

Domain-Specific High Level Synthesis of Floating-Point Computations to Resource-Shared Microarchitectures

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des Grades Doktor-Ingenieur (Dr.-Ing.)

von

Dipl.-Inform. Björn Liebig

geboren in Hildesheim

Referenten: Prof. Dr.-Ing. Andreas Koch
Prof. Dr.-Ing. Mladen Berekovic

Datum der Einreichung: 29.01.2018

Datum der mündlichen Prüfung: 13.03.2018

Darmstadt, 2018
Hochschulkenziffer: D 17

Björn Liebig: Domain-Specific High Level Synthesis of Floating-Point Computations to Resource-Shared Microarchitectures
Darmstadt, Technische Universität Darmstadt,

Jahr der Veröffentlichung der Dissertation auf TUprints: 2018

Tag der mündlichen Prüfung: 13.03.2018

Version: 1.1

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung - Keine kommerzielle Nutzung - Keine Bearbeitung
4.0 International

<https://creativecommons.org/licenses/by-nc-nd/4.0>

ERKLÄRUNG ZUR DISSERTATION

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Björn Liebig

ABSTRACT

Many scenarios demand a high processing power often combined with a limited energy budget. A way to increase the processing power without increasing the power consumption is the use of hardware accelerators. While the implementation of such an accelerator as an application specific integrated circuit comes with very high development costs, reconfigurable logic devices such as FPGAs can lower the development costs and reduce development time, thus shortening time to market. To even further reduce development costs, the development of the circuit itself can be partially automated by applying a technique called high-level synthesis. However, current high-level synthesis approaches have difficulties to handle floating-point computations, especially when it comes to large blocks of floating-point code.

The focus in this thesis targets on the efficient implementation of floating-point arithmetic in FPGAs. To improve the performance new FPGA-optimized computing units are developed. This work proposes two new architectures for floating-point fused multiply-adds, and also presents and compares two low-latency dividers based on the Goldschmidt algorithm. The proposed units significantly outperform state-of-the-art in terms of latency.

Codes from domains such as control engineering and numerical simulation often contain large loop bodies holding with (tens of) thousands of double-precision floating-point operations. Both academic as well as industrial synthesis tools have great difficulty coping with such input programs. In this thesis, the academic compiler Nymble is extended to Nymble-RS, a branch with the necessary features to handle such large blocks of floating-point code.

The proposed techniques integrated in a tool chain that translates convex solvers defined in a domain specific language to hardware. The generated accelerators reach clock frequencies of more than 200 MHz. They exceed the performance of hardware generated by a state-of-the-art high-level synthesis tools by more than 5.7x and offers speed-ups of up to 5.2x over software executing on the 800 MHz Cortex-A9 CPUs used in typical reconfigurable system-on-chips.

Furthermore, the developed techniques are used to accelerate bioinformatics simulations defined in CellML language by using C-code as intermediate representation. The generated hardware exceeds the performance of current generation desktop CPUs in most cases, while requiring only 20...30% area on a mid-sized FPGA. Meanwhile, energy savings of up to 96% are reached.

ZUSAMMENFASSUNG

Viele Anwendungen erfordern eine hohe Rechenleistung, haben aber nur ein begrenztes Energiebudget. Eine Möglichkeit die Rechenleistung zu erhöhen, ohne den Stromverbrauch zu steigern, ist die Verwendung von Hardwarebeschleunigern. Während die Implementierung eines solchen Beschleunigers als anwendungsspezifische integrierte Schaltung mit sehr hohen Entwicklungskosten einhergeht, kann der Einsatz von rekonfigurierbaren Logikbausteinen wie z.B. FPGAs die Entwicklungskosten senken und sowohl die Entwicklungszeit als auch die Zeit bis zur Markteinführung deutlich verkürzen.

Um eine weitere Reduktion der Entwicklungskosten zu erreichen, kann die Entwicklung der Schaltung durch High-Level Synthese teilweise automatisiert werden. Aktuelle High-Level Synthese Ansätze haben jedoch Schwierigkeiten bei der Bewältigung von Fließkommaberechnungen, insbesondere wenn große Blöcken von Fließkomma-Code übersetzt werden müssen.

Der Schwerpunkt dieser Arbeit liegt auf der Implementierung von Fließkommaarithmetik in FPGAs. Um die Leistung zu verbessern, werden in dieser Arbeit neue, FPGA-optimierte Recheneinheiten vorgestellt. Es werden zwei neue Architekturen für kombinierte Fließkomma Multiplizier- und Addiereinheiten vorgeschlagen. Außerdem werden zwei Divisionseinheiten mit geringer Latenz vorgestellt, die auf dem Goldschmidt Divisionsverfahren beruhen. Die vorgeschlagenen Recheneinheiten übertreffen deutlich den bisherigen Stand der Technik in puncto Latenz.

Quellcodes aus Domänen wie der Steuerungstechnik und der numerischen Simulation enthalten oft große Schleifenkörper, die zehntausende von Fließkommaoperationen mit doppelter Genauigkeit enthalten. Sowohl akademische als auch industrielle High-Level-Synthesewerkzeuge haben große Schwierigkeiten, solche Eingabeprogramme zu verarbeiten. In dieser Arbeit wird daher der akademische Compiler Nymble zu Nymble-RS erweitert, einem Entwicklungszweig, der neue Techniken zur Verarbeitung von Zwischenergebnissen und zur effizienten Hardwarewiederverwendung enthält. Diese Techniken ermöglichen es, große Schleifenkörper effizient umzusetzen.

Der vorgestellte Compiler Nymble-RS wird außerdem in eine Werkzeugkette integriert, die, ausgehend von einer abstrakten Beschreibung einer Gruppe von konvexen Optimierungsproblemen in einer domänenspezifischen Sprache, Hardwarebeschleuniger zur Lösung dieser Probleme erzeugen kann. Die generierten Hardwarebeschleuniger erreichen Taktfrequenzen von über 200 MHz und übertreffen die Rechengeschwindigkeit von Hardware, die mit aktuellen kommerziellen High-Level Synthese Programmen erzeugt wurde, um mehr als Faktor 5.7x. Sie bieten

außerdem eine Beschleunigung von bis zu Faktor 5.2 gegenüber Software, die auf einer 800 MHz Cortex-A9 CPU ausgeführt wird.

Darüber hinaus werden die entwickelten Techniken verwendet, um Bioinformatik-Simulationen zu beschleunigen. Auch hier wird eine komplette Werkzeugkette vorgestellt, die von einer Definition in der domänenspezifischen Sprache CellML unter Verwendung von C-Code als Zwischendarstellung direkt zu einem Hardwarebeschleuniger führt. Die generierte Hardware benötigt nur 20. . . 30% der Fläche eines mittelgroßen FPGA und übertrifft in den meisten Fällen die Leistung von Desktop-CPUs der aktuellen Generation. Zudem werden Energieeinsparungen von bis zu 96% erreicht.

DANKSAGUNG

An dieser Stelle möchte ich mich bei allen bedanken, die mich während der Anfertigung dieser Arbeit unterstützt haben. Ein besonderer Dank gilt Prof. Dr.-Ing. Andreas Koch für die Betreuung dieser Arbeit und die zahlreichen thematische Diskussionen. Des Weiteren danke ich Prof. Dr.-Ing. Mladen Berekovic für das kurzfristige Erstellen des Zweitgutachtens. Weiterhin möchte ich mich bei meinen Kollegen an der TU Darmstadt für die gute Zusammenarbeit bedanken. Insbesondere Jens Huthman und Julian Oppermann danke ich für Ihre Arbeit an dem Compiler Nymble, der Ausgangsbasis für meine Arbeit ist. Vor allem möchte ich aber meiner Familie danken. Ohne die große Unterstützung durch meine Frau, Eltern und Schwiegereltern wäre der Abschluss dieser Arbeit sicher nicht möglich gewesen.

