×	DP +	1011	4	24	354 MHz
ours	SP ×	proposedAcc	363	3	10	356 MHz
CoreGen	DP ×	DP +	1309	16	36	354 MHz
ours	DP ×	proposedAcc	1038	9	20	353 MHz

Table 10.5 *Synthesis results for the sum-of-products operator. The accumulator is designed to absorb at least 100,000 products in [0,1]. The accumulator parameters are* $MSB_A = \lceil \log_2(100,000) \rceil$, $MaxMSB_X = 1$ $MSB_A = -2 * w_F - 2$

Sum-of-product			Accuracy (relative error)					
			unif[0, 1]			unif[-1, 1]		
			1,000	10,000	100,000	1,000	10,000	100,000
CoreGen	SP ×	SP +	4.45e-07	2.09e-06	4.22e-06	4.42e-07	5.09e-06	8.63e-06
CoreGen	SP ×	DP +	4.31e-10	1.15e-10	5.28e-11	1.36e-08	5.01e-10	1.28e-08
ours	SP ×	proposed Acc.	2.99e-15	2.93e-15	2.91e-15	7.15e-16	1.22e-14	2.44e-15
CoreGen	DP ×	DP +	2.22e-15	6.89e-15	2.13e-14	6.83e-16	4.46e-16	9.34e-14
CoreGen	DP ×	QP +	8.25e-19	2.52e-19	4.67e-20	3.34e-16	4.53e-17	4.64e-17
ours	SP ×	proposed Acc.	1.06e-32	1.03e-32	1.01e-32	3.22e-33	3.09e-33	1.56e-32

Table 10.6 *Accuracy results for the sum-of-products operator. The accumulator used had the configuration* $MSB_A = \lceil \log_2(n) \rceil$, $MaxMSB_X = 1$, $MSB_A = -2 * w_F - 2$

accuracy of this operators is tested on synthetic examples in Table 10.6. From these tables we can clearly see that the proposed sum-of-products operator is both smaller, and more accurate.

10.4 Comparison with related work

Much research has been dedicated to converting floating-point computations to fixed-point. When an input vector is to be multiplied by a constant matrix (as happens in filters, FFTs, etc), one may use block floating-point (BFP), a technique known since the 50s and recently applied to FPGAs [13, 35]. It consists in an initial alignment of all the input significands to the largest one (bringing them all to the same exponent), after wich all the computations (multiplications by constants and accumulation) can be performed in fixed-point. The proposed accumulator could be used as a building block for BFP, however it was designed for a much larger class of application, and with a motivation of accuracy inspired by Kulisch's work [99, 98].

The group-alignment based floating-point accumulation technique of He et al [90] applies BFP to arbitrary accumulation. The inputs are first buffered into blocks (called groups here) of size m (with $m = 16$ in the paper). The numbers in a group are added using BFP. Then, these partial sums are fed to a final stage of FP accumulation that may run at $1/m$ the frequency of the first stage, and may therefore use a standard unpipelined FP adder. This is a very complex design (for SP, 443 slices without the last stage, 716 with it). Besides, the frequency of the BFP accumulator will not scale well to higher precisions without resorting to techniques similar to our partial carry save.

Luo and Martonosi [116] have described an architecture for the accumulation of SP numbers that uses two 64-bit fixed-point adders. It first shifts the input data according to the 5 lower bits of the exponent, then sends it to one of the fixed-point accumulators depending on the higher exponent bits. If these differ too much, either the incoming data or the current accumulator is discarded completely, just as in an FP adder. The critical path of the accumulator loop includes one 64-bit adder and a 3-2 compressor. The main problem with this approach (besides its complexity)

is that it is a fixed design that will not scale beyond single-precision. Another one is that the detection of accumulator overflow may stall the operator, leading to a variable-latency design. The authors suggest a workaround that imposes a limit on the number of summands to add.

Zhuo and Prasanna [162], then Bodnar et all [44] have described high-throughput matrix operations using carefully scheduled standard FP adders. Performance-wise, this approach should be comparable to ours. Still, the proposed accumulator is more generic and exposes a finer control of the accuracy-performance tradeoff.

10.5 Conclusion and future work

The accumulator design presented here perfectly illustrates the philosophy of the FloPoCo project: Floating-point on FPGA should make the best use of the flexibility of the FPGA target, not re-implement operators available in processors. The proposed accumulator is deliberately application-specific. In addition it may be tailored to be arbitrarily faster and arbitrarily more accurate than a naive floating-point approach, without requiring more resources.

This approach requires the designer to provide bounds on the orders of magnitudes of the values accumulated. We have shown that these bounds can be taken lazily. In return, the designer gets not only improved performance, but also a provably accurate accumulation process. We believe that this return is worth the effort, especially considering the overall time needed to implement a full floating-point application on an FPGA.

Thanks

The work presented was motivated by partnership between ENS Lyon and Technical University of Cluj-Napoca. I would like to thank our partners, Octavian Creţ and Radu Tudoran for their contributions to this work.

Chapter 11

High-level synthesis of perfect loop nests

In this chapter we are interested in the synthesis of a special class of loop nests into FPGA-specific accelerators, for which the computational datapath generation is done using FloPoCo. Synthesis of loop structures where the inner statements involve deeply pipelined operators (such as the one required in scientific computing), is a challenge when data-dependencies exist between subsequent loop iterations (also called loop-carried dependencies). Unfortunately, most scientific codes using nested loop constructions fall in this category. Having to wait tens or even hundreds of cycles for the result to be available at the pipeline's output before starting the next loop iteration severely impacts performance. Waiting for the result of one iteration before starting the next is not necessary when there are no loop-carried dependencies. Most current HLS tools detect this situation and perform this optimization. However, no satisfactory solution is provided when subsequent iterations do have data-dependencies.

For some applications like matrix-matrix multiplication kernel, which indeed has inter-loop dependencies, hand-coded approaches by Zhuo and Prasanna [162], and Bodnar et al. [44] efficiently use pipelined adders for the reduction operation. For this application (`C` code in Listing 11.1) the two outer loops describe execution of statements having no dependencies. Therefore, the execution of these statements can be used to constantly keep the pipeline busy, maximizing efficiency. The work presented here follows the same spirit, but applies to a general class of applications and is automated to the point that it requires minimal user intervention.

More exactly, we target the class of applications which can be described by perfectly nested loops having uniform data dependencies (see Section 11.2.1 for a terminology reminder) and where the loop bounds are affine expressions of the loop counters. The loops inner statement is implemented as a FloPoCo operator, pipelined for a specific user-defined frequency and deployment FPGA. The operator's pipeline depth is accounted for while rescheduling the code's execution in order to minimize pipeline stalling. This technique is illustrated on the popular matrix-matrix multiply kernel and also on a Jacobi stencil kernel. Next, we consider multiple execution cores for completing one task and we show that hand-guided application-specific parallelizations can often surpass the performance of classical parallelization techniques. Therefore, we propose our one-core scheduler as a stand-alone tool which can be used in the process of parallelizing codes on an application-basis. Finally, we show that the general accuracy of these codes can be improved on FPGAs by using custom formats and accounting for the application's accuracy requirements.

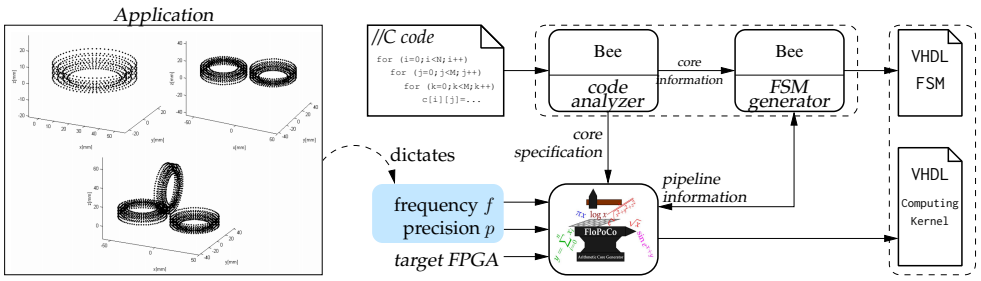


Figure 11.1 *Automation flow: the C code is first parsed by the Bee research compiler; FloPoCo is then invoked for generating the required arithmetic pipeline; the pipeline information is then passed back to the Bee compiler for use in operation scheduling; next, the pipeline depth adjustments are sent to FloPoCo for generating the final VHDL.*

11.1 Computational data-path generation

The generation of FPGA accelerators for a given computational task can be divided into several high-level steps:

- identification and generation of the arithmetic data-path (we will refer from here-on to the arithmetic data-path as an arithmetic operator). This step includes identifying application accuracy requirements and trimming the operator's internal data-path to the bare minimum which still ensures this accuracy.
- scheduling the execution of instructions on the previously arithmetic operator. High-throughput arithmetic data-paths implemented at the previous step generally feature deep pipelines: the challenge at this step is keep the pipeline as busy as possible while at the same time reducing memory accesses.
- if more performance is needed than what can be provided by using only one arithmetic operator, instantiating several operators is a option. This step introduces a new set of challenges in scheduling the computation task.

We delegate the first task: arithmetic datapath generation to be performed using the FloPoCo tool. Using FloPoCo for the arithmetic datapath generation will help minimize circuit's size for a user-given frequency due to the frequency directed pipeline-construction. A similar approach can only be found in Perry's work [128] and implemented in DSP Builder Advanced from Altera. Nevertheless, it is not clear how we could interface our compiler front-end to DSP Builder Advanced as this tool uses a Simulink graphical interface. Moreover, this would limit us to Altera FPGAs.

In the following sections we present an automatic approach for generating computational-kernel specific FSMs. We specify that although the process is conceptually automated, experienced users can intervene at any point to override the default execution of the flow, in order to optimize some steps. Figure 11.1 presents the flow datapath from input-file core specification, to output VHDL generation. The technique used will be presented in the next sections.

11.2 Efficient hardware generation

Given an input program written in C (with limitation which will be made clear) and a set of constraints on the output accuracy, the first step consists in generating an arithmetic operator using FloPoCo which will handle the computational part of the task. Next, starting with this operator, we need to generate the finite state machine (FSM) which controls the execution of the code

```
1 void mmm(float *a, float *b, float *c, int N) {
2   int i, j, k;
3
4   for (i = 0; i < N; i++)
5     for (j = 0; j < N; j++)
6       for (k = 0; k < N; k++)
7         c[i][j] = c[i][j] + a[i][k]*b[k][j];
8 }
```

Listing 11.1 *C routine for the execution a the matrix-matrix multiplication kernel*

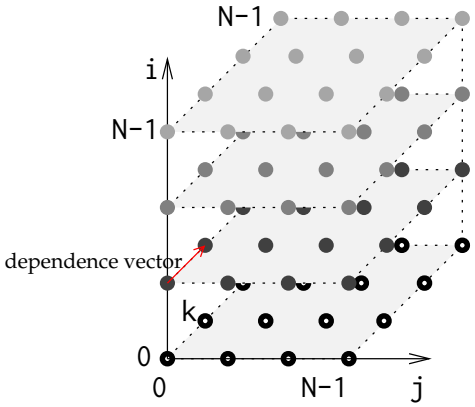


Figure 11.2 *Iteration domain for the matrix-matrix multiply code in Listing 11.1 for N=4*

by scheduling its instructions. The main goal of this step is to optimize the instruction scheduling such that the arithmetic operator is kept busy as much as possible (as few voids in the pipeline). At a higher level, we accomplish this task by reordering the initial program execution. Finally, we generate the corresponding FSM (VHDL code) corresponding to the enhanced program execution scheduling.

11.2.1 Background

In this section we briefly introduce some of the basic notions we need in order to describe our technique. The interested reader should check [84] for more details.

Iteration domains

A *perfect loop nest* is an imbrication of `for` loops where each level contains either a single `for` loop or a single assignment S. A typical example is the matrix-matrix multiply kernel given in Listing 11.1 where line 7 denotes the statement.

Each loop has a counter (i, j and k in our running example from Listing 11.1) which gets initialized when the loop starts, and is modified at each loop iteration (incremented in our example `i++`). Loops also have a continuation condition (`i<N` for example) which decides whether or not the loop will execute.

Writing $\vec{i}_1, ..., \vec{i}_n$ the loop counters, the vector $\vec{i} = (\vec{i}_1, ..., \vec{i}_n)$ is called an *iteration vector*. The set of iteration vectors $\vec{i}$ reached during an execution of the kernel is called an *iteration domain*. The iteration domain for the code in Listing 11.1 is represented by the array of points in Figure 11.2.

The execution instance of S at the iteration $\vec{i}$ is called an *operation* and is denoted by the couple $(S, \vec{i})$. As there is a single assignment in the loop nest, when we refer to an iteration we implicitly acknowledge the execution of the statement of that iteration. The ability to produce program

analysis at the *operation level* rather than at *assignment level* is a key point of our solution. We assume loop bounds and array indices to be an *affine expression* of the surrounding loop counters. Under these restrictions, the iteration domain $\mathcal{I}$ is an invariant polytope.

Dependence vectors

There exist a data dependence from iteration p to iteration q if data produced during iteration p is used at iteration q. A data dependence is *uniform* if it occurs from the iteration $\vec{i}$ to the iteration $\vec{i}+\vec{d}$ for every valid iterations $\vec{i}$ and $\vec{i}+\vec{d}$. In this case, we can represent the data dependence with the vector $\vec{d}$ that we call a *dependence vector*. In the case of matrix-matrix multiplication there exist a dependence on the accumulation in $c[1,0]$ between iteration $i=1$,$j=0$,$k=0$ (which we will denote by (1,0,0)) and iteration (1,0,1). This dependence is denoted by the vector in Figure 11.2.

When array indices are themselves uniform (*e.g.* *a[i-1]*) all the dependencies are uniform. In the following, we will restrict to this case and we will denote by $\mathcal{D} = \{\vec{d_1}, \dots \vec{d_p}\}$ the set of dependence vectors.

Many numerical kernels fit or can be restructured to fit in this model [41]. This particularly includes stencil operations which are widely used in signal processing.

Schedules and affine hyperplanes

The sequential execution of the program processes each iteration in the lexicographic order. In most cases, program optimizations boils down at specifying a new execution order. This can be done by means of a schedule.

A *schedule* is a function θ which maps each point of $\mathcal{I}$ to its execution date. It is convenient to represent execution dates by integral vectors which are processed in lexicographic order: $\theta : \mathcal{I} \to \mathbb{N}^q$.

We consider *linear schedules* $\theta(\vec{i}) = U\vec{i}$ where U is an integral matrix. If there is a dependence from an iteration $\vec{i}$ to an iteration $\vec{j}$, then $\vec{i}$ must be executed before $\vec{j}$: therefore, there the schedule must lexicographically map the execution of $\vec{j}$ before that of $\vec{j}$. We denote this by the ordering on the schedule: $\theta(\vec{i}) \ll \theta(\vec{j})$.

Each line $\vec{\phi}$ of U can be seen as the normal vector to an affine hyperplane $H_{\vec{\phi}}$, the iteration domain being scanned by translating the hyperplanes $H_{\vec{\phi}}$ in the lexicographic ordering. An hyperplane $H_{\vec{\phi}}$ *satisfies* a dependence vector $\vec{d}$ if by "sliding" $H_{\vec{\phi}}$ in the direction of $\vec{\phi}$, the source $\vec{i}$ is touched before the target $\vec{i}+\vec{d}$ for each $\vec{i}$, that is if $\vec{\phi}.\vec{d} > 0$.

We say that $H_{\vec{\phi}}$ *preserves* the dependence $\vec{d}$ if $\vec{\phi}.\vec{d} \geq 0$ for each dependence vector $\vec{d}$. In that case, the source and the target can be touched at the same iteration. $\vec{d}$ must then be solved by a subsequent hyperplane.

We can always find an hyperplane $H_{\vec{\tau}}$ satisfying all the dependencies. Any translation of $H_{\vec{\tau}}$ touches in $\mathcal{I}$ a subset of iterations which can be executed in parallel. In the literature, $H_{\vec{\tau}}$ is usually refereed as *parallel hyperplane*.

Loop tiling

With loop tiling, the iteration domain of a loop nest is partitioned into parallelogram tiles, which are executed atomically. The tiles are executed sequentially, respecting the inter-tile dependencies. For a loop nest of depth n, this requires to generate a loop nest of depth $2n$, the first n

inter-tile loops (the outer loops) describing the execution order of the tiles and the next n *intra-tile* loops (inner loops) scanning the current tile.

A *tile band* is the nD set of iterations described by the last inter tile loop, for a given value of the outer inter tile loops. A *tile slice* is the 2D set of iterations described by the last two intra-tile loops for a given value of the outer loops. See Figure 11.3 for an illustration on the matrix multiply example.

We can specify a loop tiling for a perfect loop nest of depth n with a collection of affine hyperplanes $(H_1, \ldots, H_n)$. The vector $\vec{\phi}_k$ is the normal to the hyperplane H_k and the vectors $\vec{\phi}_1, \ldots, \vec{\phi}_n$ are supposed to be linearly independent. Then, the iteration domain of the loop nest can be tiled with regular translations of the hyperplanes keeping the same distance ℓ_k between two translations of the same hyperplane H_k. The iterations executed in a tile follow the hyperplanes in the lexicographic order, it can be view as "tiling of the tile" with $\ell_k = 1$ for each k. A tiling $\mathcal{H} = (H_1, \ldots, H_n)$ is *valid* if each normal vector $\vec{\phi}_k$ preserves all the dependencies: $\vec{\phi}_k.\vec{d} \geq 0$ for each dependence vector $\vec{d}$. As the hyperplanes H_k are linearly independent, all the dependencies will be satisfied. The tiling $\mathcal{H}$ can be represented by a matrix $U_{\mathcal{H}}$ whose lines are $\vec{\phi}_1, \ldots \vec{\phi}_n$. As the intra-tile execution order must follow the direction of the tiling hyperplanes, U also specifies the execution order for each tile.

Dependence distance

The *distance* of a dependence $\vec{d}$ at the iteration $\vec{i}$ is the number of iterations executed between the source iteration $\vec{i}$ and the target iteration $\vec{i} + \vec{d}$. Dependence distances are sometimes called *reuse distances* because both source and target access the same memory element. It is easy to see that *in a full tile*, the distance for a given dependence $\vec{d}$ does not depend on the source iteration $\vec{i}$ (see Figure 11.5). Thus, we can write it $\Delta(\vec{d})$. However, the program schedule can strongly impact the dependence distance. In the following, managing the dependence distances in accordance with the pipeline depth of the operator will allow us to schedule computations so that the produced data will always be immediately consumed by the operator.

11.2.2 Working examples

In this section we illustrate the feasibility of our approach on two examples. The first example is the matrix-matrix multiplication, that has one uniform data dependency that propagates along one axis. The second example is the Jacobi 1D stencil computation having three uniform data dependencies with different distances.

Matrix-matrix multiplication

The classical code for matrix-matrix multiplication is given in Listing 11.1. The iteration domain is the set integral points lying into a cube of size `N`, as shown in Figure 11.3.

Each point of the iteration domain represents an execution of the assignment `S` with the corresponding values for the loop counters `i`, `j` and `k`. Essentially, the computation boils down to the accumulation of products between elements of A and B. The arithmetic operator needed in this case is has to compute $f(x, y, z) = x + y \times z$. We will use FloPoCo to generate this operator (more on the specifics of this operator will be given later). The architecture of this operator is given in Figure 11.6(a).