CONTENTS

1	INTRODUCTION	1
1.1	Hardware Acceleration	2
1.2	High-Level Synthesis	3
1.3	Floating-Point Arithmetic	4
1.4	Thesis Contributions	5
1.5	Thesis Structure	5
2	TECHNICAL BACKGROUND	7
2.1	Reconfigurable Devices	7
2.1.1	History of Reconfigurable Devices	7
2.1.2	Architecture of modern FPGAs	9
2.1.3	Adaptive Computer Systems	15
2.2	IEEE 754 Floating-Point Data Standard	18
2.2.1	Fused operations	20
2.2.2	Faithful Rounding	20
2.3	Convex Optimization	22
3	HIGH LEVEL SYNTHESIS FOR APPLICATION-SPECIFIC COMPUTING	25
3.1	Why use High-Level Synthesis	25
3.2	Working principle	26
3.2.1	Allocation	27
3.2.2	Scheduling	27
3.2.3	Binding	29
3.2.4	Loop Pipelining	29
3.3	Open Problems	29
4	PRIOR AND RELATED WORK	33
4.1	High Level Synthesis from C	33
4.2	Resource Sharing	34
4.3	Floating-Point Units for FPGAs	35
4.3.1	Algorithms for Digital Division	35
4.3.2	FPGA Divider Implementations	36
4.3.3	Tiling and Truncated Multiplication	37
4.3.4	Carry Save Adder	38
4.3.5	Fused Multiply Add	40
5	HARDWARE SYNTHESIS OF LARGE, IRREGULAR C CODE	41
5.1	The Nymble Compiler	41
5.2	Nymble-RS Improvements	45
5.2.1	Allocation	45
5.2.2	Iterative Scheduling	46
5.2.3	Schedule-Dependent Tree Height Optimization	47

5.2.4	BlockRAM-based Scratch Pad Memory	48
5.2.5	Support for Zynq / AXI as Target Platform	50
5.3	Operation Multiplexing using Distributed Microcode	53
5.4	Register Reduction for Intermediate Value Storage	56
5.4.1	Individual Delay Registers	56
5.4.2	Shift Registers	56
5.4.3	Spilling to Scratch Pad Memories	57
5.4.4	Recomputation vs. Storage	59
5.4.5	Combination of Methods	59
6	OPTIMIZED FLOATING-POINT OPERATIONS	61
6.1	Carry Save Fused Multiply Add	61
6.1.1	Classic FMA Architecture	62
6.1.2	Removing post-normalization	64
6.1.3	More efficient rounding	65
6.1.4	Eliminating the variable-distance shift	67
6.1.5	Floating-point representation using a PCS man- tissa	70
6.1.6	PCS-FMA Unit	72
6.1.7	Early leading zero anticipation	73
6.1.8	FCS-FMA for FPGAs with DSP pre-adders	76
6.1.9	Evaluation of the proposed FMA Units	77
6.1.10	Automatic P/FCS-FMA unit insertion in high- level synthesis	80
6.1.11	Discussion	81
6.2	Fast Lookup-based Divider	82
6.2.1	Goldschmidt Division for ASICs	82
6.2.2	Combination with Polynomial Approximation	82
6.2.3	FPGA implementation of TripleGS and PolyGS	84
6.2.4	FPGA specific optimization of latency and area	86
6.2.5	Error Analysis	91
6.2.6	Synthesis Results and Comparison to State of the Art	93
6.3	Hardware Units for Math Library Functions	96
7	CASE STUDY: CONVEX OPTIMIZATION SOLVERS	99
7.1	Tool flow and Domain-Specific Optimizations	99
7.2	Test Setup	101
7.3	Microcode-based Controllers	103
7.4	Handling of Intermediate Values	103
7.5	Design Space Exploration	104
7.6	Evaluation of 1ULP-Division Units	106
7.7	Evaluation of the FCS-FMA Unit	107
7.8	Comparison to State-of-the-Art High-Level Synthesis Tools	109
7.9	Speed-Up vs. Software on Zynq SoC	111
7.10	Discussion	111

8	CASE STUDY: SYNTHESIS OF CELLML SIMULATIONS	113
8.1	CellML-based Simulation	113
8.2	CellML-specific HLS with ODoST	114
8.3	Proposed Compilation Flow	115
8.4	Test Setup	116
8.5	Design Space Evaluation	117
8.6	Computation Accuracy	117
8.7	Comparison to State-of-the-Art HLS Tools	117
8.8	Performance / Energy relative to CPU	120
8.9	Performance / Energy relative to GPU	121
8.10	Impact of Spilling and Microcode	122
8.11	Impact of the proposed Low-Latency Division Units	123
8.12	Accelerator Tiling on Latest Generation FPGAs	125
8.13	Discussion	128
9	SUMMARY AND FUTURE WORK	129
9.1	Summary	129
9.2	Future Work	130
	Publications	133
	Bibliography	135

LIST OF FIGURES

Figure 2.1	FPLA (simplified, programming logic omitted)	8
Figure 2.2	Simple CLB with an optional flip-flop	9
Figure 2.3	Virtex-7 Slice, picture taken from [122]	10
Figure 2.4	Signal Propagation Delay Example	11
Figure 2.5	Basic Operation mode of a Virtex-7 DSP (image taken from [121])	12
Figure 2.6	Arria 10 DSP Block (image taken from [60])	13
Figure 2.7	FPGA Interconnect	14
Figure 2.8	FPGA Synthesis Tool Flow	15
Figure 2.9	Stand-alone architecture	16
Figure 2.10	Coupling via Peripheral Component Interconnect	17
Figure 2.11	Coupling via Front Side Bus	17
Figure 2.12	SPP and FPGA integrated on an System on a Chip (SoC)	18
Figure 2.13	IEEE 754 Double-precision format	19
Figure 2.14	Normalization and denormalization between floating-point (FP) operations	21
Figure 2.15	Faithful rounding of a floating-point mantissa	21
Figure 2.16	Typical excerpt from the generated solver code	23
Figure 2.17	Critical path of code in Listing 2.1	24
Figure 3.1	Steps of High-Level Synthesis Flow	26
Figure 3.2	Example for the construction and optimization of LLVM IR and CDFG	28
Figure 3.3	Loop execution with and without pipelining	30
Figure 4.1	DSP tiling and truncation of a 48 bit multiplier	38
Figure 4.2	Binary (b) and multiple carry save (cs) representations of the decimal number 14	39
Figure 4.3	Binary (b) and multiple partial carry save (pcs) representations of the decimal number 14	39
Figure 5.1	Compile Flow of Nymble/Nymble-RS	42
Figure 5.2	Schedule optimization performed to reduce register amount (red arrows indicate intermediate values that must be stored in a register)	47
Figure 5.3	Generic and schedule dependent tree height optimization at the example of a three adder tree	48
Figure 5.4	Detection and optimization of minimum / maximum computation trees	48
Figure 5.5	Collecting arrays marked for BlockRAMs storage	49

Figure 5.6	Necessary CDFG modification for arrays stored in BlockRAM	51
Figure 5.7	Zynq-7000 SoC Architectural overview (image taken from [129], some details are hidden to direct the focus on the AXI connections)	52
Figure 5.8	Proposed base design for the Zynq-7000 SoC target	52
Figure 5.9	Example for time-multiplexing operators	54
Figure 5.10	Using a shallower microcode ROMs, locally addressed by μ PCI	55
Figure 5.11	NOP chain of length two to delay intermediate results from Stage 1 to Stage 4	56
Figure 5.12	Reducing the number of NOPs using shift registers	57
Figure 5.13	Two NOP-chains are replaced by store (S) and load (L) operations	58
Figure 6.1	Example datapath with only the <i>A</i> and <i>C</i> input of FMAs in the critical path (marked red)	62
Figure 6.2	Classic FMA architecture [51] with IEEE 754-compliant operands and result (bit widths taken from [96], exponent and sign logic hidden to focus on mantissa computation)	63
Figure 6.3	Mantissa bit width increased by one bit to eliminate post-normalization	64
Figure 6.4	Modified FMA with increased mantissa width to eliminate post-normalization	65
Figure 6.5	Internal structure of mantissa multiplier with integrated rounding unit	66
Figure 6.6	Alternative structure of mantissa multiplier with integrated rounding unit for the rounding mode "round to nearest ties away from zero"	67
Figure 6.7	Modified FMA with rounding moved into the succeeding operation	68
Figure 6.8	Example for a normalization shift	68
Figure 6.9	The proposed block width	69
Figure 6.10	Examples for the block selection using 8 bit wide blocks)	69
Figure 6.11	Replacing shifting by a 6-to-1 multiplexer	70
Figure 6.12	Complete floating-point format with PCS mantissa	70
Figure 6.13	Proposed PCS-FMA architecture	72
Figure 6.14	Different forms of leading zeros in two's complement CS representation	73
Figure 6.15	Example for the uncertainty introduced by early leading zero anticipation	74

Figure 6.16	Loss of precision due to many leading 0s in maximally shifted A_M	75
Figure 6.17	FCS-FMA unit exploiting DSP block pre-adders	76
Figure 6.18	Latency (minimum clock period multiplied with the pipeline length) for FloPoCo, Xilinx and P/FCS-FMA operations on the Virtex-6	79
Figure 6.19	Computation pipeline used for accuracy comparison	79
Figure 6.20	Average mantissa error in x[50] (arithmetic mean over 20 computations)	80
Figure 6.21	Insertion of FCS-FMA units into a CDFG during high-level synthesis steps	81
Figure 6.22	TripleGS: Compute $Q = Y/X$ using a small lookup table and three Goldschmidt iterations	83
Figure 6.23	PolyGS: Compute $Q = Y/X$ using a polynomial approximation and a single Goldschmidt iteration	84
Figure 6.24	FPGA implementation of TripleGS (12 cycles, 30 DSPs)	85
Figure 6.25	FPGA implementation of the PolyGS approach (10 cycles, 20 DSPs)	87
Figure 6.26	Using the DSP pre-adders in Mul3 and Mul4	88
Figure 6.27	Using a partial carry save format for the result of Mul3	88
Figure 6.28	Truncated tiling for multipliers Mul1(a), Mul3+Mul4(b) and Mul5(c)	89
Figure 6.29	Maximum error scenario for truncated multiplication in Mul1	90
Figure 6.30	PolyGSopt implementation with reduced latency and area (8 cycles, 11 DSPs)	91
Figure 6.31	Newly developed <i>floor</i> operator	96
Figure 7.1	The tool chain used in this work (* = contribution of this thesis)	101
Figure 7.2	Dedicated state registers vs. microcode: Resources required	103
Figure 7.3	Initiation interval achieved by Nymble-RS using PolyGS, PolyGSOpt and the Xilinx divider IP	107
Figure 7.4	Maximum operation frequency (MHz) of the Nymble-RS results using PolyGS, PolyGSOpt and the Xilinx divider IP	107
Figure 7.5	ldl_solve() schedule length in absence of resource limitations with and without FMA insertion	108
Figure 7.6	Quality comparison of results of Nymble-RS and the industrial HLS tool	110

Figure 7.7	Comparison of results of Nymble-RS and the newest version of the industrial HLS tool	111
Figure 7.8	Wall clock time on ZC706 platform with and without hardware acceleration	112
Figure 8.1	CellML-to-accelerator compilation flow	115
Figure 8.2	Impact of spilling and Microcode on the design size	123
Figure 8.3	Impact of the proposed division units on the initiation interval	124
Figure 8.4	Impact of the proposed divider on the maximum operation frequency	124
Figure 8.5	Impact of the proposed divider on the single iteration wall time	125
Figure 8.6	System architecture used for multi-kernel synthesis	126
Figure 8.7	Floor plans used for 8, 12 and 16 kernels	126
Figure 8.8	Speed-up of multiple kernels on the UltraScale+ compared to the i7 6700K (single thread)	127

LIST OF TABLES

Table 2.1	Common formats defined in IEEE754-2008	19
Table 4.1	Traditional vs. modified Goldschmidt method	37
Table 5.1	LLVM passes used in Nymble (table taken from [4])	44
Table 6.1	Synthesis results	78
Table 6.2	Average energy consumption per multiply-add computation (nJ)	80
Table 6.3	Performance of placed & routed division units	95
Table 7.1	Number of operations in source code	102
Table 7.2	Post-synthesis area requirements for different intermediate value handling mechanisms	104
Table 7.3	Design space exploration. II=Initiation interval, WCT=Wall Clock Time	105
Table 7.4	Division Units Compared	106
Table 7.5	Impact of FMA insertion on the hardware kernel performance. II=Initiation interval, WCT=Wall Clock Time	109
Table 8.1	Large examples from CellML repository [75]	116
Table 8.2	Design space exploration. II=Initiation interval, WCT=Wall Clock Time	118
Table 8.3	Average relative error	119

Table 8.4	Comparison to accelerators created by industrial HLS tool (*=area constraints for pow violated by the industrial tool, f_{\max} in MHz)	120
Table 8.5	Execution Wall-Clock-Time (10 Cells, 1M iterations each), Power and Energy Consumption (FPGA vs CPU)	121
Table 8.6	Single FPGA kernel vs GK210 GPU	122
Table 8.7	Synthesis results for multiple kernel instances of test case A on the UltraScale+. II=Initiation interval, WCT=Wall Clock Time	127

LISTINGS

Listing 2.1	Solver computation structure (simplified)	23
Listing 3.1	Small example loop used in Figure 3.3	29
Listing 5.1	Single Scheduling Iteration	46
Listing 7.1	Code transformation required for non-field sensitive alias analysis	100
Listing 7.2	Transformation to canonical form of min operation	101
Listing 7.3	Simplified structure of the key function of the solver	102
Listing 8.1	Excerpt from CCGS output for [39], reformatted for readability	114

ACRONYMS

ACP	Accelerator Coherency Port
ACS	Adaptive Computing Unit
ALM	Adaptive Logic Module
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
CB	Connection Box
CDFG	Control Data Flow Graph
CGRA	Coarse-Grained Reconfigurable Array

CLB	Configurable Logic Blocks
CMDFG	Control Memory Data Flow Graph
CPLD	Complex Programmable Logic Device
CR	IEEE 754 Conforming Rounding
CS	Carry Save
CSA	Carry Save Adder
DSL	Domain Specific Language
FCS	Full Carry Save
FMA	Fused Multiply-Add
FP	Floating-Point
FPGA	Field Programmable Gate Array
FPLA	Field Programmable Logic Array
FPU	Floating-Point Unit
FR	Faithful Rounding
FSB	Front-Side Bus
GAL	Generic Array Logic
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IC	Integrated Circuit
II	Initiation Interval
IR	Internal Representation
LUT	Lookup Table
LZA	Leading Zero Anticipator
NOP	No-Operation
ODE	Ordinary Differential Equations
PAL	Programmable Array Logic
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PCS	Partial Carry Save

PL	Programmable Logic
PLA	Programmable Logic Arrays
PLL	Phase-Locked Loop
PROM	Programmable Read-Only Memory
PS	Processing System
QPI	QuickPath Interconnect
RTL	Register-Transfer Level
SB	Switch Box
SCU	Snoop Control Unit
SoC	System on a Chip
SPP	Software Programmable Processor
SRAM	Static Random-Access Memory
ULP	Unit of Least Precision
USB	Universal Serial Bus
WCT	Wall Clock Time
ZD	Zero Detector

INTRODUCTION

Emerging technologies transform ordinary everyday objects into smart, connected and autonomous "things" that need to be controlled by micro-electronics. This trend places high demands on computing units of any size, starting from small, low power controllers in wireless sensor nodes and moving up to high performance controllers placed in vehicles for autonomous driving.

While the demand for processing power is thereby growing continuously, old automatisms do not fulfil the needs any more. Growth of processor performance, reaches the it's physical limitations or at least slows down significantly. Especially clock frequencies are effected from the slowed growth since about 2007. Therefore, the single-thread performance in high-end processors has been increasing only slightly for some time.

On the other hand, heat dissipation and power consumption become more and more important. Especially for mobile (battery-powered) devices this importance is obvious. smartphone and Laptop reviews usually contain a section about battery life, often with and without load, as battery life is an important feature for the consumers. Battery-powered sensors even have a smaller power budget since battery change may be impossible or at least cause heavy maintenance costs. Sometimes energy harvesting is used to charge the battery [88], but also in such scenarios, higher power draw requires a larger harvester and therefore linearly increases the costs.

But power and heat are not only topics for wireless devices any more. Heat dissipation became a major problem even in high-performance processors as it effectively limits the total power consumption of those processors significantly. In [29] it is shown that high performance processor power consumption was growing exponentially between 1985 and 2005, but afterwards, it stagnates slightly above 100 Watt. At the same time, the exponential growth of clock frequency and single thread performance slows down. Nitrogen cooling experiments show that a higher heat dissipation would allow more power consumption and therefore higher single thread performance. However, nitrogen cooling is not possible in practice. But if dissipation of the heat is impossible, the only way to reduce heat is to reduce power draw. This makes energy efficiency an important goal for high-performance architectures as well.

Reducing power draw is also one of the main motivations of this work. This thesis is motivated by two applications that have heavy demands on performance and power budget: Convex optimization problem solvers, e.g. for collision avoidance algorithms in autonomous cars, and compute-intensive simulation of biological systems at the cell level.

1.1 HARDWARE ACCELERATION

If the capabilities of the microprocessor or CPU are not sufficient for the application, hardware accelerators can be used.

In some cases, programmable specialized processors are available as accelerators for a specific domain. For example, Graphics Processing Units (GPUs) are available in different sizes and can accelerate floating-point computations as long as massive parallelism is possible. Like microprocessors, they can only accelerate specific data types. Most GPUs focus on single-precision (32 bit) floating-point and offer only drastically reduced double-precision (64 bit) performance. An exception are expensive high-end cards like the NVidia Tesla [84].

An Application Specific Integrated Circuit (ASIC) can be designed to exactly fulfil the needs of the target application. An ASIC is an Integrated Circuit (IC) dedicated for a particular use only, e.g. a chip designed to run Bitcoin mining. ASICs deliver the higher performance and/or low power consumption in their specific domain. However, they cannot do anything outside this domain. For example, a Bitcoin miner ICs computing the SHA256 hash will not allow mining other cryptocurrencies based on different hash functions.

The major disadvantages of ASICs are the high investment costs, especially in small and medium quantities, and in any case the longer development time compared to a solution of the same problem with general purpose components and standard ICs. ASICs are described in a Hardware Description Language (HDL), such as Verilog or VHDL. Such a description is then converted to photolithographic masks used in the chip fabrication. This step can cost several million Euro regardless of how many chips are produced. This is one reason why ASICs are expensive for small and medium quantities. Furthermore, small errors may force additional iterations of the development cycles which increase costs and time to market event further.