There is a unique data dependency in this example. The dependency is carried by the innermost loop `k`, and can be expressed as a vector $\vec{d} = (0, 0, 1)$ (Figure 11.3).

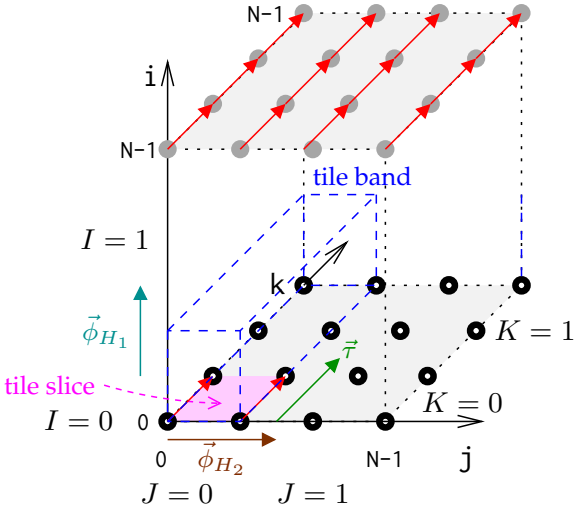


Figure 11.3 *Matrix-matrix multiplication iteration domain with tiling*

```
1   int tsi = 2;
2   int tsj = 2;
3   int tsk = 2;
4   int N=4;
5   for (I = 0; I < N/tsi; I++)
6     for (J = 0; J < N/tsj; J++)
7       for (K = 0; K < N/tsk; K++)
8         for (ii = 0; ii < tsi; ii++)
9           for (kk = 0; kk < tsk; kk++)
10            for (jj = 0; jj < tsj; jj++)
11              c[I*tsi+ii][J*tsj+jj] += a[I*tsi+ii][K*tsk+kk]*b[K*tsk+kk][J*tsj+jj];
```

Listing 11.2 *One valid tiling for the matrix-matrix multiplication*

In the case of using a pipelined operator for implementing $f(x, y, z)$, which is reasonable to assume in a high-throughput scenario, the operator would need to stall for an amount of cycles equal at least to the depth of the floating-point adder used as an accumulator.

We would like to find a better scheduling θ which maximizes the use of our computational resources. Let us consider the affine hyperplane $H_{\vec{\tau}}$ with $\vec{\tau} = (0, 0, 1)$, which satisfies the data dependency $\vec{d}$ and describes a parallel execution front. Each integral point at the intersection of this hyperplane with the iteration domain can be executed in parallel as these points have no dependencies among them. It feels natural to use these points to fill the voids in the pipeline of our arithmetic operator. Therefore, we can keep our operator busy each cycle. However, executing all these independent points (`N` in our case) increases our dependency distance to `N`. If `N` is much larger than m, the number of stages of the FPAdder, we need to store these intermediary results back in the memory. In order to avoid the costly and unnecessary memory activity we find a tiling such that the dependency distance between iteration $\vec{i}$ and $\vec{i} + \vec{d}$ is exactly equal to m ($\Delta(\vec{d}) = m$). Using such a tiling, the data produced at iteration $\vec{i}$ is available for consumption at the adder's output, and consequently also available at the adder's input via the feedback line, exactly when the iteration is started.

The tiling is obtained by first finding a parallel hyperplane $H_{\vec{\tau}}$ (here $\vec{\tau} = (0, 0, 1)$). Next, we complete the tiling by choosing hyperplanes: H_1 with $\vec{\phi}_{H_1} = (1, 0, 0)$ and H_2 with $\vec{\phi}_{H_2} = (0, 1, 0)$ such that $\mathcal{H} = (H_1, H_2, H_{\vec{\tau}})$.

The final tiled loop nest will have the six nested loops: three inter-tile loops `I`, `J`, `K` iterating over

```
1 void jacobi1d(float a[TIME][N]){
2   int i,t;
3   for (t = 0; t < TIME; t++)
4     for (i = 1; i < N-1; i++)
5       a[t][i] = (a[t-1][i-1] + a[t-1][i] + a[t-1][i+1])/3;
6 }
```

Listing 11.3 *1D Jacobi stencil computation*

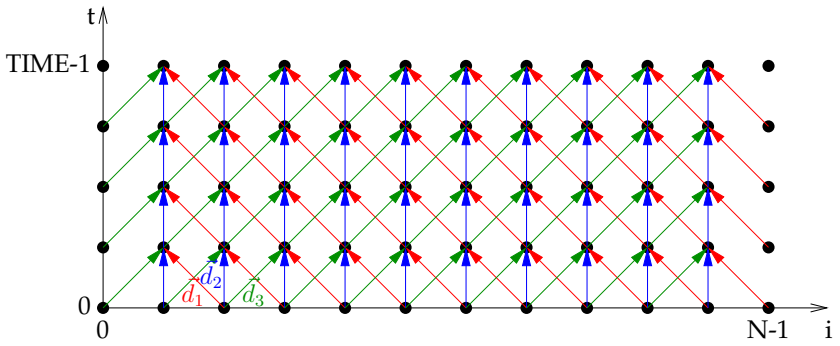


Figure 11.4 *The iteration domain and dependence vectors for 1D Jacobi stencil computation in Listing 11.3*

the tiles, and three intra-tile loops `ii`, `jj`, `kk` iterating into the current tile of coordinate `(I,J,K)`.

For each value of the outermost loop counters `(I,J,K,ii)`, the loops on `jj` and `kk` iterate into a *tile slice*. Figure 11.3 depicts the tile slice for `(I=0,J=0,K=0,ii=0)`.

We schedule each tile slice to execute consecutive iterations on the parallel front. Therefore, the main iteration vector can be expressed as `(I,J,K,ii,kk,jj)`.

We select the width of the tile size (the number of iterations to be performed in the `jj` direction) to be equal to the pipeline depth or our FPAdder, `m`. This ensures that the result produced by the adder is consumed immediately at its input. Thus, it can be fed immediately without any temporary buffering using the feedback connection. The execution order presented above allows to obtain a circuit that computes a temporary value of `c` at each cycle, and stores the temporary data inside the pipeline registers of the arithmetic operators, without any temporary storage buffer. The code corresponding to the valid tiling presented in Figure 11.3 is given in Listing 11.2.

One dimensional Jacobi stencil computation

The kernel is given in Listing 11.3. This is a standard stencil computation with two nested loops. The inner loop iterates over the elements of the array `a` (in the direction of `i`) and the outer loop iterates over the time dimension.

Although it has just two perfectly nested loops, this application poses more problems in execution due to the complex dependencies between elements. The set of dependence vectors has three elements, highlighted in Figure 11.4– $\mathcal{D} = \{\vec{d_1} = (-1,1), \vec{d_2} = (0,1), \vec{d_3} = (1,1)\}$. The iteration space and the dependence vectors are depicted in Figure 11.4.

We apply the same tiling method as in the previous example. The first step consists in finding a parallel hyperplane. One obvious solution for $H_\tau$would be $\vec{\tau} = (0,1)$. Together with the hyperplane H_{ϕ_1} with $\vec{\phi_1} = (1,0)$ this would yield a valid tiling. However, the dependence distances of this tiling are `N-1`, `N` and `N+1` which, as in the case of the matrix-matrix multiplication example are much larger than m, the pipeline depth pf our arithmetic operator.

The technique applied in the case of the previous example consisted in "tiling the tile". How-

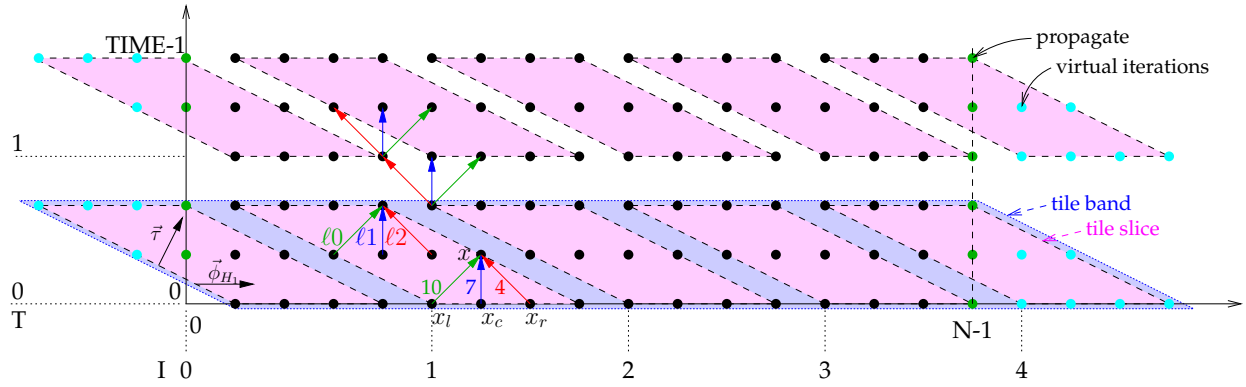


Figure 11.5 *Tiled iteration domain for 1D Jacobi stencil computation*

ever, due to the negative dependencies in $\mathcal{D}$ this technique cannot be applied in our case. The left tile would be executed first, even though it depends on data not produced yet. This violates a valid tiling.

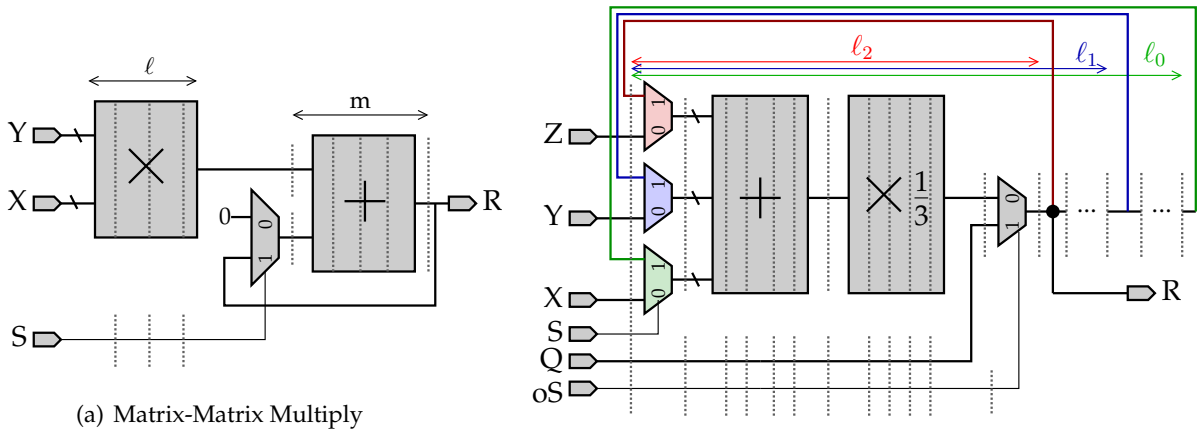
We therefore choose to use a less obvious parallel hyperplane, having the normal vector $\vec{\tau} = (2, 1)$, $H_{\vec{\tau}}$ satisfies all the data dependencies of $\mathcal{D}$. Then, we complete $H_{\vec{\tau}}$ with a valid tiling hyperplane H_1. Here, H_1 can be chosen with the normal vector $(1, 0)$. By analogy with the matrix multiply example, we denote by `(T,I,ii,tt)` the new iteration domain of the resulting tiled loops. Figure 11.5 shows the initial iteration domain divided into several tile slices. Their execution in lexicographic order according to the schedule `(T,I,ii,tt)` is indeed valid because it respects the data dependencies in $\mathcal{D}$.

For this new tiling we compute the dependency distance between the production of the data required by an iteration (we denote it by x for clarity) and its consumption at iteration x. Figure 11.5 highlights the dependency distances for our proposed tiling.

The data produced at iteration x_l (see Figure 11.5) must be available 10 iterations later, x_c must be available 7 cycles later and x_r must be available 4 cycles later. Notice that the dependence distances are the same for any point of the iteration domain, as the dependencies are uniform.

The obvious solution for hardware implementation is to add delay shift registers at the operator's output such that, when executing iteration x the data produced at iterations x_l, x_c and x_r is available at three distinct and precise points of the operator's pipeline. The precise points are given by the values of the dependencies ℓ_0, ℓ_1 and ℓ_2. We choose ℓ_2 to be equal to the operator's pipeline depth. In order to be able to access data produced at x_c at the same time as data produced at x_r we need to add some extra $\ell_1 - \ell_2$ registers. The same technique is applied for synchronizing the consumption of x_l with x_c and x_r: we require $\ell_0 - \ell_1$ extra registers. The architecture is depicted in Figure 11.6(b). Once again, the intermediate value are kept in the pipeline, no additional storage is needed when executing the points in a slice.

As the tiling hyperplanes are not parallel to the original axis, some tiles on the borders would not be full parallelograms. Inside these tiles, the dependence vectors are not longer constant. To overcome this issue, we extend the iteration domain with virtual iteration points where the pipelined operator will compute dummy data. This data is discarded at the border between the real and extended iteration domains (propagate iterations, when `i=0` and `i=N-1`. For the border cases, the correctly delayed data is fed via line `Q` (`oS=1`) in Figure 11.6(b).The C code having the tiled iteration domain is given in Listing 11.4.



(a) Matrix-Matrix Multiply

(b) Jacobi

Figure 11.6 *Computational kernels of our two motivating examples. These were generated using FloPoCo*

```
1   int T,I,ii,tt, TIME, N;
2   int th, tw;
3   for (T = 0; T < TIME/th; T++)
4     for (I=0; I < N/tw; I++)
5       for (ii=0; ii<tw; ii++)
6         for (tt=0; tt<th; tt++)
7           if (I*tw-2*tt+i == 0 || I*tw-2*tt+i == N-1) //propagate
8             a[T*th+tt][I*tw-2*tt+i] = a[T*th+tt-1][I*tw-2*tt+i];
9           else if (I*tw-2*tt+i < 0 || I*tw-2*tt+i > N-1){
10            //dummy: virtual iteration points
11          }else
12            a[T*th+tt][I*tw-2*tt+i] = (a[T*th+(tt-1)][I*tw-2*(tt-1)+(i-1)]+
13                                       a[T*th+(tt-1)][I*tw-2*(tt-1)+ i ]+
14                                       a[T*th+(tt-1)][I*tw-2*(tt-1)+(i+1)])*1/3;
```

Listing 11.4 *1D Jacobi stencil computation*

11.2.3 Parallelization

In this section we are interested in mapping these applications to multiple computing kernels in order to improve performance. We show here that the same methodology we have used for mapping the application onto a single computing kernel can effectively be used to generate the corresponding FSMs of the computing cores in a parallelization scenario.

Parallelizing the matrix-matrix multiplication kernel can be seen as simple due to the fact that both external loops i and j carry no dependencies. However, this is not entirely true if we want this parallelization to be efficient as well, with regard to memory transfers.

A naive implementation of a single computing kernel performing $C = AB$ requires $4N^3$ memory accesses: $N^3\,(read(a) + read(b) + read(c) + store(c))$. At each step two elements are ready from A and B together with the destination accumulator from C. After the computation is done, the corresponding element from c is updated in the memory. By using our technique to reschedule the execution of this core we avoid having to read and update c at each iteration step, as its value is stored inside the pipeline's registers: $N^2(N(read(a) + read(b)) + store(c))$.

We can additionally reduce this cost if we are provided with local memory. *Blocking* consists is splitting the input matrices into blocks which are fetched in pairs into the local memory. Figure 11.7 illustrates this technique. For a given block-size $p \times q$ (where we suppose for simplicity that both p and q divide N) and suppose we are provided with $2(p \times q) + (p \times p)$ local memory for buffering (sufficient to store one block from A,B and C), the external memory requirement is:

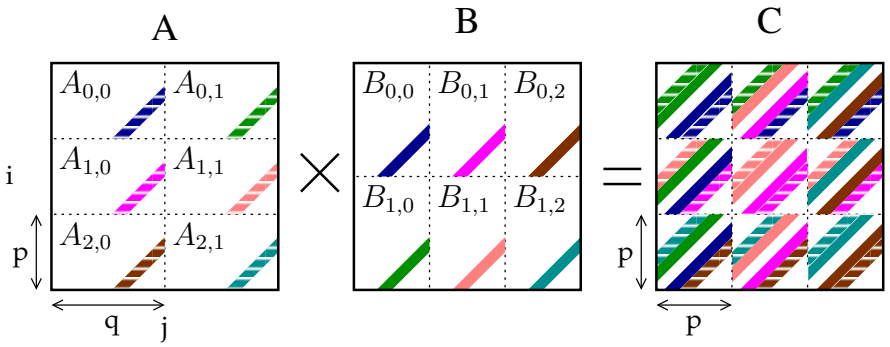


Figure 11.7 *Matrix-matrix multiply using blocking*

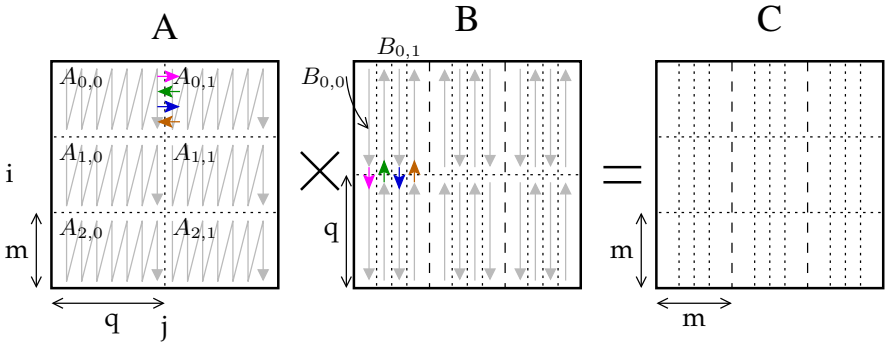


Figure 11.8 *Matrix-matrix multiply blocking applied using our technique. Scheduling of computations is modified in order to minimize external memory usage*

$$M = 2\frac{N}{p}\frac{N}{q}\frac{N}{p}(p \times q) + \left(2\frac{N}{q} - 1\right)\frac{N^2}{p^2}(p \times p)$$
$$= 2\frac{N^3}{p} + \left(2\frac{N}{q} - 1\right)N^2$$

The technique trades local memory requirement for memory bandwidth. For $p = q = N$ it reduces to storing locally the three matrices $3N^2$ buffer. The bandwidth requirement is $2N^2$ for fetching A and B and N^2 for writing C.

When the execution schedules the processing of consecutive memory blocks in the direction of j: $A_{0,0} \times B_{0,0}$, $A_{0,1} \times B_{1,0}$ etc. the same block C block will get affected, and is therefore possible to skip its writing to memory until the last product affecting it was processed ($C_{0,0}$ is written to the main memory only when $A_{0,1} \times B_{1,0}$ was complete. This reduces our memory bandwidth to $2\frac{N^3}{p} + N^2$. Now, by applying our scheduling technique, we are able to process entire computation without even needing a buffer for the C block (its values are stored inside the operator's pipeline levels). The current technique requires freezing the computational kernels the time needed to fetch a new pair of blocks from A and B.