Alternatively, ICs designed to be configured by a designer *after* manufacturing can be used. Most of them can even be configured multiple times to change the implemented circuit. They are called reconfigurable devices. The term "reconfigurable" here refers to the definition of the desired circuit structure. Like the ASIC, the reconfigurable devices circuit is formulated in a HDL, which is then translated into a configuration file using a (often vendor-specific) generator software. The reconfigurable device can be programmed using this configuration file **after manufacturing**. This means, that no photolithographic masks (or any other changes in IC production) are necessary, which reduces the development costs significantly. Devices that are not only one-time configurable, but reconfigurable, can also be reprogrammed at the customer site. This allows updates "in the field" for example to support other applications like mining an other cryptocurrency, or decoding a new video codec. Field Programmable Gate Arrays (FPGAs) are one type of reconfig-

urable devices. Some FPGAs even allow replacing parts of their circuit while they are in use. This feature is called partial reconfiguration.

Of course, this flexibility does not come for free. In order to offer programmability, the FPGA must contain additional circuits. A circuit that is programmed on an FPGA is usually slower and consumes more energy than an equivalent ASIC. Despite this, FPGAs can still offer significant performance and/or energy efficiency increases when compared to Microprocessors. In literature, FPGAs have been shown to accelerate geometric algebra [71], especially for inverse kinematics computations [50]. Using Low-Power FPGA, the energy consumption of controllers for noise reduction [5], structural health monitoring [2] and of FIR filters used in adaptronic applications [1] could be reduced compared to low-power micro controllers. Most recently, even Microsoft introduced FPGA-based technology to accelerate data center applications, such as their search engine Bing [94].

1.2 HIGH-LEVEL SYNTHESIS

Programming an FPGA requires writing a description in a HDL, which is still a very time consuming and expensive step. Additionally, it commonly requires experience in digital hardware design, computer architecture, and specialized programming tool flows. These skills will be unfamiliar to most software developers, so hardware specialists must be added to the development team of projects that require FPGA acceleration, leading to increased costs. Furthermore, testing of HDL code is done in so-called HDL simulators [80, 107]. It can be a very time consuming and therefore expensive task, especially for larger projects.

To open the potential of FPGAs up to a wider group of users, considerable research effort has been invested in automatic hardware compilers, often called High-Level Synthesis (HLS) tools. These compilers can translate programs written in software programming languages (often subsets of C) directly into a hardware description in an HDL. A recent survey [85] gives a good but incomplete overview about academic and industrial HLS tools currently available.

Often, HLS tools are restricted to generating a description of the actual hardware accelerators, sometimes referred to as the "hardware kernel". However, the hardware accelerator will often have interactions with the software running on a microprocessor. For example, a video decoder must get the input stream from somewhere. In most cases, the CPU will load it from a disc or a network and pass it to the hardware. The hardware, on the other side, must be able to receive the data from the CPU. This hardware-software communication must be implemented in Software and Hardware, so again, deep knowledge of the hardware and software is required. However, only few HLS tools can create the software-hardware interfaces as well as the low-level communication

mechanisms automatically. The tool "Nymble" [4] belongs to this group. It is used and extended in this thesis.

1.3 FLOATING-POINT ARITHMETIC

Many applications use floating-point arithmetic, often in double-precision. While in some cases, floating-point arithmetic can be replaced by more efficient fixed-point arithmetic, other applications rely on the large dynamic range that floating-point data types offer.

Most FPGAs contain functional blocks like integer-multiplier, adder or RAM (see Section 2.1.2 for more details). Lately, Altera (now bought by Intel) integrated single-precision multiply-add functional blocks in their Arria 10 FPGA series [58]. This thesis does neither target Arria 10 FPGAs nor single-precision, but the development of the Arria 10 proves that the interest in using FPGAs for floating-point computations is growing.

On other FPGAs (and for double-precision), floating-point computations must be composed using the resources available. That is, in most cases, functional blocks for integer addition and multiplication as well as Lookup-Tables (LUTs). FPGA vendors offer libraries of floating-point modules for most basic operations like addition, multiplication, division, square root, logarithm and others [57, 130]. The modules use the IEEE 754 floating-point format [55] as input and output format. They also perform the computation and rounding as defined in the standard, with some minor and well-documented exceptions, e.g. in the handling of numbers very close to zero (so called subnormals).

The IEEE 754 format was invented in 1985 as a technical standard for digital storage of floating-point values. It was supposed to improve reliability and portability of floating-point numbers. This is reasonable for the implementation of floating-point units in general purpose processors, as it ensures compatibility between different processor types and makes sure that computation on different machines leads to the same result. But IEEE 754 conformant floating-point units require many resources and show a high latency on FPGAs.

For many applications, full IEEE754-conformity is not actually required. This allows accelerators the use of other floating-point formats internally. For example, the floating-point library FloPoCo [36] uses a slightly different format, using 2 additional bits to represent exceptional values like "infinity" or "not a number". In IEEE754, these values are encoded in mantissa and exponent, which requires a special exception handling in each hardware operation. So, although FloPoCo requires two additional bits, it can save resources for exception handling logic, which allows smaller hardware. Furthermore, the rounding may be handled differently than defined in the standard. The concept of "fused multiply add" [51] or "fused data path" [72, 73] tries to remove unnecessary rounding steps between operations, while the faithful rounding

(FR) approach allows small rounding errors for improved latency and area efficiency.

1.4 THESIS CONTRIBUTIONS

This thesis examines the efficient implementation of floating-point applications reconfigurable computing platforms by using automated design flows. The following contributions are presented:

- A non IEEE 754 conformant divider for Xilinx FPGAs is presented. The divider uses functional blocks available on Virtex5 and newer Xilinx FPGAs to achieve a minimum latency per division. In terms of latency, the divider outperforms all other dividers currently available for Xilinx and Altera FPGAs. [Section 6.2]
- The thesis examines existing solutions and proposes two new architectures for floating-point Fused Multiply-Adds (FMAs), and also considers the impact of different in-fabric features of recent FPGA architectures. The units rely on different degrees of carry-save arithmetic. They improve the latency by up to 2.6x over the closest state-of-the-art competitor. [Section 6.1]
- The academic C-to-hardware compiler Nymble is extended with resource sharing capabilities to a version called Nymble-RS. In order to enable translation of large loop bodies with thousands of floating-point operations, multiplexer control and storage of intermediate values consume a lot of FPGA resources. Therefore, intermediate value spilling is proposed, a technique well known from software compilers. Furthermore, a microcoded architecture is proposed for multiplexer control. In addition, the Nymble compiler's memory interface is extended to support AXI-Interfaces for Control- and Memory accesses. [Chapter 5]
- Two case studies are presented in which the developed techniques are applied to real world applications. Nymble-RS and its resource sharing capabilities as well as the specialized floating-point computations have been used in synthesis of convex optimization solvers and biological simulations. In both cases, the application is defined in a Domain Specific Language (DSL), which is first converted to idiomatic C code. This C code is used as intermediate representation and forms the input for the Nymble-RS C-to-hardware compiler. [Chapter 7 and Chapter 8]

1.5 THESIS STRUCTURE

This thesis is structured as follows:

CHAPTER 2 gives a brief overview over the technologies used in this work. This includes an introduction to reconfigurable devices and their design flow. Furthermore, the IEEE 754 floating-point format is recapitulated. Parts of this section have been included in the background section of previous publications [6–8].

CHAPTER 3 explains the working principle of HLS. Further, benefits and problems in the use of HLS are discussed.

CHAPTER 4 gives an overview of the state of the art and related work in the fields of high-level synthesis, digital division algorithms, and FMA units.

CHAPTER 5 examines and discusses techniques used to improve the efficiency of resource sharing for large, floating-point intense loop bodies. The contribution in this chapter includes new approaches for the handling of intermediate results and for the generation of a microcoded architecture. Both approaches were published in [8]. Furthermore, method for automatic generation of scratch pad memories is presented, which was previously published in [4].

CHAPTER 6 proposes new FPGA-optimized floating-point units: two low-latency dividers, two FMA units and a floor and ceiling unit. While the dividers use IEEE 754 input and output format but non-IEEE 754 conformant rounding, the two low latency FMA units work with non-IEEE 754 conformant input and output formats. The work presented was previously published in [6, 7, 9].

CHAPTER 7 evaluates the techniques proposed in this thesis in a case study. Hardware accelerators are generated for convex optimization problem solvers. Starting from a solver description in a DSL, the study uses an existing tool to generate a C program, which is then transformed into a hardware-accelerated system using the proposed technologies. Most parts of this chapter have been published in [8].

CHAPTER 8 presents a second case study: The proposed technologies are applied to hardware synthesis of biological simulation accelerators. The case study was recently accepted for publication in [9].

CHAPTER 9 summarizes this work and gives an outlook for future pursuits.