Consider the Figure 11.8 which illustrates how our scheduling algorithm would perform if blocking was used. Note that m denotes the number of stages of our accumulator (see Figure 11.6(a)). The points executed in the i direction of are on parallel front and therefore have no data dependencies. While m is fixed by the operator's pipeline depth, the size of the internal memory

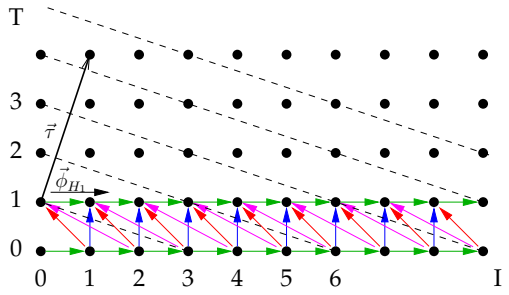


Figure 11.9 *Inter tile slice iteration domain for Jacobi 1D stencil code. The parallel hyperplane has $\vec{\tau} = (1, 3)$ and describes the tile-slices which can be executed in parallel. The dashed lines indicated various translations of the hyperplane $H_{\vec{\tau}}$ showing different levels of parallelism.*

dictates the size of q.

When sufficient local memory is available, a second well known technique, *double buffering*, is used to interlacing memory access and computations. Provided we are assigned twice the local memory we need for our enhanced blocking, $2 \times 2(p \times q)$, the idea is to fetch the next set of blocks from A and B for computation at time $t + 1$ while performing the computing stage at time t. This said, when a variable is reused on successive tiles, it is better to load it one time for all, and to avoid reloading it for each tile. An exact solution to this problem has been found recently [131]. The objective now is to try to reuse the same fetched block as much as possible.

The execution schedule is optimized such to maximize the use of the A block buffer. Successive blocks of A and B (A is by far more costly with a size of $m \times q$ whereas B has a size $q \times 1$) are fetched from the memory in the direction of j for A and i for B. Once the edge is reached (say we have finished processing $A_{0,1} \times B_{1,0}$), we keep $A_{0,1}$ (which would be costly to discard) and we load $B_{1,1}$ instead. We can clearly execute the accumulation on C iterating from $N - 1$ towards 0. This saves an important amount of external memory accesses particularly when implementing the double buffering technique.

Now, finally we consider using multiple processing elements to accomplish the task. It is easy too see that up to m PEs can work on the same block of A and on m different blocks of B ($B_{ml,m(l+1)-1}$). The local memory requirement is as much $2 \times m \times q$ for such a case (m PEs). The size of m can be increased within reasonable limits due to the embedded memories which can act as shift-registers in modern FPGA devices. Nevertheless, it is much more likely that the external memory bandwidth will be the real limitation.

11.2.4 One dimensional Jacobi stencil computation

In this section we will present two solutions to parallelize the Jacobi 1D stencil execution. The first solution is based on classical parallel execution of tile slices. Consider the execution of the tile slices in Figure 11.5. Finding what tile slices can be executed in parallel reduces to finding a hyperplane parallel $H_{\vec{\tau}}$ which in the new iteration domain of the tile slices.

The new iteration domain and the corresponding hyperplane $H_{\vec{\tau}}$ are depicted in Figure 11.9. The normal vector $\vec{\tau} = (1, 3)$ indicates that the maximum degree of parallelism is $\lceil N/3 \rceil$. One could increase this to $\lceil N/2 \rceil$ at the expense of performing a different tiling than Figure 11.5 shows. In the new tiling the tile slices at $T = 1$ would be described by the transition of the same hyperplane $H_{\vec{\tau}}$ as for $T = 0$. This increase the complexity of the border conditions (where we propagate or execute virtual points). We believe that the complexity of the conditions in such an implementation would severely affect the performance of our FSM and we did not consider it further.

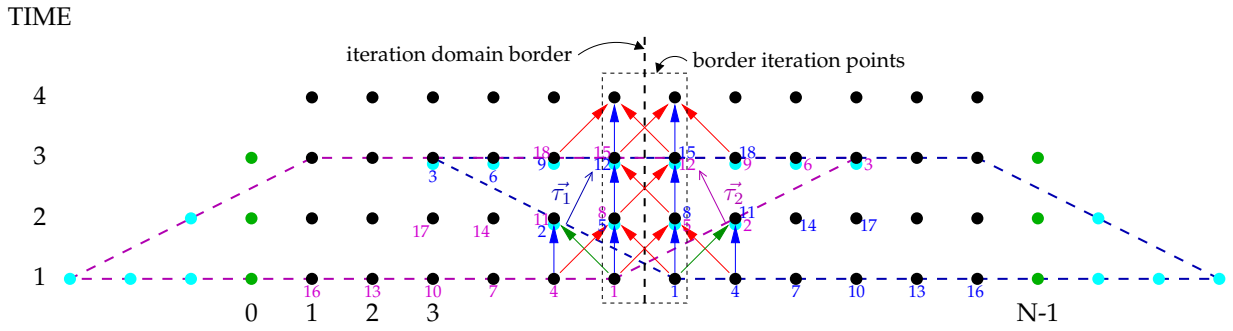


Figure 11.10 *An alternative to executing the Jacobi Kernel using 2 processing elements.*

Our second proposed parallelization solution will be described next. It was initially supposed to be example-specific, however its execution can be extended to some reduced set of application classes presenting dependence symmetries. The benefits of this solution are: a wider degree of parallelism in execution and a reduced local memory size.

Figure 11.10 presents the basic principle behind our proposed solution for two PEs. The iteration domain is split into two parts (suppose for clarity that N is even in this example): right part is tiled as previously described in Figure 11.5 and the left part part tiling is mirrored (symmetrical) to that on the right.

The tile slices intersect the neighboring iteration domains. The set of points described by this intersection represent virtual iteration points.

The border iteration points carry the dependencies between the tile slices of neighboring iteration domains. On these points, the green incoming dependence represents a datum computed by neighboring PE which must be communicated. Thanks to the symmetry of the execution schedule, two symmetric iteration points are executed at the same time. This means that two symmetric border iteration points are executed at the same time. Consider for example the iteration points executed at time 1 on Figure 11.10, say P1 on the left and P2 on the right, and consider the red dependence starting from P1 to a point P3 executed by the right PE. The corresponding datum should be communicated exactly at the execution of P3, which is the same as the symmetric of P3 in the left PE. This means that the left PE should communicate the datum as for a vertical dependence.

From the architecture perspective this involves widening the green multiplexer of each accelerator with one input from the neighboring blue extraction point and modifying the select line of the multiplexer so to fetch the correct data for these border points.

Figure 11.11 illustrates the simplicity of this architecture. When recursively instantiating multiple pairs of accelerators the tails of the tile slices will similarly overlap. The border iteration point at this intersections will be solved by the blue dependency from neighbor. Consequently, the red multiplexer will have a third input fed from from the second neighbor's blue dependency.

Notice that this method could be easily applied to any stencil computation. The only difficulty is to insert a wire to communicate the data at the relevant time. Indeed, it can happen that the symmetric of P3 is not targetted by a dependence starting from P1. In this case, the execution distance with P1 should be computed as in the step c, and extra wire/registers should be added.

11.2.5 Lessons

In this section, we have derived by hand several parallel pipelined accelerators by following different methodologies. We have started from the sequential accelerators generated with the

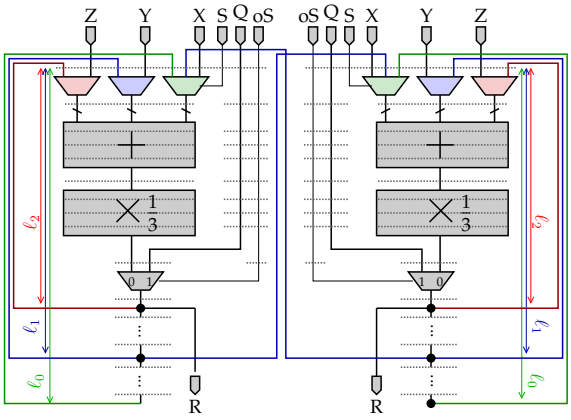


Figure 11.11 *Architecture for the second proposed parallelization of Jacobi 1D*

technique described in the previous section.

For data parallel examples like matrix multiplication the parallelization is trivial and consists in instantiating multiple parallel computational cores each having assigned a subdomain of the global iteration domain.

Unfortunately, for examples like Jacobi 1D, the parallelization is not trivial. Due to many data dependencies, the parallel hyperplanes are skewed. There exist an infinite number of such parallel hyperplanes. One has to chose a tradeoff between maximizing the parallelism and not increasing dramatically the number of delay registers. The second solution that consists in cutting the domain into subdomains which execute using a mirror-like schedule seems to be more adapted for stencil examples as it benefits the most from FPGA structure and fast direct links between adjacent computational cores. This solution should be used for stencil examples on FPGA platforms and could be easily automatized.

11.2.6 Algorithm

In the following we formalize the ideas presented intuitively on our working examples and present a two-step algorithm to translate a loop kernel written in C into an hardware accelerator using pipelined operators efficiently. Firstly, we describe how to get the tiling followed by an explanation on how to generate the control FSM respecting the schedule induced by the loop tiling.

Step 1: Scheduling the kernel

The key idea is to tile the program in such a way that the distance associated to each dependence is constant. Then, it would be always possible to reproduce the solution described for the Jacobi 1D example.

The only issue is to ensure that the minimum dependence distance is equal to the pipeline depth of the FloPoCo operator. The idea presented on the motivating examples is to force the last intra-tile inner loop L_{par} to be parallel. This way, for a fixed value of the outer loop counters, there will be no dependence among iterations of L_{par}. The dependencies will all be carried by the outer-loops, and then, the dependence distances will be fully customizable by playing with the tile size associated to the loop enclosing immediately L_{par}, L_{it}.

This amounts to finding a parallel hyperplane $H_{\vec{\tau}}$ (*step a*), and to complete it with others hyperplanes $H_1, \ldots, H_{n-1}$ (assuming the depth of the loop kernel is n) in order to form a valid

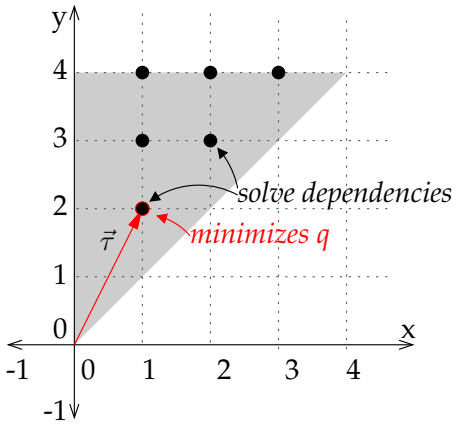


Figure 11.12 *The solution to the ILP finding τ for the Jacobi example*

tiling (*step b*).

Now, it is easy to see that the hyperplane $H_{\vec{\tau}}$ should be the (n-1)-*th* hyperplane (implemented by L_{it}), any hyperplane H_i being the last one (implemented by L_{par}). Roughly speaking, L_{it} pushes $H_{\vec{\tau}}$, and L_{par} traverses the current 1D section of $H_{\vec{\tau}}$, feeding the pipeline with parallel point.

It remains in *step c* to compute the actual dependence distances as an affine function of tile sizes. Then, given a FloPoCo operator with a certain minimum pipeline depth m we can easily find a proper tile size for which the minimum dependence distance is $\geq m$. For the remaining dependence distances, ($\geq m$) one needs then to insert shift registers at the output of the operator's pipeline in order to keep all the dependencies of a point x inside the pipeline. We will detail these three steps in the following.

Step a. Find a parallel hyperplane $H_{\vec{\tau}}$

This can be done with a simple integer linear program (ILP). Here are the constraints:

- $\vec{\tau}$ must *satisfy every dependence*: $\vec{\tau} \cdot \vec{d} > 0$ for each dependence vector $\vec{d} \in \mathcal{D}$.
- $\vec{\tau}$ must *reduce the dependence distances*. Notice that the dependence distance is increasing with the decrease in angle between $\vec{\tau}$ and a dependence vector $\vec{d}$.
  Also notice that the value of the inner product $(\vec{\tau} \cdot \vec{d})$ is increasing with the decrease in angle between $\vec{\tau}$ and a dependence vector $\vec{d}$. It is therefore sufficient to minimize the quantity: $q = \max(\vec{\tau} \cdot \vec{d_1}, \ldots, \vec{\tau} \cdot \vec{d_p})$.
  We build the constraints $q \geq \vec{\tau} \cdot \vec{d_k}$ for each k between 1 and p, which is equivalent to $q \geq \max(\vec{\tau} \cdot \vec{d_1}, \ldots, \vec{\tau} \cdot \vec{d_p})$.

It remains to find the objective function. We want to minimize q. Then, for the minimal value of q, we want to minimize the coordinates of $\vec{\tau}$. This amounts to look for the *lexicographic minima of the vector* $(q, \vec{\tau})$. This can be done with standard ILP techniques [83]. On the Jacobi1D example, this gives the following ILP, with $\vec{\tau} = (x, y)$:

$$\begin{array}{ll} \min_{\ll} & (q, x, y) \\ \text{s.t.} & y - x > 0 \wedge y > 0 \wedge x + y > 0 \\ & q \geq x - y \wedge q \geq x + y \wedge q \geq x \end{array}$$

The ILP is solved in Figure 11.12 for the Jacobi example.

Step b. Find the remaining tiling hyperplanes

Let us assume a nesting depth of n, and let us assume that $p < n$ tiling hyperplanes $H_{\vec{\tau}}$,

$H_{\vec{\phi}_1}, \ldots, H_{\vec{\phi}_{p-1}}$ were already found. We can compute a vector $\vec{u}$ orthogonal to the vector space spanned by $\vec{\tau}, \vec{\phi}_1, \ldots, \vec{\phi}_{p-1}$ using the internal inverse method [45]. Then, the new tiling hyperplane vector $\vec{\phi}_p$ can be built by means of ILP techniques with the following constraints.
- $\vec{\phi}_p$ must be a *valid tiling hyperplane*: $\vec{\phi}_p.\vec{d} \geq 0$ for every dependence vector $\vec{d} \in \mathcal{D}$.
- $\vec{\phi}_p$ must be *linearly independent* to the other hyperplanes: $\vec{\phi}_p.\vec{u} \neq 0$. Formally, the two cases $\vec{\phi}_p.\vec{u} > 0$ and $\vec{\phi}_p.\vec{u} < 0$ should be investigated. As we just expect the remaining hyperplanes to be valid, without any optimality criteria, we can restrict to the case $\vec{\phi}_p.\vec{u} > 0$ to get a single ILP.

Any solution of this ILP gives a valid tiling hyperplane. Starting from $H_{\vec{\tau}}$, and applying repeatedly the process, we get valid loop tiling hyperplanes $\mathcal{H} = (H_{\vec{\phi}_1}, \ldots, H_{\vec{\phi}_{n-2}}, H_{\vec{\tau}}, H_{\vec{\phi}_{n-1}})$ and the corresponding tiling matrix $U_{\mathcal{H}}$. It is possible to add an objective function to reduce the amount of communication between tiles. Many approaches give a partial solution to this problem in the context of automatic parallelization and high performance computing [45, 115, 160]. However how to adapt them in our context is not straightforward and is left for future work.

Step c. Compute the dependence distances

Given a dependence vector $\vec{d}$ and an iteration $\vec{x}$ in a tile slice the set of iterations $\vec{i}$ executed between $\vec{x}$ and $\vec{x} + \vec{d}$ is exactly:

$$D(\vec{x}, \vec{d}) = \{\vec{i} \mid U_{\mathcal{H}}\vec{x} \ll U_{\mathcal{H}}\vec{i} \ll U_{\mathcal{H}}(x + \vec{d})\}$$

Remember that $U_{\mathcal{H}}$, the tiling matrix computed in the previous step, is also the intra-tile schedule matrix. By construction, $D(\vec{x}, \vec{d})$ is an integral polyhedron (conjunction of affine constraints). Then, the dependence distance $\Delta(\vec{d})$ is exactly the number of integral points in $D(\vec{x}, \vec{d})$ (that does not depend on $\vec{x}$). The number of integral points in a polyhedron can be computed with the Ehrhart polynomial method [55] which is implemented in the polyhedral library [10]. Here, the result is a degree 1 polynomial in the tile size ℓ_{n-2} associated to the hyperplane H_{n-2}, $\Delta(\vec{d}) = \alpha\ell_{n-2} + \beta$. Then, given a fixed input pipeline depth δ for the FloPoCo operator, two cases can arise:
- Either we just have *one dependence*, $\mathcal{D} = \{\vec{d}\}$. Then, solve $\Delta(\vec{d}) = \delta$ to obtain the right tile size ℓ_{n-2}.
- Either we have *several dependencies*, $\mathcal{D} = \{\vec{d_1}, \ldots, \vec{d_p}\}$. Then, choose the dependence vectors with smallest α, and among them choose a dependence vector $\vec{d_m}$ with a smallest β. Solve $\Delta(\vec{d_m}) = \delta$ to obtain the right tile size ℓ_{n-2}. Replacing ℓ_{n-2} by its actual value gives the remaining dependence distances $\Delta(\vec{d_i})$ for $i \neq m$, that can be sorted by increasing order and used to add additional registers to the FloPoCo operator in the way described for the Jacobi 1D example (see Figure 11.6(b)).

Step 2: Generating the control FSM

This section explains how to generate the FSM that will control the pipelined operator according to the schedule computed in the previous section. A direct hardware generation of loops, which is usually used, would produce multiple synchronized FSMs, each FSM having an initialization time (initialize the counters) resulting in an operator stall on every iteration of the outer loops. We avoid this problem by using the Boulet-Feautrier algorithm [46] to generate a single loop that executes one instruction per iteration.

The method takes as input the tiled iteration domain and the scheduling matrix ($U_{\mathcal{H}}$) and uses ILP techniques to generate two functions: `First` and `Next`. The operation returned by `First` represents the first operation to be executed.

Then, the Next function computes the next operation to be executed given the current operation. The generated code looks like:

```
1 I := First();
2 while(I ≠ ⊥) {
3     Execute(I);
4     I := Next(I);
5 }
```

where Execute(I) is a macro in charge of sending the correct control signals to compute the iteration I of the tile loop. The functions First and Next are directly translated into VHDL if conditions. When these conditions are satisfied, the corresponding iterators are updated and the control signals are set.

The signal assignments in the FSM do not take into account the pipeline level at which the signals are connected. Therefore, we use additional registers to delay every control signal with respect to its pipeline depth. This ensures a correct execution without increasing the complexity of the state machine.

Parallelization

There are classical techniques of parallelizing the execution of a given tiling. They all basically consist in finding a parallel hyperplane which describes the tiles which have no interdependencies. Although this technique works for all examples, we believe that kernel-specific parallelizations can yield better performances, as in the case of the Jacobi kernel. In this direction, we propose to generalize the Jacobi parallelization to codes presenting dependence symmetries.

11.3 Computing kernel accuracy and performance

In this section we show, on our two working examples that the accelerator's implementation cost can be significantly reduced by designing operators which account for the application's accuracy requirements. In other words, given an average target relative error (which roughly gives average number of valid result bits) we give here an heuristic for choosing the intermediary floating-point formats based on a worst case error analysis. The validity of these heuristics is then tested on several examples.

11.3.1 Matrix-matrix multiplication

Let's consider the matrix-matrix multiplication $C \leftarrow AB$, where the elements of these matrices are floating-point numbers having w_E bits for representing the exponent and w_F bits for representing the fraction.

The standard iterative operator used in matrix-matrix multiplication performs $\sum_{k=0}^{N-1} a_{i,k}b_{k,j}$. For relatively small values of N this sum can be performed in parallel. For larger values of N an iterative operator $c_{i,j} \leftarrow c_{i,j} + a_{i,k}b_{k,j}, k \in 0..N-1$ is used.

The iterative operator implementation requires assembling one FP multiplier and one FP adder which serves as an accumulator. First, we consider that the elements of the input matrices A and B are exact and the instantiated FP operators employ the round-to-nearest rounding mode (the result of a calculation is rounded to the nearest floating-point number).

We denote by $fl(\cdot)$ the evaluation in floating-point arithmetic of an expression and we assume that the basic arithmetic operators $+, -, \cdot, /$ satisfy:

$$fl(x \text{ op } y) = (x \text{ op } y)(1+\delta), |\delta| \leq \texttt{ulp}/2$$

Table 11.1 *Minimum, average and maximum relative error out of a set of 4096 runs, for $N = 4096$, the elements of A and B are uniformly distributed on the positive/entire floating-point axis. The third architecture uses truncated multipliers having an error of 1* `ulp` *with* `ulp` $= 2^{-w_F-6}$. *Implementation results are given for a Virtex-4 speedgrade-3 FPGA device*

Architecture	Sign	Min	Average	Max	Performance
SP in/out,	+	1.55e-08 (2^{-25})	5.19e-05 (2^{-14})	1.06e-04 (2^{-13})	21 clk, 368MHz, 565 sl., 4 DSP
SP intern	±	3.00e-11 (2^{-34})	9.27e-06 (2^{-16})	1.68e-03 (2^{-9})	
SP in/out,	+	9.34e-10 (2^{-29})	4.72e-07 (2^{-21})	1.49e-06 (2^{-19})	32 clk, 308MHz, 1656 sl., 16 DSP
DP intern	±	3.00e-11 (2^{-34})	3.99e-06 (2^{-17})	8.42e-04 (2^{-10})	
SP in/out,	+	1.11e-10 (2^{-33})	5.29e-07 (2^{-20})	1.64e-06 (2^{-19})	22 clk, 334MHz, 952 sl., 1 DSP
$w_F + 6$ intern	±	3.02e-11 (2^{-34})	5.14e-06 (2^{-17})	1.29e-03 (2^{-9})	

In plain words we state that the maximum rounding error introduced by one of the above basic operations is bounded by $1/2$ `ulp` and is in average $1/4$ `ulp`.

During the iterative calculation of $c_{i,j}$ (a dot product between one vector of A and one of B) the rounding errors build-up at each iteration. Possible cancellations at each iteration prevent us from finding a practical static error bound in the general case. Therefore, we decide to provide an approximate static error bound, for each element of c by discarding the cancellation effects [91]. Let's consider as an example the dot product between two vector having two elements:

$$\begin{aligned}\widehat{p}_0 &= a_0 b_0(1+\delta_0)\\ \widehat{p}_1 &= a_1 b_0(1+\delta_1)\\ \widehat{s}_0 &= (\widehat{p}_0+\widehat{p}_1)(1+\delta_2)\\ &= a_0 b_0(1+\delta_0)(1+\delta_2) + a_0 b_0(1+\delta_1)(1+\delta_2)\end{aligned}$$

From here on we don't wish to distinguish between the δ_i so we use a notation due to Higham [91] which denotes products of the form $(1+\delta_i)...(1+\delta_{i+k-1})$ with $(1\pm\delta)^k$. Using this new notation, the error of the N-length dot-product kernel is:

$$\begin{aligned}\widehat{c}_N &= (\widehat{c}_{N-1} + a_{i,N-1}\, b_{N-1,j}(1\pm\delta))(1\pm\delta)\\ &= a_{i,0}\, b_{0,j}(1\pm\delta)^N + \sum_{k=1}^{N-1} a_{i,k}\, b_{k,j}(1\pm\delta)^{N+1-k}\end{aligned}$$

A simplified way to express this, due to Higham [91] is using the following notation:

$$\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1+\theta_n, \rho_i \in \{-1,1\}$$

where:

$$|\theta_n| \leq \frac{nu}{1-nu} = \gamma_n$$

The dot product can then be written as:

$$\widehat{c}_N = a_{i,0}\, b_{0,j}(1+\theta_N) + \sum_{k=1}^{N-1} a_{i,k}\, b_{k,j}(1+\theta_{N+1-k})$$

Table 11.2 *Minimum, average and maximum relative error for elements of an array in the Jacobi stencil code over a total set of 4096 runs, for $T = 1024$ iterations in the time direction. The numbers are uniformly distributed within wF exponent values. Implementation results are given for a Virtex-4 speedgrade-3 FPGA device*

Architecture	Min	Average	Max	Performance
SP	1.29e-11 (2^{-35})	2.56e-06 (2^{-18})	5.24e-04 (2^{-10})	32 clk, 395MHz, 954 slices
SP in/out, DP int.	1.90e-11 (2^{-38})	2.12e-08 (2^{-25})	5.83e-08 (2^{-24})	44 clk, 308MHz, 2280 slices
SP in/out, $w_F + 3$ int	1.78e-11 (2^{-35})	6.97e-08 (2^{-23})	4.53e-06 (2^{-17})	31 clk, 313MHz, 1716 slices

The error will exhibit the largest value when all sub-products have the same magnitude, and the rounding errors will all have the same sign. We will denote this bound by Δ. A well known rule of thumb [91] states that given an error bound Δ, the average error will roughly be $\sqrt{\Delta}$. The number of invalid bits due to roundings alone is bounded by $\log_2(\Delta)$ and is equal, on average to $\log_2(\sqrt{\Delta})$. This value was indeed validated experimentally as presented in Table 11.1. which reports the minimum, average and maximum relative errors for the vector product, the basic block in the matrix-multiplication algorithm. The input vectors have been populated using positive random numbers for one set of tests, and both positive and negative random numbers for the second set, uniformly distributed on the corresponding floating-point axis (uniformly distributed exponents).

The average relative error reported for a standard single-precision architecture using positive inputs (in order to avoid the effects of cancellation) is of the order 2^{-14}. The error bound obtained using equation 11.3.1 is about 4100 `ulp`. Using the previously mentioned rule of thumb, we expect that the average relative error in this case to be $\sqrt{4100} \approx 64.03$. Therefore the number of invalidated bits is equal to $\lceil \log_2(64.03) \rceil = 7$. Which gives an expected average relative error of 2^{-16} which is close to the 2^{-14} obtained experimentally.

The second architecture listed in table 11.1 processes the same SP input data using double-precision operators. The result is finally rounded back to single-precision. As expected, the accuracy of this architecture is improved, at a significant increase in operator size.

The third architecture processes the same SP input data using internal operators with a slightly larger precision ($wF + 6$ bits). Additionally, the floating-point multiplier is is implemented using truncated multipliers [40] (allow reducing the number of DSP blocks over classical implementations). Due to the extended fraction, the `ulp` value for this architecture is 2^{-29}. Accounting for the lower multiplier accuracy and the final conversion back to single precision, this architecture should still be roughly 2^6 times more accurate than the SP version. Indeed, experimental results presented in table 11.1 confirm that the average relative error for this implementation is of the order of 2^{-20}, 2^6 times smaller than the 2^{-14} for SP.

The second row for each architecture presents same relative error values when the input numbers are uniformly distributed on the entire floating-point axis (positive and negative) making cancellations possible. In average, each run had 7 cancellations. It can be observed that in such a situation, the three different architectures report similar numbers for the relative errors. Improving accuracy in such a case could be accomplished by avoiding cancellations as much as possible, allowing the computing unit to reorder the operations on the fly. Unfortunately, the proposed scheduling solution requires deterministic execution of operations which will not be the case in such an architecture.

11.3.2 One dimensional Jacobi stencil computation

The Jacobi stencil computation offers similar optimization opportunities. The main statement executes the averaging of three consecutive members of array a at time t to update the middle

index at time $t+1$.

We can model the impact of the rounding errors on this code using the arithmetic model previously introduced. Consider the assembly of standard floating-point operators.

$$\begin{aligned}\widehat{a}_{t+1,k} =&(((\widehat{a}_{t,k-1}+\widehat{a}_{t,k-1})(1+\delta_1)+\widehat{a}_{t,k+1})(1+\delta_2)\frac{1}{3})(1+\delta_3)\\ =&\frac{1}{3}\left(\widehat{a}_{t,k-1}(1+\theta_3)+\widehat{a}_{t,k}(1+\theta_3)+\widehat{a}_{t,k+1}(1+\theta_2)\right)\end{aligned}$$

The error bound after T steps is of the order θ_{3T}. In the case of an FPGA architecture, this error bound can be reduced to θ_{2T} by using a 3-input adder:

$$\begin{aligned}\widehat{a}_{t+1,k} =&((\widehat{a}_{t,k-1}+\widehat{a}_{t,k-1}+\widehat{a}_{t,k+1})(1+\delta_1)\times\frac{1}{3})(1+\delta_2)\\ =&\frac{1}{3}\left(\widehat{a}_{t,k-1}(1+\theta_2)+\widehat{a}_{t,k}(1+\theta_2)+\widehat{a}_{t,k+1}(1+\theta_2)\right)\end{aligned}$$

Using the same rule or thumb we estimate that the average error for a single-precision implementation with two floating-point adders and one constant multiplier will be $2^{-23+5}=2^{-18}$ ($\lceil\log_2(\sqrt{|\theta_{3T}|})\rceil=5$). This is indeed confirmed by the data presented in Table 11.2.

The our specific implementation (third line in table 11.2) uses a fused 3-input adder in order to enhance accuracy by saving one rounding error. Moreover, it uses an extended format of $wF+3$ bits. The average error in `ulps` one would expect from this implementation is $\lceil\log_2(\sqrt{|\theta_{2T}|})\rceil=4$ which invalidates 4 lower bits. Fortunately, the extended precision should absorb 3 of those, leaving the relative error of the order 2^{-22}. This is indeed confirmed by Table 11.2.

11.3.3 Lessons

The heuristic we propose is very simple, works for codes involving the basic operations: $+,-,\times,\div,\sqrt{x}$ working in floating-point arithmetic. The first task consists in defining the average accuracy requirement of the application (how many bits we expect, on average to be valid in our result), which we denote by γ. Why this average number of bits and not the worst case accuracy? Because in floating-point arithmetic, due to cancellations (subtraction of two very close values) errors can be amplified theoretically at every subtraction, possibly loosing all the result's accuracy.

Next, we express the accumulation of rounding errors (by discarding the possible amplifying effect of cancellations) using the model of floating-point arithmetic previously introduced (the interested reader should check the excellent book by Higham [91]). This gives us a worst case relative error (considering that no cancellations have amplified any error in the process) which we denote by Δ. We use the rule-of-thumb presented in [91]: the *average relative error* of the result is roughly equal to $\sqrt{\Delta}$. The average number of invalidated bits, due to this error is $\zeta=\lceil\log_2(\sqrt{\Delta})\rceil$. The working precision we chose for our circuit is therefore $\psi+\zeta$ in order to attain an average output accuracy of ψ.

11.4 Reality check

Table 11.3 presents synthesis results for both our running examples, using a large range of precisions, and two different FPGAs. The results presented confirm that precision selection plays an important role in determining the maximum number of operators to be packed on one FPGA.

Table 11.3 *Synthesis results for the full (including FSM) MMM and Jacobi1D codes. Results obtained using using Xilinx ISE 11.5 for Virtex5, and QuartusII 9.0 for StratixIII*

Application	FPGA	Precision (w_E, w_F)	Latency (cycles)	Frequency (MHz)	Resources REG	(A)LUT	DSPs
Matrix-Matrix Multiply N=128	Virtex5(-3)	(5,10)	11	277	320	526	1
		(8,23)	15	281	592	864	2
		(10,40)	14	175	978	2098	4
		(11,52)	15	150	1315	2122	8
		(15,64)	15	189	1634	4036	8
	StratixIII	(5,10)	12	276	399	549	2
		(9,36)	12	218	978	2098	4
Jacobi1D stencil N=1024 T=1024	Virtex5(-3)	(5,10)	98	255	770	1013	-
		(8,23)	98	250	1559	1833	-
		(15,64)	98	147	3669	4558	-
	StratixIII	(5,10)	98	284	1141	1058	-
		(9,36)	98	261	2883	2266	-
		(15,64)	98	199	4921	3978	-

Table 11.4 *Synthesis results for the parallelized MMM and Jacobi1D. Results obtained using using Quartus II 10.1 for StratixIII with* $w_E = 8, w_F = 23$

Application	Par. factor	Frequency (MHz)	Resources REG	(A)LUT	M9K	DSPs
Matrix-Matrix Multiply N=128	1	308	701	614	3	4
	2	282	1317	999	5	8
	4	303	2473	1789	12	16
	8	302	4842	3291	20	32
	16	281	9582	6291	32	64
Jacobi1D stencil N=1024 T=1024	1	311	1217	1199	9	-
	2	295	2394	2095	21	-
	4	283	4600	3853	38	-
	8	274	9018	7314	69	-
	16	251	17806	14218	132	-

As it can be remarked from the table, our automation approach is both flexible (several precisions) and portable (Virtex5 and StratixIII), while preserving good frequency characteristics.

The generated kernel performance for one computing kernel is: 0.4 GFLOPs for matrix-matrix multiplication, and 0.56 GFLOPs for Jacobi, for a 200 MHz clock frequency. Thanks to program restructuring and optimized scheduling in the generated FSM, the pipelined kernels are used with very high efficiency. Here, the efficiency can be defined as the percentage of useful (non-virtual) inputs fed to the pipelined operator. This can be expressed as the ratio $\#(\mathcal{I} \setminus \mathcal{V})/\#\mathcal{I}$, where $\mathcal{I}$ is the iteration domain and $\mathcal{V} \subseteq \mathcal{I}$ is the set of virtual iterations. The efficiency represents more than 99% for matrix-multiply, and more than 94% for Jacobi 1D. Taking into account the kernel size and operating frequencies, tens, even hundreds of pipelined operators can be packed per FPGA, resulting in significant potential speedups.

Table 11.4 presents synthesis results of the parallelization for both our running examples on the StratixIII FPGA using the single precision format. As expected, due to massive parallelism and no inter parallel process communication, for matrix multiplication example the scaling in terms of resources is proportional to the parallelization factor. The maximum operating frequency remains fairly constant. Jacobi 1D scales very well too. A small increase in utilized resources is due to the increase in the multiplexer size in order to fit signals from neighbor computational cores. The frequency remains fairly constant. This proves that our method is well suited for FPGA implementation.

There exists several manual approaches like the one described in [77] that presents a manually implemented acceleration of matrix-matrix multiplication on FPGAs. Unfortunately, the paper lacks of detailed experimental results, so we are unable to perform correct performance comparisons. Our approach is fully automated, and we can clearly point important performance optimization. To store intermediate results, there approach makes a systematic use of local SRAM memory, whereas we rely on pipeline registers to minimize the use of local SRAM memory. As concerns commercial HLS tools, the comparison is made difficult due to lack of clear documentation as well as software availability to academics.

11.5 Conclusion and future work

In this chapter we have presented a FPGA-specific approach to synthesizing programs described by perfectly nested loops with affine dependencies. The technique uses the information on the pipeline depth of the arithmetic operator implementing the inner statement in order to reschedule the program execution. Once a scheduling has been found, the arithmetic operator's architecture is attached with a specific interface consisting of multiplexers and shift-registers. This technique is FPGA-specific in the sense that the arithmetic operator with corresponding shift registers and multiplexers are generated on an application-basis.

The technique only writes data in the memory at tile boundary. In the context of multiple parallel processing elements, we have also presented an FPGA-specific technique that allows inter PE communication by simply connecting their interfaces. This technique discards the need for buffers in inter-process communication for our restricted class of applications.

We have also presented a heuristic method that given the average target accuracy for an application allows dimensioning the internal floating-point arithmetic data-path to obtain this accuracy. This technique can be easily automated and integrated in the same compiler tool. The savings in terms of resource usage implied by this technique are significant.

In the future, it would be interesting to extend our technique to non-perfect loop nests. This requires to consider each assignment as a process, the whole kernel being a network of communicating processes. Several model of process networks can be investigated, depending on the communication medium between processes (FIFOs or buffers).

Thanks

Most of the material presented in this chapter is based on a collaboration with the members of the COMPSYS team Christophe Alias and Alexandru Plesco under the supervision of my thesis advisor Florent de Dinechin. I would like to thank them all for their contributions.

CHAPTER 12
Using FloPoCo to solve Table Maker's Dilemma

The IEEE 754-2008 standard for Floating-Point arithmetic [17] suggests (yet does not dictate) that some elementary functions should be correctly rounded. That is, given a rounding function $\circ$ (e.g., round to nearest even, or round to $\pm\infty$), when evaluating function f at the floating-point number x, the system should always return $\circ(f(x))$.

One of the main objectives of the Arénaire project is building a fast mathematical library for these functions. Reaching this goal requires first solving a problem called the *Table Maker's Dilemma* for each target floating-point format and each elementary function. The problem requires massive amounts of computations which can be performed using computing environments and number formats substantially different from the target environment and floating-point format. In this chapter we propose a generic algorithm for this problem which maps well on modern FPGAs. A parametric description of the whole system architecture is designed entirely using the FloPoCo framework. Using FloPoCo to describe such a complex project has allowed us to validate the framework in new conditions, and to gain knowledge on how the framework can be improved.

12.1 The Table Maker's Dilemma

For the sake of simplicity, we assume from here on that the rounding function is round to nearest even.

On the target environment, to compute $f(x)$ in a given format, where x is a FP number, we must first compute an approximation to $f(x)$ with a given accuracy [123], which we round to the nearest FP number in the considered target format. The problem is the following: find what must the accuracy of the approximation be to make sure that the obtained result be always equal to the "exact" $f(x)$ rounded to the nearest FP number. To solve that problem we have to locate, for each considered target floating-point format and for each considered function f, the *hardest to round* (HR) points, that is the floating-point numbers x such that $f(x)$ is closest to the exact middle of two consecutive floating-point numbers (we call such a middle a midpoint), without being exactly a midpoint.

Assuming (after some re-normalization) that x and $f(x)$ are between 1 and 2, that the target floating-point format is a binary format of precision p, we need to find the largest possible value of m such that there exist a FP number x that satisfies:

– $f(x)$ is not exactly equal to a midpoint;

– the binary representation of $f(x)$ has the form

$$\overbrace{\underbrace{1.xxxxx\cdots xxx}_{p \text{ bits}}1000000\cdots000000}^{m \text{ bits}}xxx\cdots \quad \text{or} \quad \overbrace{\underbrace{1.xxxxx\cdots xxx}_{p \text{ bits}}0111111\cdots111111}^{m \text{ bits}}xxx\cdots;$$

Two different algorithms have been suggested for dealing with this problem:

1. **L-algorithm** first presented in Lefèvre's PhD dissertation [107, 108] allowed Lefèvre and Muller to publish the first tables of HR points for the most common functions in double-precision/binary64 FP arithmetic [109]
2. **SLZ algorithm** introduced by Stehlé, Lefèvre and Zimmermann [143, 144], was used to find the HR points for the exponential function in decimal64 arithmetic [110].