TECHNICAL BACKGROUND

This section provides an overview of the technology references used in this thesis. At first, as most of this work is related reconfigurable devices, especially FPGAs, the history structure and required development tool flow of these devices is discussed in moderate detail. Then, the IEEE 754 floating-point format is recapitulated briefly, as it's understanding is required for the sections dealing with floating-point operations. Finally, a brief overview of convex optimization is given, which is a key component of many advanced control systems (e.g., trajectory planning for collision avoidance in autonomous cars) and has been a major motivation for this work.

2.1 RECONFIGURABLE DEVICES

Reconfigurable devices allow the configuration of the circuit *after* production. They have been developed to reduce non-recurring costs for IC production.

2.1.1 *History of Reconfigurable Devices*

In the early seventies semiconductor companies developed a mask-programmable ICs consisting arrays of programmable AND gates which is connected to arrays of programmable OR gates. These Programmable Logic Arrays (PLAs) were configured using a modified metal layer during production of the IC, so the programming was not done outside the factory.

Later on, Field Programmable Logic Arrays (FPLAs) became available which allowed programming by the user or "in the field". Similar to (field) Programmable Read-Only Memory (PROM), the programming was done using fuse or anti-fuse technology. While fuses allow the destruction of connections inside the chip by applying a high current during programming, anti-fuses employ an insulation layer that is burned irreversibly by the application of a (higher) programming voltage. Figure 2.1 presents the working principle of a FPLA. Note that the programming logic is omitted to improve readability.

Logical functions can also be implemented in a (P)ROM. For example a ROM with five input pins and three output pins can be used to implement three boolean functions with five inputs each. These three functions must share the same five inputs (or use a sub set of them). However, the result of *every* combination of inputs must be stored in the ROM, which increases the data to be stored. Therefore, a PLA

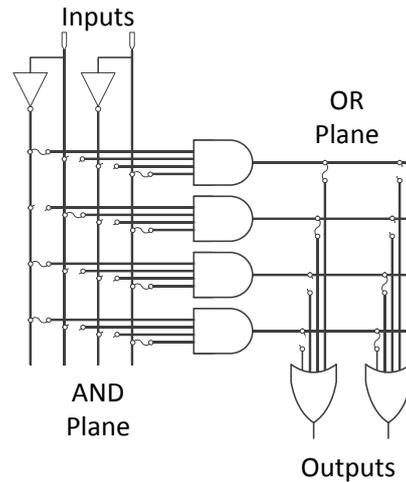


Figure 2.1: FPLA (simplified, programming logic omitted)

can often implement a logical function more efficient than read-only memory (less transistors are necessary to implement the function).

Differing from PLA architecture, Programmable Array Logic (PAL) devices have a set of transistor cells arranged on a programmable-AND plane connected to a fixed-OR plane. The architecture is simpler because the programmable OR array is omitted, which made the parts faster, smaller and cheaper. One-time programmable PLAs were later replaced by Generic Array Logic (GAL), which can be erased and re-programmed.

PALs and GALs are available only in small sizes. For larger logic circuits, Complex Programmable Logic Devices (CPLDs) can be used. A single CPLD IC contains the equivalent of several PALs linked by programmable interconnections. CPLDs can replace thousands or even hundreds of thousands of logical gates.

While the evolution of the PALs lead to CPLDs, a different development approach appeared. It resulted in devices based on the gate array technology. They are called Field Programmable Gate Array (FPGA). FPGAs consists of many Configurable Logic Blocks (CLBs) as well as configurable interconnects that allow the blocks to be connected to each other as required. The logic blocks are often based on one or multiple Lookup Tables (LUTs) and can be configured to implement any combinational function with the given number of inputs (see Figure 2.2). CLBs usually contain one or multiple flip-flops (FF), or even more complex storage elements, to optionally store the result. In addition, full-adders or multiplexers may be added by the FPGA vendor. It is very common for FPGAs to store the configuration in volatile memory, which makes a reprogramming after each power on necessary, but allows unlimited and faster programming.

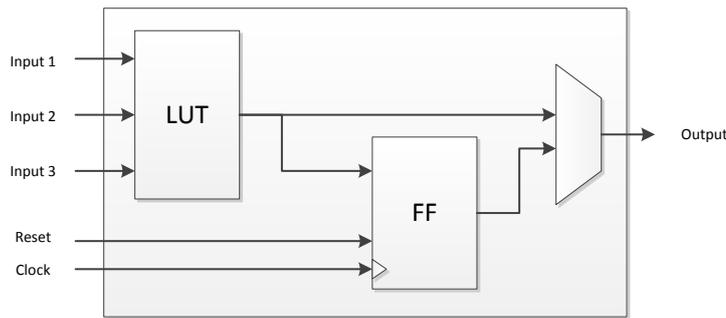


Figure 2.2: Simple CLB with an optional flip-flop

2.1.2 Architecture of modern FPGAs

As stated above, FPGAs consist of CLBs (and IO-Blocks) connected via a configurable interconnect. In addition to these generic blocks, modern FPGAs may contain multiple specialized functional blocks. All of these blocks are connected by the configurable interconnect:

- Blocks of Static Random-Access Memory (SRAM)
- Integer multipliers or multiply-add units of different bit widths (DSPs)
- Phase-Locked Loops (PLLs) or complex clock management units
- One or multiple clock distribution networks
- High performance IO connectors (e.g. PCI-Express or 100G Ethernet connectors, 30 Gigabit/s-Transceiver)
- Single-precision floating-point Multiply-Add Units
- Complete microprocessors, e.g. Power-PC or ARM cores

2.1.2.1 Configurable Logic Blocks

While the CLBs of a small, low power Microsemi (formerly Actel) Igloo FPGA are similar to the CLB shown in Figure 2.2, larger FPGAs from the major manufacturers Xilinx and Intel come with a more complex design.

In the Intel Stratix V, the CLBs are called Adaptive Logic Modules (ALMs). They contain four 3-Input LUTs and two 4-Input LUTs that can be combined to act as two 5-Input LUTs. Furthermore, two adders and four registers are available [56].

On current Xilinx FPGAs, e.g. Virtex-7, each CLB is divided into two parts, called Slices. Figure 2.3 shows a schematic view of such a slice. It contains four look-up tables, eight registers and numerous multiplexers, some of which are controlled by the configuration and others by the connected dynamic signals. Each LUT has six independent inputs (e.g.

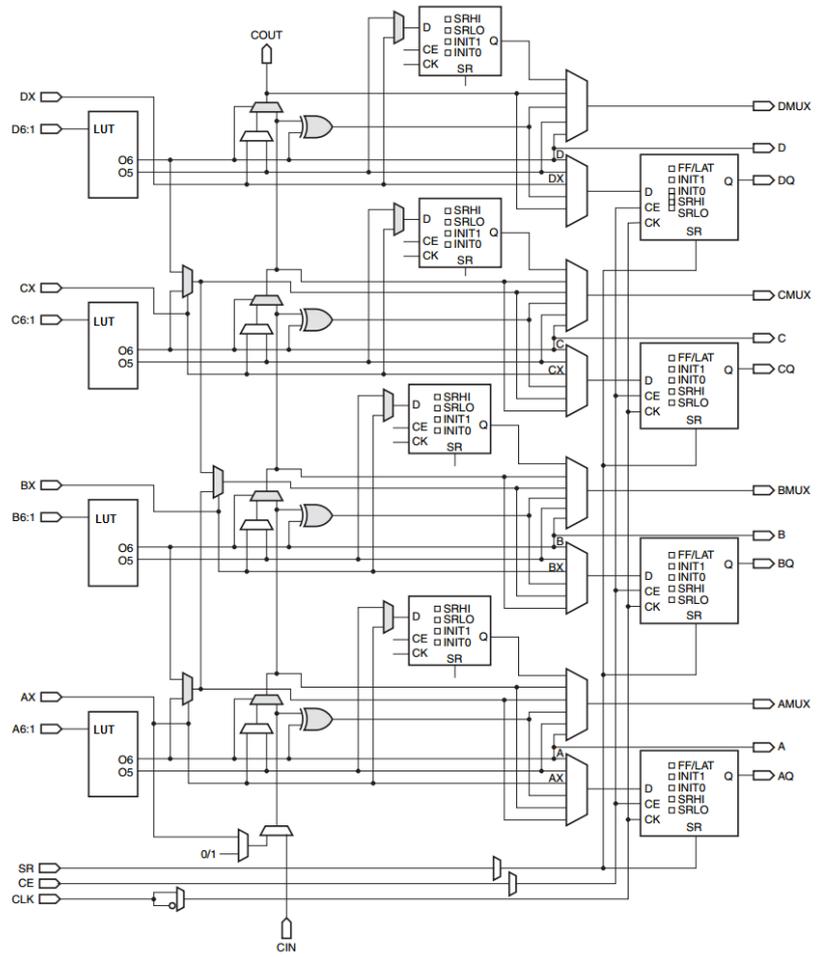


Figure 2.3: Virtex-7 Slice, picture taken from [122]