The SLZ-algorithm has a better asymptotic complexity than the L-algorithm. However, when the target format is the double precision/binary64 format, they require similar computational delays: weeks of computation for all input exponents (hours of computation for one input exponent), using massive parallelism. The problem with these algorithms does not only lie in this huge computation delay: it lies in the fact that a very complex algorithm, implemented in a very complex program, runs for weeks and just outputs one result: what confidence can we have in that result?

The HR points are one of the few weak parts of libraries such as CRLibm [59] where each function of that library comes with a theorem of the form "if the HR points have been rightly computed then the function always outputs a correctly rounded result". Hence, our major goal here is to design a very simple, very regular algorithm (therefore suited for FPGA implementation): if it outputs the same results as the L-algorithm, then this will give much confidence in these results.

The method we are going to suggest here has a worse asymptotic complexity than the L- and SLZ-algorithms. And yet, due to its simplicity, the hidden constant in the complexity term is so small that, still with double precision/binary64 as a target, our method will require similar delays on a single FPGA.

12.2 Proposed algorithm

Defining $u = 2^{1-p}$, we will compute the values of $f(1) \bmod 2^{1-p}$, $f(1+u) \bmod 2^{1-p}$, $f(1+2u) \bmod 2^{1-p}$, $f(1+3u) \bmod 2^{1-p}$, ..., $f(2) \bmod 2^{1-p}$, with a given, predetermined, accuracy $2^{-\mu}$ (with μ larger than p and—for probabilistic reasons [123]—less than $2p$).

Each time we find a value $f(1+ku) \bmod 2^{1-p}$ extremely close to 2^{-p} (i.e., whose leading bits are of the form $011111111\cdots$ or $10000000\cdots$), we output the value of k for some further testing. The major difficulty here is that since there are 2^{p-1} values $f(1+ku) \bmod 2^{1-p}$, and p is fairly large (a typical value is 53 for the double precision/binary64 format), the computation of these values must be done very quickly.

To do this, we will approximate f by some polynomial P (with an accuracy of approximation significantly better than $2^{-\mu}$), and compute the successive values $P(1+ku) \bmod 2^{1-p}$ using a modulo 2^{1-p} adaptation of the well-known *tabulated differences method* [96].

12.2.1 The tabulated differences method

Let P be a polynomial of degree n and x_0 a real. We define $x_k = x_0 + ku$ for $k > 0$, and we wish to compute the successive values $P(x_0)$, $P(x_1)$, $P(x_2)$, $P(x_3)$, ... The tabulated differences method is based on the fact that if we define the following "discrete derivatives":

– $P^{(1)}(x) = P(x+u) - P(x)$;

– $P^{(2)}(x) = P^{(1)}(x+u) - P^{(1)}(x)$;
– ...
– $P^{(n)}(x) = P^{(n-1)}(x+u) - P^{(n-1)}(x)$;

then $P^{(n)}$ is a constant C. This leads to the following algorithm.

Initialization compute $P(x_0)$, $P(x_1)$, $P(x_2)$, ..., $P(x_n)$, and deduce from these values all the possible partial discrete derivatives, of which we keep the initial vector at point x_0: $P^{(n)}(x_0) = C$, $P^{(n-1)}(x_1)$, $P^{(n-2)}(x_2)$, ..., $P^{(2)}(x_{n-2})$, $P^{(1)}(x_{n-1})$, $P(x_n)$.

Iteration The following recurrence computes the value of this vector at point $x_{k+1} = x_k + u$ out of the vector at point x_k.

$$\left\{\begin{array}{lcl} P^{(n-1)}(x_{k+2}) & = & P^{(n-1)}(x_{k+1}) + C \\ P^{(n-2)}(x_{k+3}) & = & P^{(n-2)}(x_{k+2}) + P^{(n-1)}(x_{k+2}) \\ \vdots & \vdots & \vdots \\ P^{(2)}(x_{k+n-1}) & = & P^{(2)}(x_{k+n-2}) + P^{(3)}(x_{k+n-2}) \\ P^{(1)}(x_{k+n}) & = & P^{(1)}(x_{k+n-1}) + P^{(2)}(x_{k+n-1}) \\ P(x_{k+n+1}) & = & P(x_{k+n}) + P^{(1)}(x_{k+n}) \end{array}\right. \tag{12.1}$$

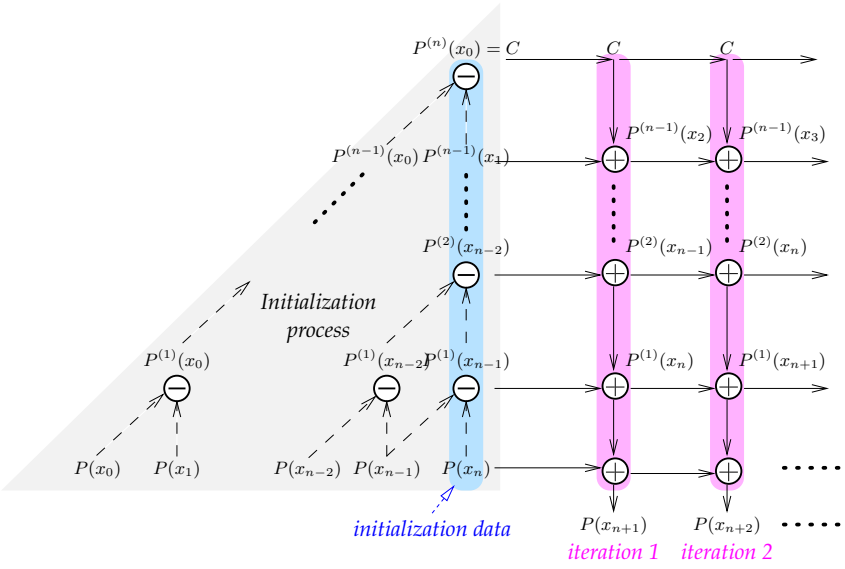


Figure 12.1 *The tabulated difference method*

The computations in (12.1) are done *modulo* 2^{1-p}: they are simple fixed-point additions, with the bits of weight $\geq 2^{1-p}$ being ignored. Notice that the n additions in (12.1) are straightforwardly pipelined, hence, once the initialization is done, *computing a new value $P(x_k)$ takes the time of one addition*. It should also be noted that the problem is embarrassingly parallel: Although the iteration itself is intrinsically sequential, the full domain of an elementary function in double-precision may be split into arbitrarily many sub-domains, and we may perform initialization/iteration processes in parallel for each sub-domain. We indeed aim at processing hundreds of sub-intervals in parallel within a single FPGA.

The initial values $P(x_0)$, $P(x_1)$, $P(x_2)$, ..., $P(x_n)$ cannot be computed exactly in practice. They are correct within some rounding error, and we will see in the following that when performing

(12.1), these errors accumulate quite quickly. After some value of k, say $k_{\max}$ this accumulated error becomes unacceptable, and we have to invoke the initialization process again, with x_0 replaced by $x_0 + k_{\max}u$. In other words, the size of a sub-interval is dictated by an error analysis that will be the subject of Section 12.2.2.

The initialization process requires $n+1$ polynomial evaluations, and $n(n+1)/2$ subtractions. There are two possible ways of performing it:

1. on the FPGA itself, or
2. in software on the host computer.

In any case, we chose $k_{\max}$ so that the initialization time is totally overlapped by the iterations (12.1).

The initialization process first involves evaluating the polynomial in $n+1$ points using a classical multiplication-based scheme (typically Horner's). Modern FPGAs contain up to several thousand small multipliers that could be used for this purpose, but designing an architecture for this initialization would add a lot to the FPGA design effort. In the sequel of this chapter, we therefore choose the simpler second approach. It also has the advantage of exploiting the computing power of the host processor. However, there is a price to pay: due to limited bandwidth between the host and the FPGA, iteration (12.1) must run for a much longer $k_{\max}$. We will see in next section that this entails a significantly wider data-path, hence more resource consumption for the iteration hardware, possibly cancelling the benefits of saving the initialization hardware. This question remains to study quantitatively.

Let us now formalize the dependency between $k_{\max}$ and the datapath width.

12.2.2 Error analysis

Let us bound the difference between the value computed $F(x_k)$ and the true value of the function $f(x_k)$. We may first decompose this error as follows:

$$F(x_k) - f(x_k) \;=\; (F(x_k) - P(x_k)) + (P(x_k) - f(x_k))$$

The second term is the approximation error, and the Remez approximation algorithm will allow us to keep it as small as needed. Let us focus on the first term, the rounding error.

Let $\delta(i)(x_0 + ku)$ be the overall rounding error in the initial evaluation of $P^{(i)}(x_0 + ku)$. Notice that the additions in (12.1) are performed in fixed-point, ignoring the outgoing carries: they do not induce any error, yet they propagate the initial rounding errors on the $P^{(i)}(x_0)$. Let ϵ be a bound on the errors on $P^{(n)}(x_0) = C$, $P^{(n-1)}(x_0 + u)$, $P^{(n-2)}(x_0 + 2u)$, ..., $P^{(1)}(x_0 + (n-1)u)$. We easily find

$$\delta^{(n-1)}(x_0 + ku) \leq \delta^{(n-1)}(x_0 + (k-1)u) + \epsilon,$$

so that

$$\delta^{(n-1)}(x_0 + ku) \leq (k+1)\epsilon.$$

From that, we deduce

$$\begin{array}{rcl} \delta^{(n-2)}(x_0 + ku) & \leq & \delta^{(n-2)}(x_0 + (k-1)u) + \delta^{(n-1)}(x_0 + (k-1)u) \\ & \leq & \delta^{(n-2)}(x_0 + (k-1)u) + k\epsilon, \end{array}$$

so that

$$\delta^{(n-2)}(x_0 + ku) \leq (1 + 2 + 3 + \cdots + k)\epsilon = \frac{k(k+1)}{2}\epsilon.$$

Similarly,

$$\delta^{(n-3)}(x_0 + ku) \leq \left(1 + \tfrac{2(2+1)}{2} + \tfrac{3(3+1)}{2} + \cdots + \tfrac{(k-1)k}{2}\right)\epsilon = \tfrac{(k-1)(k)(k+1)}{6}\epsilon.$$

An elementary induction shows that the bound on the error of the computed value of $P(x_0 + ku)$ satisfies

$$\delta^{(0)}(x_0 + ku) \leq \frac{(k-n+2)\cdots(k-1)(k)(k+1)}{n!}\epsilon. \tag{12.2}$$

12.2.3 An example: the exponential function

All parameters in the method are function-dependent, so we cannot give a general performance result (although it should not vary much with the function). Hence, we give here some figures related to the exponential function on the input interval $[1, 2)$. We take $f(x) = \frac{1}{2}e^x$ to normalize the output to $[1, 2)$.

We first split the input interval into 2^m sub-intervals, each of size 2^{-m}, and we will compute one approximation polynomial on each sub interval, using the Remez algorithm. The trade-off here is between the degree of the obtained polynomials (the smaller, the better for the subsequent evaluation) and the number of Remez polynomial to compute. A good choice here is $m = 15$: on each of the $2^{15} = 32768$ sub-intervals, a polynomial of degree 4 approximates $f(x)$ with an accuracy better than 2^{-90}, and the Sollya tool is able to compute all these polynomials and formally validate their accuracy [53] in about 3 hours.

Now we must choose $k_{\max}$, which dictates the length of an evaluation run between reinitializations. Having decided that reinitializations are performed on the host processor, we now have to take into account the limits on 1/ computing power of the host processor and 2/ data bandwidth between processor and FPGA. A larger $k_{\max}$ means fewer initializations, but larger data-path, hence slower operation and less parallelism. Note also that if we have P parallel iteration cores, the host must serve them all.

Our current trade-off is to take $k_{\max} = 2^{20}$. Equation (12.2) with degree $n = 4$ tells us that the error is smaller than $2^{76}\epsilon$. For our target accuracy of 2^{-85} modulo 2^{52}, we need to have $85-52 = 33$ valid bits at the end of the computation. A datapath width of $33 + 76 = 109$ bits ensures this accuracy. Assuming $M = 2^8$ parallel iterations on the FPGA, the host must be able to compute one initialization every $2^{20}/2^8 = 4096$ FPGA cycles. FPGA cycles are typically 10 times slower than processor cycles, so the host has roughly 40,000 cycles to compute each initialization. Efficient multiple-precision libraries such as GMP and MPFR make this possible. Host-FPGA bandwidth, in this scenario, is not a problem.

12.3 Our design

A natural technology for implementing this type of algorithm is the FPGA. There are several reasons for that:

- for a given set of input parameters we need to perform a big, one-off computation. Once completed we can reconfigure the FPGA for a different set of input parameters.
- the implemented method is based on binary additions for which FPGAs are very well optimized to perform.

Nevertheless, manually implementing such a complex, multi-parametrized design using a hardware description language (HDL) is a tedious and error-prone task. We have decided to use the FloPoCo framework for the parametric architectural description for two main reasons:

- the full architecture should by fully parametric in order to easily explore different trade-offs. Due to the high-abstraction level provided by C++, the numerous parameters: polynomial degree, data-path bit-widths, FIFO sizes, number of processing elements etc. are more easily managed. The generated VHDL has no parameters or `FOR... GENERATE` constructs which makes it easier to verify.

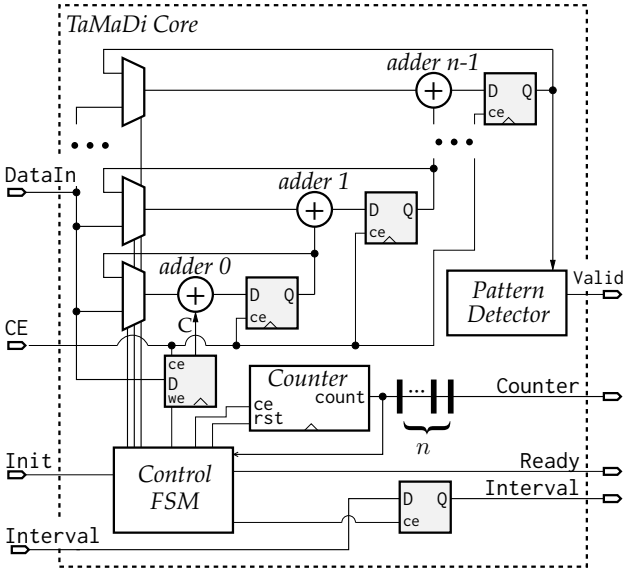


Figure 12.2 *Polynomial Evaluator based on the tabulated differences method*

– the implementation of such a complex project in FloPoCo will provides us with precious feedback on how to improve the framework.

12.3.1 Functional model

TaMaDi Core

The core component of our design is the polynomial evaluator based on the tabulated differences method. The architecture of this component is depicted in Figure 12.2. Its main entities are the $n-1$ adders chained together which are used to evaluate the vector of discrete derivatives.

Each computation starts with the component receiving a '1' value on the `Initialize` line together with a unique interval identifier on the `Interval` bus. During the next $N+1$ clock cycles, the values of the initialization vector $C = P^{(n)}(x_0)$, ..., $P(x_0 + nu)$ are received in sequence on the `DataIn` bus, and fill the pipeline. A counter is used to keep track of k in evaluating $P(x_0 + ku)$. The output of the n^{th} adder feeds a pattern detector unit, implemented as wide `AND`. A value of '1' at the output of the pattern detector signals that the value present on the output `Counter` bus, together with the interval identifier, points to one HR case. The component raises the `Ready` line to '1' when it has finished the allowed number of iterations and needs a new reinitialization.

The architecture of the TaMaDi Core (Core) is perfectly suited to FPGA hardware. Adders benefit from the fast carry-chains which allow the simple ripple carry adder (RCA) scheme to be implemented efficiently. The pattern detector may also take advantage of these fast carry-chains in Xilinx devices. On Altera devices, the 6-input ALUT feature is used to implement this using multi-level logic. The inter-LAB direct connections allow fast frequencies. In any case, the component can be pipelined as it is outisde the loop's critical path.

Table 12.1 presents area and timing post place-and-route results of the TaMaDi Core on modern FPGAs from Xilinx [23, 20] and Altera [27, 28] for the exponential function example presented in Section 12.2.3. The area of one Core occupies a very small fraction of these FPGA. The largest StratixV from Altera(5SGXAB) having 1052K LUTs and 1588K REGs can, in theory accommodate over 1500 Cores while the largest Virtex6 from Xilinx(XC6VLX760) having 758K LUTs and 1516K REGs can accommodate roughly 1000 Cores, if one also considers the interfaces overhead.

Table 12.1 *Post place-and-route results of the TaMaDi Core PE*

Datapath width	Degree N	FPGA	Frequency	Area LUTs	Area REGs
120	4	StratixIV	237 MHz	584	750
		StratixV	359 MHz	585	750
		Virtex5	262 MHz	640	646
		Virtex6	332 MHz	640	646

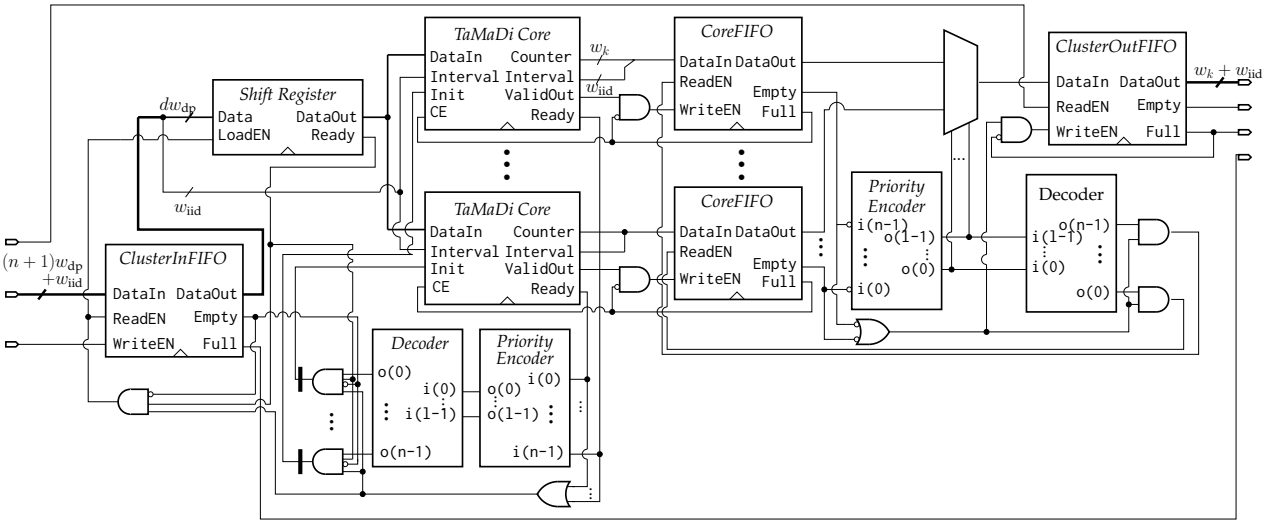


Figure 12.3 *Overview of the TaMaDi Cluster architecture*

TaMaDi Cluster

Multiple TaMaDi Cores may be assembled in a larger component named TaMaDi Cluster, whose architecture is depicted in Figure 12.3.