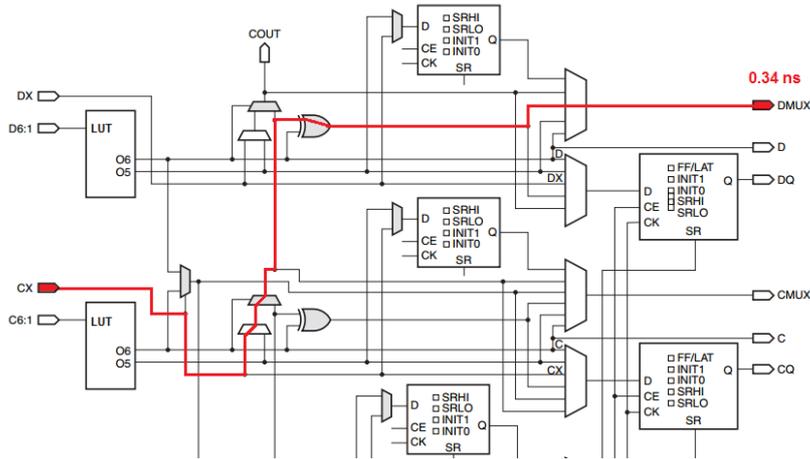


Figure 2.4: Signal Propagation Delay Example

A1 to A6) and two independent outputs (O5 and O6). It can implement a single six-input boolean function or two five-input Boolean functions, as long as these two functions share common inputs. The four left registers can be configured to be as either edge-triggered D-type flip-flops or level-sensitive latches. The four additional registers have been added to the Xilinx Slice design recently in the Virtex-6 generation. Prior CLBs contained only four registers. They can only be configured as edge-triggered D-type flip-flops. About 30% of the Slices contain additional logic that allows the use of the configuration memory of the LUT as random-access memory (called LUT-RAM or distributed RAM).

The maximum signal delay from each input to each output is known. For example, it takes 0.34 ns for a bit change in the input CX to reach DMUX a Virtex-7 of speed grade -2 [127] (see Figure 2.4).

2.1.2.2 DSP Blocks

Figure 2.5 shows a typical DSP block, in this case it shows the DSP48E1 taken from Virtex-7 documentation [121]. The block can be used to compute $(A + D) * B + C$. However, it is highly configurable. For example, the pre-adder ($A + D$) can be disabled, so that the DSP behaves similar to the Virtex-5 DSP (DSP48E), which had no pre-adder (and no D input). It is also possible to disable the multiplication. In this case, the DSP can for example act as 3-input adder.

Skipping the pre-adder will usually improve latency. To further improve the maximum operation frequency, there are multiple pipeline registers, each of which can be enabled or disabled separately. Maximum operation frequencies are known in advance for all possible combinations of register usage. They are documented in the "Switching Characteristics" of each Xilinx device family (e.g. [127]). Furthermore, all Xilinx Place and Route (P+R) tools have access to these values.

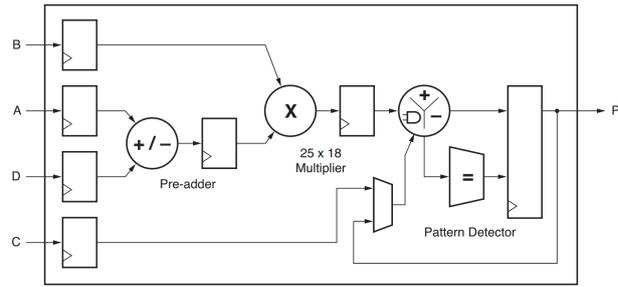


Figure 2.5: Basic Operation mode of a Virtex-7 DSP (image taken from [121])

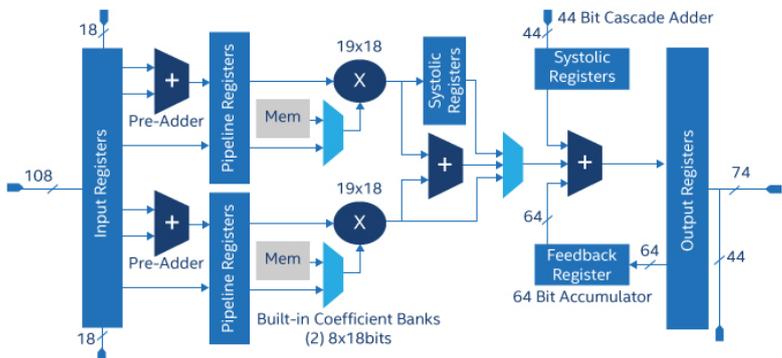
Note that Xilinx documentation is not always correct. For example, in *Switching Characteristics of Virtex-6* [120], the latency from input A or B to the register after the multiplication (MREG) is said to be 2.36ns (Speedgrade -3). However, in the Virtex-6 generation, the pre-adder was introduced which is why the latencies of A and B to MREG are no longer equal. The value of 2.36ns, given in [120], is correct for input A with the pre-adder enabled. The latency of input B to MREG is smaller (as there is no pre-adder in this path). The Xilinx ISE timing analysis tool reveals the correct value of 1.252 ns.

In 2013, Altera introduced a new feature for FPGA DSP Blocks with their Arria 10 FPGA [59]. They support three different modes of operation which are shown in Figure 2.6: In standard-precision mode, they allow two 19x18 bit integer multiplications followed by an addition and accumulation. In high-precision mode, they perform one 27x27 bit integer multiply-accumulate. In the new floating-point mode, they offer a single-precision floating-point multiply-accumulate or multiply-add.

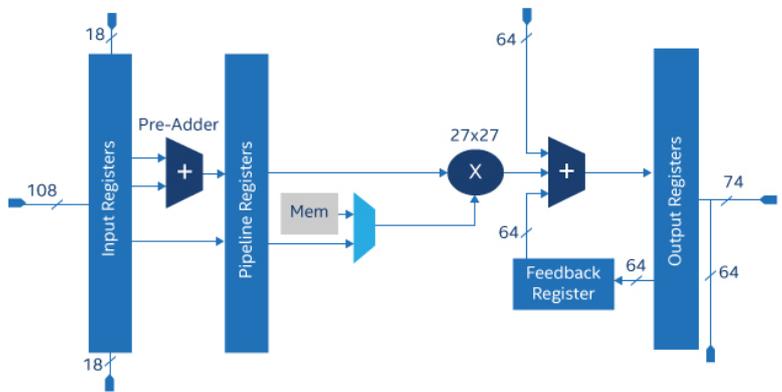
2.1.2.3 On Chip RAM

In addition to the registers inside the CLBs, most FPGAs come with on-chip SRAM. Functional blocks containing this RAM are often called RAM blocks, BlockRAM or BRAM. They often allow different operation at configurable bit widths. For example, a Xilinx 36 KBit BlockRAM can be configured to store 512 72-bit words or 2048 18-bit words, just to name two of the seven possible configurations. Furthermore, the BlockRAM contains two ports that can be configured independently. On-Chip BlockRAM allows much higher bandwidth and lower latency compared to off-chip RAM.

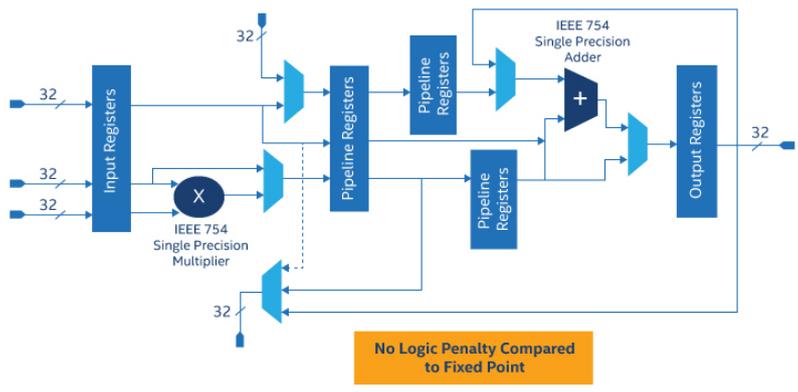
The newest generation of Xilinx FPGAs to this day, the Virtex UltraScale+ FPGAs [125], come with two new memory options. The new UltraRAM is comparable to BlockRAM, but larger, e.g., multiples of these 288KBit blocks can easily be connected together to create larger memory arrays. Furthermore, some UltraScale+ devices offer High Bandwidth Memory which consists of one or two HBM stacks adjacent to the FPGA die [81]. It is connected to the FPGA using dedicated memory controllers.