The presented system has several parameters:

- M the number of TaMaDi Cores in the system. The maximum value of this parameter depends on the size of one Core and the size of the FPGA. The practical value for this parameter also depends on the bandwidth between FPGA and the host system, Core datapath width, degree and reinitialization interval.
- size of the input and output FIFOs, also depends on bandwidth, M and Core characteristics.
- size of CoreFIFO. Their dimension can be as small as one element. However, for good performance their size should be dimensioned according to the probability of finding HR cases in that interval and the output bandwidth.

The TaMaDi Cluster is connected to the host system (or the next hierarchical level) by means of two FIFOs. Data is fed by the host system to ClusterInFIFO while this FIFO is not full. Each element on this FIFO has the structure depicted in Figure 12.4.

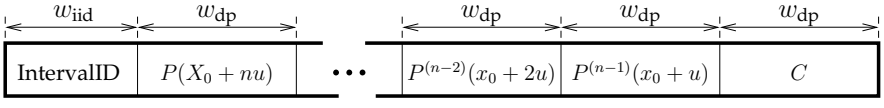


Figure 12.4 *Structure of one element in the ClusterInFIFO*

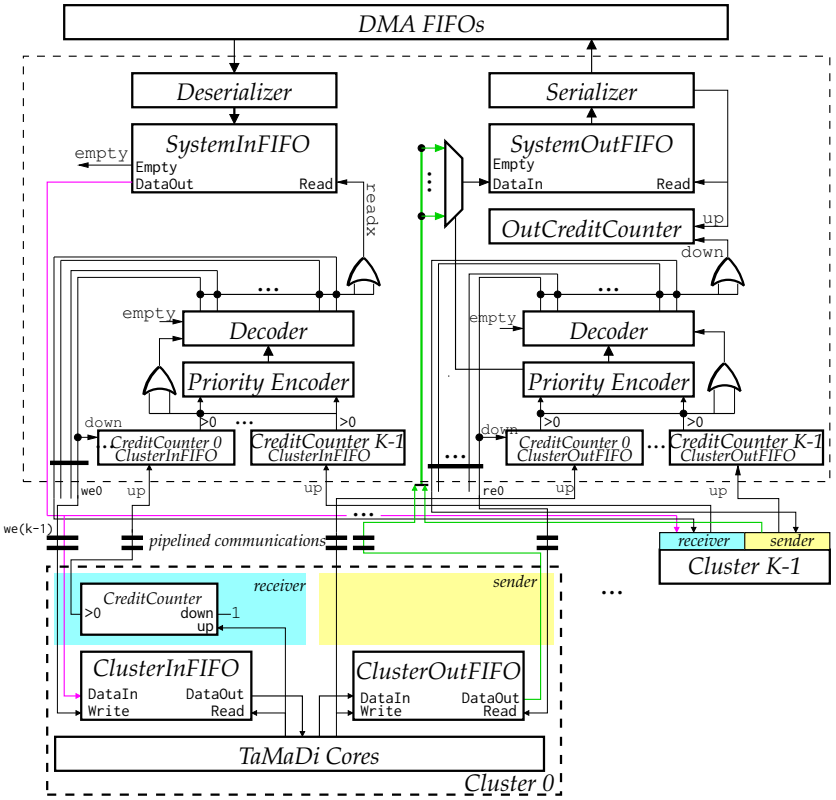


Figure 12.5 *Global system dispatcher interface*

The ClusterInFIFO element contains the necessary information to bootstrap one processing element. Once a TaMaDi Core is ready to process new information (signaled '1' on the corresponding output `Ready` port) the input FIFO is popped one element. The uppermost w_{iid} bits of information containing the interval ID are fed to the processing element together with a value on '1' on the corresponding `Initialize` input pin. The lowermost $(N+1)w_{\text{dp}}$ bits are loaded into a $N+1$-level shift-register in order be serialized in chunks of w_{dp} bits. During the next $N+1$ clock cycles, the shift-register feeds the TaMaDi Core as the pipeline starts.

When the TaMaDi Cores signals the detection of a HR case, the information concerning this case (counter value and interval identifier, totaling $w_k + w_{\text{iid}}$ bits) is pushed into the corresponding Core output FIFO.

The data from the CoreFIFOs is then placed in the ClusterOutFIFO whenever this FIFO is not full. Simple priority encoders on both inputs and outputs manage the access to the Cluster input and output FIFOs.

TaMaDi System

The TaMaDi Cluster has low resource count and fast clock speeds for modest number of Cores (up to 32). However, this multiplexed data dispatch architecture scales badly due to several issues: (1) size of the multiplexers and priority encoder-decoder circuitry (2) the long lines between the dispatcher (shift-register in our implementation) and the computing cores (TaMaDi Cores).

By grouping multiple TaMaDi Cores into a small number of Clusters (where the size of the Cluster remains reasonably small say 16), we could potentially use the same dispatching architecture at a macro-level.

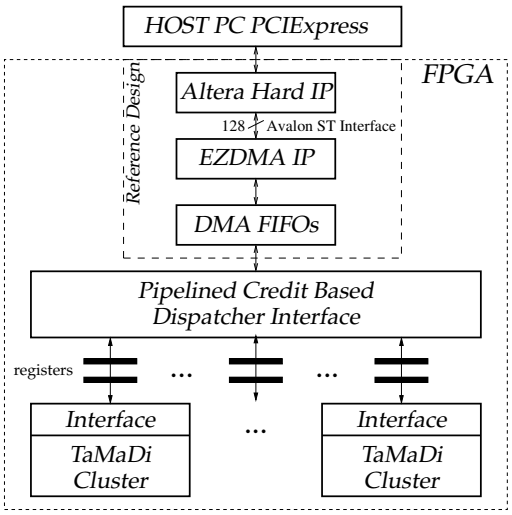


Figure 12.6 *Global system architecture*

However, when filling up a large FPGA chip, a new problem arises: connections between the dispatcher and the Clusters become very long, introducing large delays. To overcome this, we have used a different, credit-based, dispatcher, depicted in Figure 12.5. It allows us to pipeline the communication lines to Clusters, thus breaking the long delays into several shorter ones.

A detailed view of this sender/receiver interface is presented on the bottom of Figure 12.5. The receiver part of the interface is tightly coupled to the TaMaDi Cluster. It consists of a counter, initialized to the size of the Cluster Input FIFO. The value of this counter represents the number of elements that the Cluster Input FIFO can receive before being full. At each clock cycle, the counter will send one credit to the dispatcher's sender interface and will block when no credits are available.

This credits are counted in the dispatcher's sender interface. Whenever this counter has credits (value > 0), it asserts a `ready` signal. This signal indicates that the cluster is ready to receive an initialization vector. A priority encoder/decoder circuit is then used to select the Cluster to which the initialization vector will be sent to.

The `initializationData` and `write-enable` signals are sent synchronously through the pipelined connection to the input FIFO of the TaMaDi Cluster. When this data will be read from this FIFO, the credit will be incremented in the Cluster's receiver interface.

The protocol passes credits between the interfaces in a circular manner. The maximal number of credits is equal to the size of the TaMaDi Cluster Input FIFO. In the worse case there will be no credits in any of the credit counters, no data in the pipeline connection, thus all the credits are transformed into data elements inside the input FIFO.

A similar system is used for the Cluster sender/ dispatcher retrieval part. In addition, an System Output Credit Counter is used to to keep track of the number of empty elements of the System Output FIFO. This counter is decremented whenever a read signal is sent to the TaMaDi Cluster and is incremented whenever the System Output FIFO is popped one element.

Full Prototype

For prototyping purposes we use the Stratix IV GX development board featuring a Stratix IV GX EP4SGX530KH40C2 FPGA. The board communicates with a host PC by means of a PCI Express 2.0 8x interface that can provide up to 3.4 GB/s full-duplex.

The Altera PCI Express hard IP together with the PLDA EZDMA2 IP [5] offered in the PLDA reference design ensure a simple FIFO interface for our pipelined credit-based Dispatcher Interface (Figure 12.6). The PLDA host driver offers a high-level API interface for feeding and retrieval of information from the DMA FIFOs by means of multiple DMA channels (2 in our case).

12.3.2 Bandwidth requirement

In this section we will compute the bandwidth requirement of the entire TaMaDi System. First, we need to compute the bandwidth of TaMaDi Cluster depicted in Figure 12.3.

We use the following notations:
f circuit frequency
K the number of TaMaDi Clusters
M number of TaMaDi Cores within a Cluster
N approximation polynomial degree
w_{dp} the datapath width of the TaMaDi Core
$k_{\max}$ the number of iterations between re-initializations
$w_k = \lceil \log_2(k_{\max}) \rceil$, width in bits of the iteration counters.
η the maximum number of intervals to be processed by the system
$w_{\text{iid}} = \lceil \log_2(\eta) \rceil$, width in bits of interval identifiers
ξ the probability of finding one HR case
The input bandwidth for one TaMaDi Core:

$$B_{\text{Core}}^{\text{in}} = \left((N+1)w_{\text{dp}} + w_{\text{iid}}\right) \frac{f}{k_{\max} + N + 1} \tag{12.3}$$

and the output bandwidth of one Core is:

$$B_{\text{Core}}^{\text{out}} = \xi(\lceil log_2(k_{\max}) \rceil + w_{\text{iid}}) \frac{f}{k_{\max} + N + 1} \tag{12.4}$$

The Core bandwidth is $B_{\text{Core}} = B_{\text{Core}}^{\text{in}} + B_{\text{Core}}^{\text{out}}$. Considering that a TaMaDi Cluster has M such Cores, the total bandwidth requirement for a Cluster is $B_{\text{Cluster}} = M \cdot B_{\text{Core}}$. A TaMaDi System is composed out of K Clusters, therefore requiring a bandwidth equal to: $B_{\text{System}} = K \cdot B_{\text{Cluster}} = K \cdot M \cdot B_{\text{Core}}$.

Table 12.2 presents the dependency between the parameters of a Core, its area and the required bandwidth for keeping it busy at 100MHz. A larger bandwidth requirement leads to more pressure on the I/Os but a smaller Core size, which allows fitting more in one single FPGA.

For a system comprising of 100 TaMaDi Cores, each requiring a bandwidth of 5.42 MBit/s the bandwidth requirement is approximatively 0.5 Gb/s which seems to be reasonably within our available bandwidth potential. Nevertheless, this configuration would require us generating more than 57 TBytes of reinitialization data (some of which can indeed be generated on-the-fly) compared to a more manageable 1.9 TBytes required for a $w_{\text{dp}} = 120$.

For a 200-Core system $B_{System} = 3.4$ Mbits/s, which can easily be provided by FPGA platforms connected through to the host system through the PCIE bus, Ethernet interface and even USB2.0.

12.3.3 Performance estimation

We are currently using the Altera StratixIV development kit based on an EP4SGX530KH40C2 FPGA to prototype our system. This gives us an environment for estimating the performance and scalability of our architecture. However, as presented in this section the amount of computation

Table 12.2 *Dependency between TaMaDi Core parameters, its area and the necessary bandwidth/-Core for a StratixIV. Similar results hold for other FPGAs*

Parameters				Area		Bandwidth
N	w_{dp}	$k_{\max}$	w_{iid}	LUTs	REGs	(Mbit/s)
4	120	2^{20}	32	584	750	0.017
4	81	8,192	39	400	531	5.42

needed for an elementary function, on one exponent value is of the order of tens of hours. We therefore envision mapping this architecture on even larger FPGAs, and even multi-FPGA based systems.

In this section we provide a performance estimation for the case of the exponential function for double precision $p = 52$ (we consider one input exponent). Considering the 2^{20} iterations until having to reinitialize the Core, the total number of intervals to process is about $4.3 \cdot 10^9$.

Table 12.3 shows the dependency between system frequency, number of Cores and task completion.

On our current FPGA prototyping system we conservatively estimate to be able to pack 200 PE which yields a realistic execution time of approximatively 50 hours.

12.3.4 Reality Check

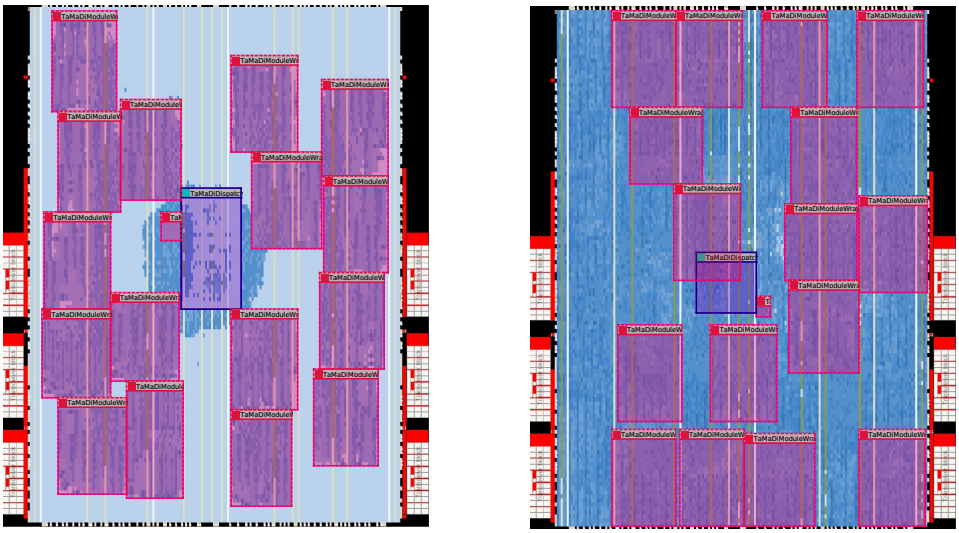
We have tested the real performance of different configurations of our proposed systems on the prototyping StratixIV development-kit. The purpose of these tests was to show the *performance* of our solutions and to determine the degree of *scalability* of the proposed architectures (both the simple-dispatcher and the credit-based dispatcher solution). The results are shown in Table 12.4

First, the results obtained validate that, once the number of Cores exceeds a certain threshold (32 for StratixIV), the credit-based dispatch offers a more attractive solution. As the number of Cores scales up, the credit-based dispatch will continue to function at frequencies over 150 MHz, as the critical path of this system is in the adders of the TaMaDi Core. Nevertheless, due to chip congestion, once the global resource utilization is above 80% frequencies are expected to drop as well for this architecture. In our experiments we have created logical regions for helping the place-and-route tool better place the cluster modules of the credit-based dispatch archtitecture. The logical regions were restricted to one cluster, with separate regions for the dispatcher and deserializer units. Moreover, we have fixed the placement of the dispatcher region in the central area of the FPGA, in order to minimize wire length to the Clusters. Figure 12.7(a) presents the placement of the 128 cores using logical regions for all clusters. As it can also be observed from Table 12.4 the logical regions fit nicely, as only half the FPGA resources are occupied.

When trying to place 256 cores using the same methodology, we were unsuccessful. The project itself had no problem fitting (only 82% of the resources were used) if no logical regions are used. As the size of the logical region was determined by the number of M9K memories used

Table 12.3 *Performance estimates for double-precision exponential (one input exponent)*

		Frequency		
		100	150	200
Cores	100	125h	83.4h	62.5h
	200	62.5h	41.7h	31.3h
	400	31.2h	20.9h	15.7h
	800	15.7h	10.4h	7.9h



(a) 128 cores (b) 256 cores

Figure 12.7 *Placement of the synthesized the TaMaDi System using logical regions*

to implement the FIFOs of one Cluster, some of logic and registers in that region was underutilized. Our strategy in this case was to assign logic regions to the dispatcher and 16 out of the 32 clusters, and leave the synthesizer pack the rest of the 16 clusters in the remaining area. Figure 12.7(b) presents the placement of the 256 cores using this strategy. This has allowed us to obtained better frequencies for this number of cores than by not using logical regions.

However, when reading this table one should consider that, as the size of the FPGA increases linearly, the time needed to compile the project on the FPGA increases at best polynomially. In other words, if for the 16 cores StratixIV design, compilation took some tens of minutes on a fast server, StratixV designs took tens of hours to compile.

A solution to improve the compile/execution time ratio is to use a multi-FPGA based system, comprising of multiple similar FPGAs, such as the multi-FPGA prototyping board DN7020K10 from Dini Group [7], comprising 16 Altera StratixIV FPGAs. The TaMaDi System would be compiled once, then replicated on these FPGAs. For simplicity one FPGA will also contain a dispatcher interface and will be connected to the host system. We estimate that one such system would complete the execution of one exponent in less than 2 hours.

All in all, depending on the available FPGA, the order of magnitude of the time required to process one input exponent is between a few hours and two days. Although the double precision/binary64 format has 2046 possible exponents, we do not need to perform such a calculation for every exponent: the exponential of a number larger than 710 is an overflow, and if $|u| \leq 2^{-54}$, then e^u correctly rounded to that format is equal to 1. The most up-to-date implementation of the L-algorithm takes 45 hours to process one input exponent on a fairly recent FP core (AMD Opteron 2.19 GHz). Since these algorithm are *very* different and are run on very different machines, we suggest using both of them, which allows one to get much confidence in the obtained results.

12.3.5 FloPoCo impact

The work we performed on this project has confirmed that FloPoCo can be used with success to design entire computational systems. It did however suggest that some improvements can be made in order to further increase design productivity:

– counters are a basic design block in computing systems but don't entirely fit the definition

Table 12.4 *Post place-and-route results of the TaMaDi System. The Core parameters are:* $w_{dp} = 120$ *bits and* $N = 4$

FPGA	Cores	Freq.	Area			Completion
			LUTs	REGs	M9K	Time
StratixIV (EP4SGX530KH40C2) *simple-dispatch*	16	196 MHz	10,614 (2%)	14,725 (3%)	17	398.9h
	32	174 MHz	20,021 (4%)	26,250 (6%)	17	224.7h
	64	154 MHz	48,416 (11%)	61,234 (14%)	148	126.9h
	128	111 MHz	95,428 (22%)	118,944 (28%)	276	88.05h
	256	97 MHz	189,298 (45%)	234,432 (55%)	532	50.4h
StratixIV (EP4SGX530KH40C2) *credit-based dispatch*	16 (2x8)	198 MHz	13,159 (3%)	21,586 (5%)	53	394.9h
	32 (4x8)	193 MHz	25,014 (6%)	39,402 (9%)	87	202.6h
	64 (8x8)	168 MHz	59,213 (14%)	89,051 (21%)	308	116.4h
	128 (16x8)	168 MHz	96,534 (22%)	156,370 (36%)	592	58.2h
	256 (32x8)	127 MHz	232,649 (54%)	348,335 (82%)	1172	38.5h

of operators FloPoCo was designed to support. Providing these counters as basic FloPoCo primitives would reduce design time. The main difficulty does not lie in the feedback loop, but rather on the `chip enable` and reset signals. By default, pipelined FloPoCo operators are connected to a global clock `clk` and a global reset `rst`. This is however insufficient for more complex situations: the interval counter of the TaMaDi Core needs to be reseted each time `Init='1'`. A specific reset signal needs to be mapped to this component.

– by default, FloPoCo operators are meant to function in a pipelined fashion, at each clock cycle. However, there are situations when we would desire stopping the execution of a component for a certain time: if the TaMaDi Core's corresponding CoreFIFO is `full`, we need to stop the execution of the core until the CoreFIFO has at least one free element. Otherwise, some HR cases could be lost. Adding optional `CE` signals to generated components is a necessity if we desire using FloPoCo in such context.
– in this project we saw that using logic-regions, in combination with design partitions significantly improves compilation and place-and-route times. Doing it by hand using the vendor tools is a possibility, however, FloPoCo could allow exporting the partition information alongside with `vhdl` file. We are considering adding this feature to all FloPoCo operators. In the case of TaMaDi Clusters we know that the shape of this region will be influenced by the number of M9K memories available in that region for implementing the FIFOs. In the case of more complex operators, making use of logic, DSP blocks and BRAMs, the decision might not be that simple. We are currently investigating these possibilities.

Some of the components used in this project, such as the FIFOs, priority encoder/decoder, and serializer/deserializer units are also available as stand-alone FloPoCO operators.

12.4 Conclusion

We have suggested an algorithm and an FPGA architecture that make it possible to find hardest-to-round points for elementary functions in double precision. This requires huge computations, but they are done once for all, and allow one to design efficient libraries or hardware for elementary function evaluation. The achieved performance is slightly better than the one obtained using Lefèvre's L-algorithm but the real gain is not there: it lies in the fact that if, with a completely different method that runs on a completely different hardware, we obtain the same results, this gives much confidence in these results.

Thanks

I would kindly like to thank Jean-Michel Muller for bringing this computational intensive task to us. I would also like to thank Alexandru Plesco for his contributions to the design and optimization the FPGA architecture. Also, I gratefully acknowledge the donation of a DK-DEV-4GX530N board by the Altera University which will be used as our production board.

13 Chapter 13
Conclusions and Perspectives

The increasing capacities and new added features, like embedded multipliers, have made FPGA devices attractive for accelerating applications. A large class of targeted applications make extensive use of floating-point arithmetic. Their acceleration is directly dependent on the availability of high-performance floating-point operators which are hard (and sometimes impossible) to design by hand using standard HDLs. In this thesis we have proposed the FloPoCo framework for the development of arithmetic operators.

The FloPoCo framework brings several features to help the development of operators of different granularities which make it the most advanced tool for building flexible arithmetic pipelines for FPGA.

- Firstly, FloPoCo provides a vast library of highly-efficient operators: fixed-point including adders (multi-operand as well), multipliers (regular, truncated and constant), shifters, leading-zero and leading-one counters, and two generic function evaluators *HOTBM* [66] and *FunctionEvaluator* (Chapter 7) and many more; floating point operators including the square-root, exponential, logarithm, power; the *FPPipeline* meta-operator allowing fast floating point pipeline assembly. These operators can be optimized for several target FPGAs.
- Secondly, it provides a pipelining infrastructure which decouples the task of describing a combinatorial operator and the task of pipelining it (the pipelines are correct by construction). Pipelining is frequency-driven and uses abstract FPGA models to capture the device's features and routing information. Frequency-driven pipelining based on these models minimizes resource consumption and latency. Moreover, operator design and pipelining based on abstract FPGA models ensures that FloPoCo operators are future-proof. Porting all operators to a new FPGA, which is not substantially different from our currently supported FPGAs, should reduce to adding a new target class to the FloPoCo hierarchy.
- Finally, FloPoCo offers a built-in test-bench generation suite which allows testing the designed operators against their mathematical specification using specialized mathematical libraries. This suite can and will be extended to support automatically testing entire pipelines built by operator assembly.

FloPoCo is a relatively young project (3 and a half years). It is used in several academic and industrial projects and has received several external contributions.

- The PandA Project [1] from Politecnico di Milano uses FloPoCo as a back-end for core generation. They also contribute to the project and maintain the automake tools.
- The PivPav project from University of Paderborn is an open-source circuit library with benchmarking facilities which offers both FloPoCo and CoreGen alternatives to their op-

1. http://trac.elet.polimi.it/panda/

erator generator backend [87].
- The Greco project from Universidade Federal de Pernambuco uses FloPoCo for designing DSP architectures (Fast Fourier Transform) [136].
- FloPoCo is also used at Imperial College, London [118, 135].
- The ADACSYS (Advanced Acceleration Systems) startup is currently using FloPoCo as their core generator. They have provided us use of their hardware testing infrastructure for on-chip operator testing.
- The Prüftechnik Group (Industrial maintenance and quality control) also uses FloPoCo. They have also contributed to the framework with the description of an Altera CycloneII target FPGA.
- NASA evaluated FloPoCo in their project *Low Power Supercomputing in Space* [141].
- The pipelined adder architectures of FloPoCo are used in the Computer Arithmetic curricula at George Manson University (http://ece.gmu.edu/coursewebpages/ECE/ECE645/S11/)

Other academics using FloPoCo include: University of Cape Town, South Africa, U.T. Cluj-Napoca, T.U. Hamburg, University of Essex, U. Madrid, T. U. Muenchen, T. U. Kaiserslautern, CalTech, U. Perpignan, U. Tokyo, Virginia Tech U.

When this project started, our main goal was to have in FloPoCo an efficient and flexible floating-point mathematical library. In this process we had to spend a significant amount of effort optimizing the most often encountered subcomponents: adders and multipliers.

- Therefore, we have proposed several pipelined adder architectures which allow fine-grain integration in the sub-cycle accurate FloPoCo framework. We have also presented an improved family of short-latency architectures based on the carry-select architecture, which take full advantage of the fast-carry chains of modern FPGAs. Addition is a pervasive operation which makes the design of these basic blocks of primal importance for the FloPoCo project.
- The multipliers families presented here exploit the flexibility of the FPGA target and they are original in that respect. The Karatsuba-like multipliers significantly reduce DSP usage both for FPGAs with square multipliers, but also on FPGAs with rectangular ones for larger input width. The tiling-based multiplier family takes the best advantages of rectangular multipliers and offers performant multipliers up to double-precision. The tiling-based truncated multipliers are a precious resource for implementing high-performance polynomial evaluators. All these multiplier architectures are essential building blocks for coarser FloPoCo operators.

In our way to a complete floating-point libm, we also had to build a generic tool for fixed-point function evaluation. Our implementation, *FunctionEvaluator*, scales beyond the precisions of existing works and provides significant better performance compared to the literature. On the one hand, this tool provides FPGA-specific flexible implementations of fixed-point function, often needed digital signal processing. On the other hand, it provides an effective implementation of fixed-point functions needed for implementing the components of our objective floating-point libm.

The first concrete member of the FloPoCo libm is a floating-point square root operator based on the previous FunctionEvaluator. This operator is smaller and more performant than what the literature offers. A second member of this library is the presented floating point exponential operator. It produces last-bit accurate architectures, is fully parameterizable and is optimized for a wider range of FPGA targets, range of precisions, latency/frequency trade-offs. The operator makes good use of the DSP blocks and embedded memories of high-end FPGAs, and significantly outperforms previous works in performance and resources consumption.

We hope that other operators of the same quality will soon be part of the FloPoCo libm. Work is currently on the way to implement the powering function, and the next step will be to imple-

ment the trigonometric and hyperbolic functions, and also a DSP-oriented architecture for the logarithm. These tasks are much simplified thanks to our generic fixed-point function evaluator.

Now that the our floating-point libm already contains some very efficient components we are receiving positive feedback regarding the *FPPipeline* operator, which assembles a full floating-point datapath starting from a C-like description. This encourages us to further improve this component. Some possible enhancements include:

- adding support for flow-control statements (`if()...else`) which are simply implemented as multiplexers.
- the possibility to use custom floating-point formats for each operation.
- support boolean and custom precision integer and fixed-point data-types and array structures. Supporting these new data-types at the level of FPPipeline requires extending our operator library with basic fixed-point operators, but also providing a better interface for FunctionEvaluator, one which doesn't require the user to manually pre-scale the input and output of the operator to $[0, 1)$.

FloPoCo is used as a backend for operator generation for the HLS tool developed in the PandA project. However, this tool is tightly coupled with FloPoCo, which might not always be possible for vendor tools. In order to facilitate the use of FloPoCo arithmetic pipelines by these tools, a standardized interface is required. The FPPipeline component might be a good entry point towards interfacing FloPoCo to these tools.

In this thesis we have ourselves explored using the frequency-driven pipelined FloPoCo operators in the context of synthesizing perfect loop nests with uniform data dependencies, where loop iterations carry dependencies. It is known that for this restricted class of applications, when the inner statement is implemented as a deeply pipelined operator, current HLS tools have poor performances. We have shown here that we can efficiently reschedule the code execution to account for the operator's pipeline depth and keep the operator's pipeline busy. Future work in this direction includes extending the supported application class to codes with non-uniform dependencies.

Using multiple processing elements to accelerate the execution poses new problems. Indeed, polyhedral parallelization techniques can help us find and exploit the parallelism. However, more FPGA-specific techniques, which allow minimizing communications and resources can be manually found. We believe that there is still research to be done for the FPGA context. Particularly, we plan to extend and generalize the parallelization technique presented for the Jacobi stencil for non-symmetric data dependencies. Once these techniques are matured, they can be included in commercial HLS tools and applied each time this type of kernels are encountered.

Considering that the FPGAs are flexible and efficient enough to implement custom datapaths with FloPoCo, we hope that the top entry of the top 10 predictions of the FFCM conference [2] which reads "FPGAs will have floating point cores", will turn-out out to be wrong! Having in mind that GPUs already offer massive numbers of floating-point cores, FPGAs should go further on their own way, which has always been flexibility. Flexibility allows for application-specific mix-and-match between integer, fixed point and floating point numbers, between adders, multipliers, dividers, and even more exotic operators [145, 62].

We have shown in this work that by using this flexibility, FPGAs can be used with success to accelerate both arithmetic datapaths and also small computational kernels. The speedup of these datapaths over microprocessor systems can be significant, with a much lower power consumption. However, why are then FPGAs still being used so confidentially on the acceleration market, where in just a few years GPUs have become so common? The answer to this question might not be the higher price of these devices, but may be the bad reputation regarding the programming and interfacing of these devices. Nevertheless, last years have brought significant progress in these

2. http://www.fccm.org/top10.php

directions (the QSys system builder from Altera and the Embedded Design Kit from Xilinx). We hope FloPoCo also participates to this effort of making FPGA-based acceleration more common.

Many things remain on the roadmap. However, the FPGA community's gratitude towards our initiative (keeping things free and open-source) and what we have accomplished so far, acts as an important driving force for us, FloPoCo developers.

The best and most versatile free floating point unit out there is FloPoCo.
Check it out: http://flopoco.gforge.inria.fr/
It outperforms even expensive professional solutions.
(http://embdev.net/topic/215370 Forum)

FloPoCo is really an amazing piece of software, can handle complex floating point exponentials, trig, as well as standard operators. The nice thing about it is one of the input modes, you just write the expression in a text file and it will generate an FPU with the required hardware to perform the operations in the given expression...
(http://www.edaboard.com/thread202615.html Forum)

Bibliography

[1] CatapultC Synthesis. http://www.mentor.com.

[2] GMP, the GNU multi-precision library. http://gmplib.org/.

[3] Impulse-C. http://www.impulseaccelerated.com.

[4] MPFR library: multiple-precision floating-point computations with correct rounding. http://www.mpfr.org/.

[5] PLDA EZDMA2 DMA for PCI Express Hard IP. http://www.plda.com.

[6] ISE 11.4 CORE Generator IP. http://www.xilinx.com/ipcenter/coregen/updates_11_4.htm.

[7] DN7020K10 'Uncle of Monster' Altera Stratix IV ASIC Prototyping Engine. http://www.dinigroup.com.

[8] Forte Design Systems: Cynthesizer. http://www.forteds.com.

[9] Megawizard plug-in manager. http://www.altera.com.

[10] Polylib – a library of polyhedral functions. URL http://www.irisa.fr/polylib.

[11] *Our History*, 1985. http://www.xilinx.com/company/history.htm.

[12] *Virtex-II Platform FPGA Handbook*, 2000.

[13] FFT/IFFT block floating point scaling, 2005. http://www.altera.com/literature/an/an404.pdf.

[14] *StratixII Device Handbook*, 2007. http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf.

[15] *Virtex-II Platform FPGA User Guide*, 2007. www.xilinx.com/support/documentation/user_guides/ug002.pdf.

[16] *XtremeDSP for Virtex-4 FPGAs*, 2008. http://www.xilinx.com/support/documentation/user_guides/ug073.pdf.

[17] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–58, 29 2008. doi: 10.1109/IEEESTD.2008.4610935.

[18] *Virtex-4 FPGA User Guide*, 2008. http://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

[19] *Spartan-3 Generation FPGA User Guide*, 2009.

[20] *Virtex-6 FPGA Configurable Logic Block User Guide*, 2009. http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.

[21] *Virtex-5 FPGA XtremeDSP Design Considerations*, 2010. http://www.xilinx.com/support/documentation/user_guides/ug193.pdf.

[22] *StratixIII Device Handbook*, 2010. http://www.altera.com/literature/hb/stx3/stratix3_handbook.pdf.

[23] *Virtex-5 FPGA User Guide*, 2010. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[24] *Virtex-6 FPGA Memory Resources User Guide*, 2010. http://www.xilinx.com/support/documentation/user_guides/ug363.pdf.

[25] *Virtex-6 FPGA DSP48E1 Slice*, 2011. http://www.xilinx.com/support/documentation/user_guides/ug369.pdf.

[26] LogiCORE IP CORDIC v4.0, 2011. http://www.xilinx.com/support/documentation/ip_documentation/cordic_ds249.pdf.

[27] *StratixIV Device Handbook*, 2011. http://www.altera.com/literature/hb/stratix-iv/stx4_5v1.pdf.

[28] *StratixV Device Handbook*, 2011. http://www.altera.com/literature/hb/stratix-v/stratix5_handbook.pdf.

[29] DSP Builder – Advanced blockset with timing-driven Simulink synthesis, 2011. http://www.altera.com/products/software/products/dsp/adsp-builder.html.

[30] QuartusII Design Software, 2011. http://www.altera.com/products/software/quartus-ii/subscription-edition/design-entry-synthesis/qts-des-ent-syn.html.

[31] Floating-point megafunctions user guide, January 2011. http://www.altera.com/literature/ug/ug_altfp_mfug.pdf.

[32] ISE Design Suite 13, 2011. http://www.xilinx.com/support/download/index.htm.

[33] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12 (3):288 – 298, march 2004. ISSN 1063-8210. doi: 10.1109/TVLSI.2004.824300.

[34] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: http://doi.acm.org/10.1145/1465482.1465560. URL http://doi.acm.org/10.1145/1465482.1465560.

[35] Raymond Andraka. Hybrid floating point technique yields 1.2 gigasample per second 32 to 2048 point floating point FFT in a single FPGA. In *High Performance Embedded Computing Workshop*, 2006. http://www.andraka.com/files/HPEC2006.pdf.

[36] J.M. Arnold, D.A. Buell, D.T. Hoang, D.V. Pryor, N. Shirazi, and M.R. Thistle. The Splash 2 processor and applications. In *Computer Design: VLSI in Computers and Processors*, pages 482 –485, oct 1993. doi: 10.1109/ICCD.1993.393329.

[37] M.G. Arnold and S. Collange. A real/complex logarithmic number system ALU. *IEEE Transactions on Computers*, 60(2):202 –213, feb. 2011. ISSN 0018-9340. doi: 10.1109/TC.2010.154.

[38] D. Bakalis, K. D. Adaos, D. Lymperopoulos, M. Bellos, H. T. Vergos, G. Ph. Alexiou, and D. Nikolos. A core generator for arithmetic cores and testing structures with a network interface. *Journal of Systems Architecture*, 52:1–12, January 2006. ISSN 1383-7621. doi: 10.1016/j.sysarc.2004.12.006. URL http://portal.acm.org/citation.cfm?id=1131948.1131949.

[39] Zachary K. Baker and Viktor K. Prasanna. Efficient hardware data mining with the apriori algorithm on fpgas. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:3–12, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/FCCM.2005.31.

[40] Sebastian Banescu, Florent de Dinechin, Bogdan Pasca, and Radu Tudoran. Multipliers for floating-point double precision and beyond on FPGAs. In *International Workshop on Higly-Efficient Accelerators and Reconfigurable Technologies (HEART)*. ACM, jun 2010.

[41] Cedric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, Olivier Temam, A Group, and Inria Rocquencourt. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03), LNCS*, pages 209–225, 2003.

[42] Rachid Beguenane, Jean-Luc Beuchat, Jean-Michel Muller, and Stéphane Simard. Modular multiplication of large integers on FPGA. In *Asilomar Conference on Signals, Circuits and Systems*, pages 1361–1365, 2005.

[43] P. Belanović and M. Leeser. A library of parameterized floating-point modules and their use. In *International Conference on Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 657–666. Springer, 2002.

[44] Michael R. Bodnar, John R. Humphrey, Petersen F. Curt, James P. Durbano, and Dennis W. Prather. Floating-point accumulation circuit for matrix applications. In *International Symposium on Field-Programmable Custom Computing Machines*, pages 303–304. IEEE Computer Society, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/FCCM.2006.41.

[45] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM International Conference on Programming Languages Design and Implementation (PLDI'08)*, pages 101–113, Tucson, Arizona, jun 2008.

[46] Pierre Boulet and Paul Feautrier. Scanning polyhedra without do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 4–9, 1998. ISBN 0-8186-8591-3.

[47] W.S. Briggs and D.W. Matula. A 17×69 bit multiply and add unit with redundant binary feedback and single cycle latency. In *11th Symposium on Computer Arithmetic*, pages 163 –170, 1993. doi: 10.1109/ARITH.1993.378096.

[48] N. Brisebarre and S. Chevillard. Efficient polynomial l^∞-approximations. In *18th IEEE Symposium on Computer Arithmetic (ARITH 18)*, pages 169–176, Los Alamitos, CA, June 2007. IEEE Computer Society.

[49] Nicolas Brisebarre, Florent de Dinechin, and Jean-Michel Muller. Integer and floating-point constant multipliers for FPGAs. *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 0:239–244, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/ASAP.2008.4580184.

[50] P. E. Ceruzzi. The early computers of Konrad Zuse, 1935 to 1945. *Annals of the History of Computing*, 3(3):241–262, 1981.

[51] K.D. Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, May 1994.

[52] R.C.C. Cheung, Dong-U Lee, W. Luk, and J.D. Villasenor. Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(8):952 –962, aug. 2007. ISSN 1063-8210. doi: 10.1109/TVLSI.2007.900748.

[53] S. Chevillard, J. Harrison, M. Joldes, and Ch. Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoretical Computer Science*, 412(16):1523 – 1543, 2011.

[54] S. Chevillard, Ch. Lauter, and M. Joldes. Users manual for the Sollya tool, Release 2.9. http://sollya.gforge.inria.fr/, February 2011.

[55] Philippe Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *International Conference on Supercomputing (ICS'96)*, pages 278–285. ACM, 1996.

[56] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[57] Octavian Creţ, Florent de Dinechin, Ionuţ Trestian, Radu Tudoran, Laura Creţ, and Lucia Văcariu. FPGA-based acceleration of the computations involved in transcranial magnetic stimulation. In *Southern Programmable Logic Conference*, pages 43–48. IEEE, 2008.

[58] L. Dadda and V. Piuri. Pipelined adders. *IEEE Transactions on Computers*, 45(3):348–356, Mar 1996. ISSN 0018-9340. doi: 10.1109/12.485573.

[59] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J.-M. Muller. CR-LIBM, a library of correctly-rounded elementary functions in double-precision. Technical report, LIP Laboratory, Arenaire team, Available at https://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf, December 2006.