(a) Standard-precision



(b) High-precision mode



(c) Floating-point mode

Figure 2.6: Arria 10 DSP Block (image taken from [60])

2.1.2.4 *Interconnect*

As stated earlier, the CLBs and other functional blocks connect to each other through a programmable routing network or *interconnect*. This routing interconnect consists of wires and configurable switches. Figure 2.7 shows the (simplified) working principle of the interconnect. Switch Boxes (SBs) connect vertical and horizontal wires while the CLBs and other functional blocks are connected to the interconnect via connection Connection Boxes (CBs). This type of FPGA design is also called Island-Style, as the logic islands are surrounded by a sea of routing interconnect.

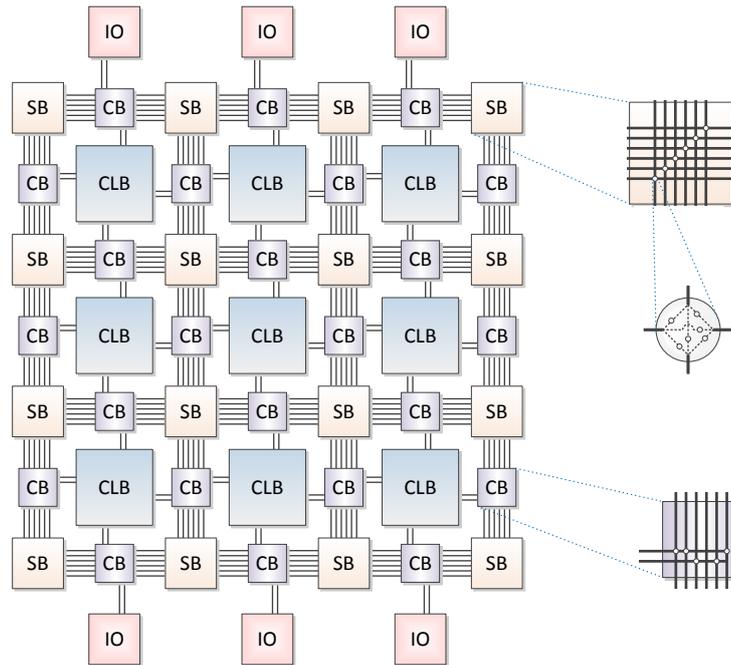


Figure 2.7: FPGA Interconnect

The view presented here is simplified. Modern FPGAs implement improved designs. For example, a percentage of the wires can be designed as long-lines that span multiple blocks. They require fewer switches for the same distance, thereby reducing routing area and delay.

2.1.2.5 *Synthesis flow*

In order to program an FPGA with a circuit defined in an HDL, multiple steps are required. These steps are executed by multiple vendor tools. Furthermore, third party tools exist in some cases. Figure 2.8 shows the typical tool flow. The starting point is the circuit to be implemented, defined in an HDL.

In the first step "logic synthesis", the HDL is optimized and converted into a net of boolean gates and Flip-Flops. Usually, technology-independent optimization techniques are applied in this step. However, as the input HDL and the output net list may contain technology-

dependent constructs like DSP-Blocks, this step cannot be considered fully technology-independent.

In the second step, the technology-independent nets are mapped into target-technology cells. In the case of FPGAs, the basic cells are N-input LUTs, flip-flops, and functional blocks as described before. Technology-dependent optimizations can be performed, targeting objectives like latency, area, or power.

The placement step assigns each of the cells in the mapper output to a resource on the FPGA chip. For each logic block on the FPGA the configuration is thus defined in this step. Three optimization goals are common: minimization of the required wiring (wire-length driven placement), balanced wiring density (routability-driven placement) and maximized circuit speed (timing-driven placement).

In the routing step, the placed blocks are interconnected as defined by the mapper. A state-of-the-art routing algorithm for FPGAs is the Pathfinder algorithm presented in [79]. Note that in some cases, the routing step may change the configuration of the logic blocks. For example, empty CLBs may be used as pass-through or the register placement can be changed if required.

Finally, the routed design is then converted to the format used to configure the FPGA. This step is called bit-stream generation.

As stated before, the starting point for the synthesis flow presented in this section is a HDL description. The manual creation of this HDL code is still a very time consuming and expensive step. Therefore, it is desirable to start with the synthesis flow at a "higher level". Section 3 will describe such HLS in detail.

2.1.3 Adaptive Computer Systems

As discussed in Section 1.1, reconfigurable devices such as FPGAs can be used as an accelerator for special functionalities. While most of the functionality remains on a Software Programmable Processor (SPP), time or energy critical *kernels* are extracted and translated to a reconfigurable device. Such a system composed of a microprocessor and a reconfigurable device is called Adaptive Computing Unit (ACS) [64].

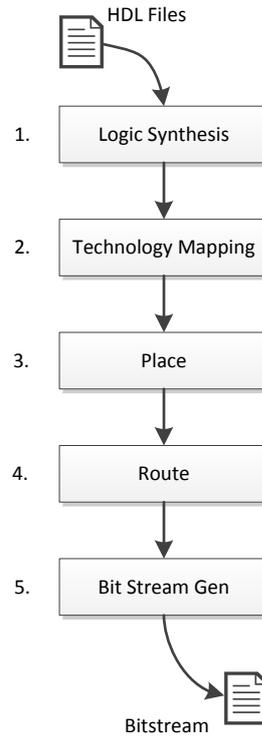


Figure 2.8: FPGA Synthesis Tool Flow

While the name ACS is common in high performance computing, in embedded domain the term reconfigurable computing platforms may be used [69]. However, both share the same base architecture and there is no clear boundary between the two.

Depending on the application and how its execution is divided between the microprocessor and the reconfigurable device, a high bandwidth and/or low latency connection between both processing elements may be required. Multiple architectures for this connection are found in literature.

STAND-ALONE In the most loosely coupled stand-alone architecture, the reconfigurable device is connected to the SPP via relatively slow peripheral interfaces (see Figure 2.9). Often, serial connections like Universal Serial Bus (USB) or COM ports are used to allow communication between the reconfigurable device and the software running on the SPP. In this architecture, the reconfigurable device has no direct access to the host computers address space, peripherals or memory. An example for such a connection is the HaLOEWEn platform [3, 91]. The FPGA is connected to the GPIO pins of the microprocessor, which allow only moderate data transfers.

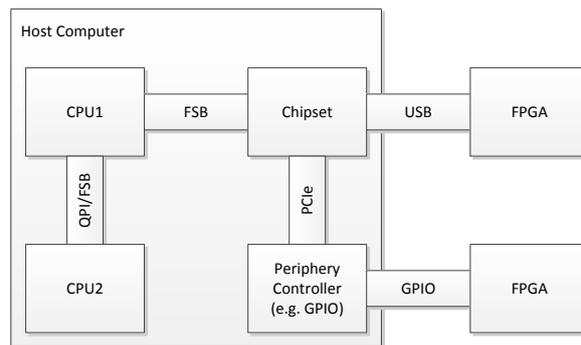


Figure 2.9: Stand-alone architecture

PERIPHERAL COMPONENT INTERCONNECT Modern FPGAs, especially high-end models, come with integrated support for Peripheral Component Interconnect (PCI) or Peripheral Component Interconnect Express (PCIe) [126]. Evaluation boards equipped with these FPGAs can often be plugged directly into a free PCIe Slot, e.g., the Virtex-7 VC707 Evaluation Kit [128] (see Figure 2.10). This way, the FPGA is mapped into the host computer's I/O port address space or memory-mapped address space. Furthermore, the FPGA also has access to the host address space, which allows accessing the main memory or other PCI(e) devices. Depending on the number of lanes and the PCIe revision, high bandwidth may be available, e.g., some Virtex-7 devices support an eight-lane PCIe 3.0 connection which theoretically delivers up to 7.875 GB/s.

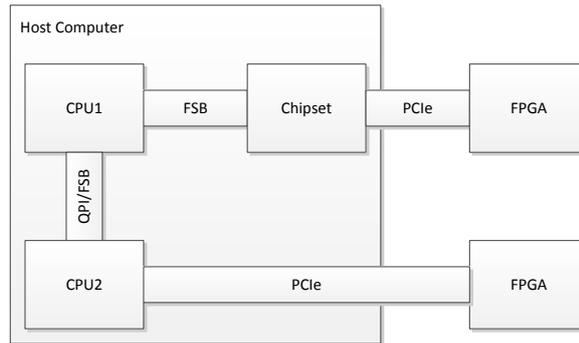


Figure 2.10: Coupling via Peripheral Component Interconnect

FRONT SIDE BUS AND QUICKPATH INTERCONNECT An even tighter coupling is accomplished by directly connecting the reconfigurable device to the front side bus of the SPP (see Figure 2.11). An example is the Convey HC-1(ex), which uses the second slot of a dual processor motherboard as an FPGA connector [14]. While one slot is fitted with a 2.13 GHz Intel Xeon host CPU, the other is connected to a board hosting four user-programmable Virtex-5 or Virtex-6 FPGAs. In addition to low latency data transfers, this design allows cache coherency between the CPU and the FPGA. More recent work targets Intel QuickPath Interconnect (QPI) [86], which is a point-to-point processor interconnect that replaced the Front-Side Bus (FSB) in Xeon (Nehalem) and afterwards [66].