[60] Florent de Dinechin and Arnaud Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, 2005.

[61] Florent de Dinechin and G. Villard. High precision numerical accuracy in physics research. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 559:207–210, 2006.

[62] Florent de Dinechin, Jérémie Detrey, Ionuţ Trestian, Octavian Creţ, and Radu Tudoran. When FPGAs are better at floating-point than microprocessors. Technical Report ensl-00174627, École Normale Supérieure de Lyon, 2007. http://prunel.ccsd.cnrs.fr/ensl-00174627.

[63] Florent de Dinechin, Bogdan Pasca, Octavian Creţ, and Radu Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *IEEE International Conference on Field-Programmable Technology*, pages 33–40. IEEE, 2008.

[64] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. Multiplicative square root algorithms for FPGAs. In *International Conference on Field Programmable Logic and Applications*. IEEE, aug 2010.

[65] Florent de Dinechin, Jean-Michel Muller, Bogdan Pasca, and Alexandru Plesco. An FPGA architecture for solving the Table Maker's Dilemma. In *International Conference on Application-specific Systems, Architectures and Processors*, 2011.

[66] Jérémie Detrey. *Arithmétiques réelles sur FPGA : virgule fixe, virgule flottante et système logarithmique*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, January 2007. URL http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2007/PhD2007-01.pdf.

[67] Jérémie Detrey and Florent de Dinechin. FPLibrary: operators for the design of real number processing cores on FPGA, sep 2003. URL http://perso.ens-lyon.fr/jeremie.detrey/publications/pub/Det2003_rnc_slides.pdf.

[68] Jérémie Detrey and Florent de Dinechin. A parameterizable floating-point logarithm operator for FPGAs. In *39th Asilomar Conference on Signals, Systems & Computers*, pages 1186–1190, Pacific Grove, CA, USA, November 2005. IEEE Signal Processing Society. doi: 10.1109/ACSSC.2005.1599948.

[69] Jérémie Detrey and Florent de Dinechin. Table-based polynomials for fast hardware function evaluation. In *Application-specific Systems, Architectures and Processors*, pages 328–333. IEEE, 2005.

[70] Jérémie Detrey and Florent de Dinechin. A parameterized floating-point exponential function for FPGAs. In *Field-Programmable Technology*. IEEE, 2005.

[71] Jérémie Detrey and Florent de Dinechin. Floating-point trigonometric functions for FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 29–34, Amsterdam, Netherlands, aug 2007. IEEE. doi: 10.1109/FPL.2007.4380621.

[72] Jérémie Detrey and Florent de Dinechin. Parameterized floating-point logarithm and exponential functions for FPGAs. *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing*, 31(8):537–545, 2007. doi: 10.1016/j.micpro.2006.02.008.

[73] Jérémie Detrey and Florent de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *Journal of VLSI Signal Processing*, 49(1):161–175, 2007. doi: 10.1007/s11265-007-0048-7.

[74] Jérémie Detrey, Florent de Dinechin, and Xavier Pujol. Return of the hardware floating-point elementary function. In *18th IEEE Symposium on Computer Arithmetic (ARITH 18)*, pages 161–168, Montpellier, France, jun 2007. IEEE Computer Society. doi: 10.1109/ARITH.2007.29.

[75] Padma Devi, Ashima Girdher, and Balwinder Singh. Improved carry select adder with reduced area and low power consumption. *International Journal of Computer Applications*, 3(4):14–18, June 2010. Published By Foundation of Computer Science.

[76] Christopher Doss and Robert L. Riley, Jr. FPGA-based implementation of a robust IEEE-754 exponential unit. In *Field-Programmable Custom Computing Machines*, pages 229–238. IEEE, 2004. doi: 10.1109/FCCM.2004.38.

[77] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *ACM/SIGDA symposium on Field-Programmable Gate Arrays (FPGA)*, 2005.

[78] P. Echeverría and M. López-Vallejo. An FPGA implementation of the powering function with single precision floating-point arithmetic. In *Proceedings of the 8th Conference on Real Numbers and Computers*, Santiago de Compostela, Spain, 2008.

[79] Pedro Echeverría, David Thomas, Marisa López-Vallejo, and Wayne Luk. An FPGA run-time parameterisable log-normal random number generator. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *Lecture Notes in Computer Science*, pages 221–232. 2008. http://dx.doi.org/10.1007/978-3-540-78610-8_22.

[80] M. Ercegovac. Radix-16 evaluation of certain elementary functions. *IEEE Transactions on Computers*, C-22(6):561–566, 1973.

[81] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004. ISBN 1-55860-798-6.

[82] Hossam A. H. Fahmy and Michael J. Flynn. The case for a redundant format in floating point arithmetic. In *16th Symposium on Computer Arithmetic*, pages 95–102. IEEE Computer Society, 2003.

[83] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3): 243–268, 1988.

[84] Paul Feautrier and Christian Lengauer. The polyhedron model. *Encyclopedia of Parallel Computing*, 2011.

[85] A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang. Automating customisation of floating-point designs. In *Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 523–533. Springer, 2002.

[86] Gokul Govindu, Ronald Scrofano, and Viktor K. Prasanna. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In *International Conference on Engineering Reconfigurable Systems and Algorithms*, pages 137–148, 2005.

[87] Mariusz Grad and Christian Plessl. An open source circuit library with benchmarking facilities. In *The International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 144–150, 2010.

[88] Y. Gu, T. VanCourt, and M.C. Herbordt. Accelerating molecular dynamics simulations with configurable circuits. *Computers and Digital Techniques*, 153(3):189 – 195, may 2006. ISSN 1350-2387. doi: 10.1049/ip-cdt:20050182.

[89] John Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics*, pages 113–130, 1999.

[90] Chuan He, Guan Qin, Mi Lu, and Wei Zhao. Group-alignment based accurate floating-point summation on FPGAs. pages 136–142, 2008. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.3335.

[91] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 2nd edition, 2002. ISBN 0-89871-521-0.

[92] Miaoqing Huang and David Andrews. Modular design of fully pipelined accumulators. In *IEEE International Conference on Field-Programmable Technology*, pages 118–125, 2010.

[93] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Faster floating-point square root for integer processors. In *International Symposium on Industrial Embedded Systems (SIES'07)*, pages 324 –327, july 2007. doi: 10.1109/SIES.2007.4297353.

[94] Nachiket Kapre and André DeHon. Accelerating SPICE Model-Evaluation using FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 0:37–44, 2009. doi: 10.1109/FCCM.2009.14. URL http://dx.doi.org/10.1109/FCCM.2009.14.

[95] N. G. Kingsbury and P. J. W. Rayner. Digital filtering using logarithmic arithmetic. *Electronic Letters*, 7:56–58, 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[96] D. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, 3rd edition, 1997.

[97] P. Kornerup and D. W. Matula. Finite precision lexicographic continued fraction number systems. In *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1985. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[98] Ulrich Kulisch. Circuitry for generating scalar products and sums of floating point numbers with maximum accuracy. United States Patent 4622650, 1986.

[99] Ulrich W. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, 2002. ISBN 3211838708.

[100] M. Langhammer. Floating point datapath synthesis for FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 355 –360, sept. 2008. doi: 10.1109/FPL.2008.4629963.

[101] M. Langhammer. Floating point datapath synthesis for FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 355 –360, sept. 2008. doi: 10.1109/FPL.2008.4629963.

[102] Martin Langhammer. Foundation of FPGA acceleration, 2008. http://www.rssi2008.org/proceedings/industry/Altera.pdf.

[103] Christoph Lauter and Florent de Dinechin. Optimising polynomials for floating-point implementation. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 7–16, 2008.

[104] B. Lee and N. Burgess. Parameterisable floating-point operations on FPGA. In *Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1064 – 1068, nov. 2002. doi: 10.1109/ACSSC.2002.1196947.

[105] D.-U. Lee, A.A. Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Transactions on Computers*, 54(12):1520 – 1531, dec. 2005. ISSN 0018-9340. doi: 10.1109/TC.2005.201.

[106] D.-U. Lee, J.D. Villasenor, W. Luk, and P.H.W. Leong. A hardware gaussian noise generator using the Box-Muller method and its error analysis. *IEEE Transactions on Computers*, 55 (6), 2006.

[107] V. Lefèvre. *Moyens Arithmétiques Pour un Calcul Fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[108] V. Lefèvre. New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*, pages 68–75. IEEE Computer Society Press, Los Alamitos, CA, June 2005.

[109] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, Vail, CO, June 2001. doi: http://doi.ieeecomputersociety.org/10.1109/ARITH.2001.930115.

[110] V. Lefèvre, D. Stehlé, and P. Zimmermann. Worst cases for the exponential function in the IEEE 754r decimal64 format. In *Reliable Implementation of Real Number Algorithms: Theory and Practice*, volume 5045 of *Lecture Notes in Computer Sciences*, pages 114–126. Springer, Berlin, 2008.

[111] P.H.W. Leong. Recent trends in FPGA architectures and applications. In *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, pages 137 –141, jan. 2008. doi: 10.1109/DELTA.2008.14.

[112] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, 2001.

[113] Y. Li and W. Chu. Implementation of single precision floating point square root on FPGAs. In *FPGAs for Custom Computing Machines*, pages 56–65. IEEE, 1997.

[114] G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *FPGAs for Custom Computing Machines*. IEEE, 2002.

[115] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL'97)*. ACM Press, jan 1997.

[116] Z Luo and M Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions on Computers*, 49:208–218, 2000.

[117] Bryce Mackin and Nathan Woods. FPGA acceleration in HPC: A case study in financial analytics. *XtremeData Whitepaper*, November 2006. http://oldwww.xtremedatainc.com/pdf/FPGA_Acceleration_in_HPC.pdf.

[118] Manouk Manoukian and George Constantinides. Accurate floating point arithmetic through hardware error-free transformations. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 6578 of *Lecture Notes in Computer Science*, pages 94–101. 2011. http://dx.doi.org/10.1007/978-3-642-19475-7_11.

[119] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[120] Peiro Marcos Martinez, Valls Javier, and Boemo Eduardo. On the design of FPGA-based multioperand pipeline adders. In *XII Design of Circuits and Integrated System Conference*, 1997.

[121] P.L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362 – 369, march 2005. ISSN 0018-9340. doi: 10.1109/TC.2005.49.

[122] J. M. Muller. *Arithmétique des Ordinateurs*. Masson, Paris, 1989.

[123] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, 2nd edition, 2006. ISBN 0-8176-4372-9.

[124] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[125] Romana Naik and Hakim Shah. Synthesis of carry select adder in 65 nm FPGA. In *TENCON 2008 - 2008 IEEE Region 10 Conference*, pages 1–6, nov. 2008. doi: 10.1109/TENCON.2008.4766397.

[126] D. Strenski O. Storaasli, W. Yu and J. Maltby. Performance evaluation of FPGA-based biological applications. Cray Users Group Proceedings, 2007. http://ft.ornl.gov/~olaf/pubs/CUG07Olaf17M07.pdf.

[127] E. Pearse O'Grady and Chung-Hsien Wang. Performance limitations in parallel processor simulations. *Transactions of the Society for Computer Simulation International*, 4:311–330, October 1987. ISSN 0740-6797. URL http://portal.acm.org/citation.cfm?id=58390.58393.

[128] Steve Perry. Model based design needs high level synthesis: a collection of high level synthesis techniques to improve productivity and quality of results for model based electronic design. In *Conference on Design, Automation and Test in Europe*, DATE '09, pages 1202–1207, 2009. ISBN 978-3-9810801-5-5. URL http://portal.acm.org/citation.cfm?id=1874620.1874909.

[129] J.-A. Pineiro, M.D. Ercegovac, and J.D. Bruguera. Algorithm and architecture for logarithm, exponential, and powering computation. *Computers, IEEE Transactions on*, 53(9):1085 – 1096, sept. 2004. ISSN 0018-9340. doi: 10.1109/TC.2004.53.

[130] José-Alejandro Piñeiro and Javier D. Bruguera. High-speed double-precision computation of reciprocal, division, square root, and inverse square root. *IEEE Transactions on Computers*, 51(12):1377–1388, December 2002. doi: 10.1109/TC.2002.1146704.

[131] Alexandru Plesco. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators*. PhD thesis, École Normale Supérieure de Lyon, 2010.

[132] Robin Pottathuparambil and Ron Sass. A parallel/vectorized double-precision exponential core to accelerate computational science applications. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 285–285, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2. doi: http://doi.acm.org/10.1145/1508128.1508198. URL http://doi.acm.org/10.1145/1508128.1508198.

[133] Thomas B. Preußer and Rainer G. Spallek. Mapping basic prefix computations to fast carry-chain structures. In *International Conference on Field Programmable Logic and Applications*, pages 604–608. IEEE, aug 2009.

[134] Eric Roesler and Brent E. Nelson. Novel optimizations for hardware floating-point units in a modern FPGA architecture. In *International Conference on Field-Programmable Logic and Applications*, pages 637–646, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44108-5. URL http://portal.acm.org/citation.cfm?id=647929.740071.

[135] Antonio Roldao Lopes and George Constantinides. A fused hybrid floating-point and fixed-point dot-product for fpgas. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science*, pages 157–168. 2010. http://dx.doi.org/10.1007/978-3-642-12133-3_16.

[136] Arthur Umbelino Alves Rolim. Transformada rápida de fourier para fpga. www.cin.ufpe.br/~auar/Apresentacao%20FFT%202-6.ppt, 2011.

[137] David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Form. Methods Syst. Des.*, 14:75–125, January 1999. ISSN 0925-9856. doi: 10.1023/A:1008669628911. URL http://portal.acm.org/citation.cfm?id=607542.607571.

[138] Michael J. Schulte, Kent E. Wires, and James E. Stine. Variable-correction truncated floating point multipliers. In *Asilomar Conference on Signals, Circuits and Systems*, pages 1344–1348, 2000.

[139] E. M. Schwarz, M. M. Schmookler, and S. D. Trong. Hardware implementations of denormalized numbers. In *16th IEEE Symposium on Computer Arithmetic*, pages 70–78, Washington, DC, 2003. IEEE Computer Society. ISBN 0-7695-1894-X.

[140] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *International Symposium on Field-Programmable Custom Computing Machines*, page 155, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7086-X.

[141] Robert Shuler, Li Chen, Andrew J. Hartnett, and Dave Rutishauser. Low power supercomputing in space. http://www.nasa.gov/pdf/499470main_jsc_shuler_low_power_supercomputing_in_space.pdf, 2010.

[142] Shreesha Srinath and Katherine Compton. Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks. In *Field Programmable Gate Arrays*, pages 51–58, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-911-4.

[143] D. Stehlé, V. Lefèvre, and P. Zimmermann. Worst cases and lattice reduction. In *Proceedings of the 16th Symposium on Computer Arithmetic (ARITH'16)*, pages 142–147, 2003.

[144] D. Stehlé, V. Lefèvre, and P. Zimmermann. Searching worst cases of a one-variable function. *IEEE Transactions on Computers*, 54(3):340–346, March 2005.

[145] Dave Strenski. FPGA floating point performance – a pencil and paper evaluation. *HPCWire*, January 2007. http://www.hpcwire.com/features/FPGA_Floating_Point_Performance.html.

[146] Song Sun and J. Zambreno. A floating-point accumulator for fpga-based high performance computing applications. In *IEEE International Conference on Field-Programmable Technology*, pages 493 –499, dec. 2009. doi: 10.1109/FPT.2009.5377624.

[147] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson, and C. R. Role. CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. *IEEE Journal of Solid-State Circuits*, 19(4):497–506, 1984.

[148] Zheng-Hua Tan, Børge Lindberg, and Enrico Bocchieri. Fixed-point arithmetic. In *Automatic Speech Recognition on Mobile Devices and over Communication Networks*, Advances in Pattern Recognition, pages 255–275. Springer London, 2008. ISBN 978-1-84800-143-5. http://dx.doi.org/10.1007/978-1-84800-143-5_12.

[149] Arnaud Tisserand. High-performance hardware operators for polynomial evaluation. *International Journal of High Performance Syststem Architectures*, 1:14–23, April 2007. ISSN 1751-6528. doi: http://dx.doi.org/10.1504/IJHPSA.2007.013288. URL http://dx.doi.org/10.1504/IJHPSA.2007.013288.

[150] Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *12th International Symposium on Field Programmable Gate Arrays*, pages 171–180, New York, NY, USA, 2004. ACM. ISBN 1-58113-829-6. doi: http://doi.acm.org/10.1145/968280.968305.

[151] I. H. Unwala and E. E. Swartzlander. Superpipelined adder designs. In *Circuits and Systems, 1993., ISCAS '93, 1993 IEEE International Symposium on*, pages 1841–1844, May 1993.

[152] Álvaro Vázquez and Elisardo Antelo. Implementation of the exponential function in a floating-point unit. *The Journal of VLSI Signal Processing*, 33:125–145, 2003. ISSN 0922-5773. URL http://dx.doi.org/10.1023/A:1021102104078. 10.1023/A:1021102104078.

[153] Xiaojun Wang, Sherman Braganza, and Miriam Leeser. Advanced components in the variable precision floating-point library. In *Field-Programmable Custom Computing Machines*, pages 249–258. IEEE Computer Society, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/FCCM.2006.21. URL http://dblp.uni-trier.de/db/conf/fccm/fccm2006.html#WangBL06.

[154] Maciej Wielgosz, Ernest Jamro, and Kazimierz Wiatr. Highly efficient structure of 64-bit exponential function implemented in FPGAs. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *Lecture Notes in Computer Science*, pages 274–279. 2008. http://dx.doi.org/10.1007/978-3-540-78610-8_28.

[155] Maciej Wielgosz, Ernest Jamro, and Kazimierz Wiatr. Accelerating calculations on the RASC platform: A case study of the exponential function. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5453 of *Lecture Notes in Computer Science*, pages 306–311. 2009. http://dx.doi.org/10.1007/978-3-642-00641-8_33.

[156] Maciej Wielgosz, Ernest Jamro, and Kazimierz Wiatr. Hardware implementation of the exponent based computational core for an exchange-correlation potential matrix generation. In *Parallel Processing and Applied Mathematics*, volume 6067 of *Lecture Notes in Computer Science*, pages 115–124. 2010. http://dx.doi.org/10.1007/978-3-642-14390-8_13.

[157] Kent E. Wires, Michael J. Schulte, and Don McCarley. FPGA resource reduction through truncated multiplication. In *International Conference on Field Programmable Logic and Applications*, pages 574–583. Springer-Verlag, 2001. ISBN 3-540-42499-7.

[158] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, 1994.

[159] SZ Xing and WWH Yu. FPGA adders: Performance evaluation and optimal design. *IEEE Design & Test Of Computers*, 15:24–29, 1998.

[160] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000. ISBN 0-7923-7933-0.

[161] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C.C. Cheung, D.-U. Lee, R.C.C. Cheung, and W. Luk. Reconfigurable acceleration for monte carlo based financial simulation. In *IEEE International Conference on Field-Programmable Technology*, pages 215 –222, dec. 2005. doi: 10.1109/FPT.2005.1568549.

[162] Ling Zhuo and Viktor K. Prasanna. High performance linear algebra operations on reconfigurable systems. In *ACM/IEEE conference on Supercomputing*. IEEE, 2005. doi: http://dx.doi.org/10.1109/SC.2005.31.