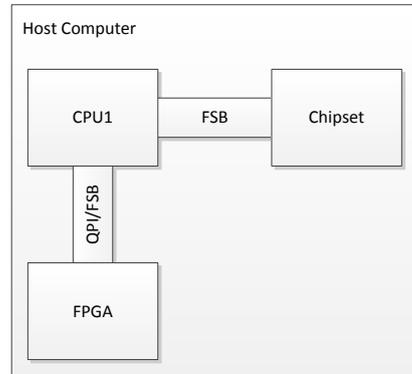


Figure 2.11: Coupling via Front Side Bus

SYSTEM ON A CHIP Figure 2.12 shows an architectures that combines the SPP and the reconfigurable device on a single chip. This architecture is similar to the front side bus based coupling described above, but the higher level of integration may lead to an improved latency and/or energy consumption. Intel and Xilinx offer SoC FPGAs, which integrate an ARM-based hard processor system (including cache, peripherals and memory interfaces) with an reconfigurable FPGA fabric [59, 129]. On the Xilinx Zynq, which is used as target platform in

this work, this architecture allows low latency accesses to the SPPs 2nd level cache. Some approaches even target an integration in the processor pipeline [104, 109] to further reduce latency. This way, reconfigurable device be can accessed by extending the SPPs instruction set.

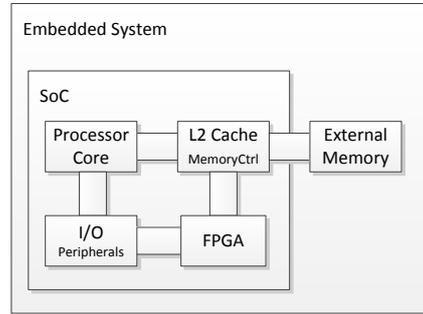


Figure 2.12: SPP and FPGA integrated on an SoC

For programming an adaptive computer system, the application must be *partitioned* in a hardware and a software part. Parts of the application that have high demands on computational power must be identified. Furthermore, the amount of communication between those compute kernels and the rest of the applications must be considered to not exceed the available bandwidth of the target ACS. Although approaches exist for automatic partitioning [42], most industrial and academic high-level synthesis tools delegate this task to the user.

2.2 IEEE 754 FLOATING-POINT DATA STANDARD

As this thesis examines the efficient implementation of floating-point applications reconfigurable computing platforms, a short recapitulation of the IEEE 754 floating-point format is given in this Section. Parts of this section have been published in [6, 7].

The IEEE 754-1985 standard defines the floating-point representations currently in widespread use. In 2008 it was superseded by IEEE 754-2008 [55]. A finite number R is represented in this format by the three components named mantissa (M), exponent (E) and sign (S), so that

$$R = M * 2^{E-b} * (-1)^S$$

where the bias b is a positive integer. The standard defines a number of basic formats. A format is defined by specifying bit widths of the M and E fields as well as defining the bias values. Table 2.1 lists some of the formats defined in IEEE 754-2008, including binary32 and binary64 which are more commonly known as single-precision and double-precision. Note that one bit of the mantissa is not stored, as explained below.

Table 2.1: Common formats defined in IEEE754-2008

Name	Mantissa bit width	Exponent bit width	Exponent Bias
binary16	10+1	5	15
binary32	23+1	8	127
binary64	52+1	11	1023
binary128	112+1	15	16383

Figure 2.13 shows the structure of the widely used double-precision format in memory. It is composed of the three parts mantissa, exponent and sign.

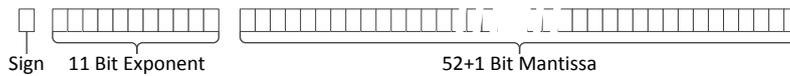


Figure 2.13: IEEE 754 Double-precision format

The formats specified by the standard also ensure unique representations of each number, thus avoiding the ambiguity arising, e.g., from $1.5 * 2^3 = 0.75 * 2^4$. This is achieved by scaling the mantissa s.t. its most-significant **1** bit actually becomes the most significant bit (msb) of the M field in the standardized binary representation. Since this leads to all numbers (with the exception of Zero) having an M field beginning with a **1** bit, the leading bit is no longer explicitly stored (implied **1**).

This scaling process is called *normalization*. It has to be performed after every computation for the result to be in valid IEEE 754 number representation and usually includes rounding the mantissa. For rounding, the standard defines five modes, which differ in the way how ties are handled.

For numbers that cannot be represented by the formula $R = M * 2^{E-b} * (-1)^S$, the following interpretations are defined:

NULL A null is stored by setting all bits of the mantissa and exponent to zero. This is an exception because the mantissa starts with an implicit leading one.

SUBNORMALS A further exception are numbers with a very small magnitude having zero as exponent but a non-zero mantissa. These so-called *subnormals* do also not have an implied **1** as msb in mantissa. This allows numbers closer to zero to be represented because the mantissa is allowed to take a value smaller than 1.0, which is obviously impossible for a mantissa with an implicit leading one.

INFINITY In case of an overflow or a division by zero, the result should be positive or negative infinity. Infinity is encoded by setting all

exponent bits to one and all mantissa bits to zero. The sign is used to indicate a positive or negative infinity.

NOT A NUMBER NaN is used to indicate an invalid value, e.g. the result of a negative number's square root. It is stored by setting all exponent bits to one and at least one mantissa bit to non-zero. The standard defines quiet and signalling NaNs, which differ in the highest mantissa bit being 1 for quiet and 0 for signalling NaNs.

A more detailed survey of the fundamentals of floating-point operations on FPGAs is given in [49].

2.2.1 *Fused operations*

Scaling (especially rounding) and the encoding of non-normal values may require a serious amount of time. Each computation is followed by a three steps: First, the normalization scales the result to a mantissa to the required form ("1.xxxx"). Afterwards, IEEE 754 conformant rounding must be performed to compute a mantissa in the required bit size (e.g. 52+1 bits for double-precision). Furthermore, a post normalization step follows. It checks if an overflow occurred during rounding and adjusts the scaling again in such case. Finally, the exceptional number representations discussed above must be encoded after each step (and decoded before the next operation).

For high-performance computation, it can be worthwhile to avoid normalization between two operations. This allows the computations to be *fused* together and performing the normalization only at the end of the fused region (see Figure 2.14). However, this action violates the IEEE 754 standard and the result will generally differ from an standard-conform implementation.

This technique has often been used to increase the performance of multiply-add operations, combining them into FMA operations. In this manner, the steps "normalization", "rounding", "post-normalization" and in some cases also "denormalization," can be avoided. Inside of these fused operators, non-standard floating-point formats can be used, generally allowing improved area / latency tradeoffs and a better match to the target technology of the specific implementation. Furthermore, if required, the intermediate results can also be represented in formats providing greater accuracy than the standard formats.

2.2.2 *Faithful Rounding*

Another way to further reduce latency and area consumption of floating-point units is to perform Faithful Rounding (FR) instead of IEEE 754 Conforming Rounding (CR). The latter requires the selection of the single representable value closest to the infinitely accurate result, while

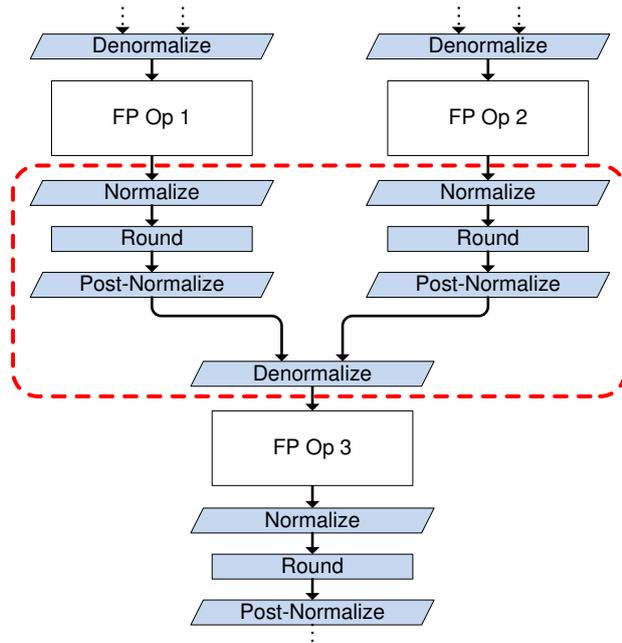


Figure 2.14: Normalization and denormalization between floating-point (FP) operations

the former can arbitrarily return any one of the pair of closest representable values bracketing the accurate result.

In literature the error is often expressed as a multiple of a Unit of Least Precision (ULP), i.e., the value that the least significant digit represents if it is 1. For CR rounding, the maximum error is 0.5 ULP, while FR has a maximum error of up to 1.0 ULP (or 1-ULP accuracy). Figure 2.15 shows a rounding example in which a 31 bit number is rounded to fit into the 23+1 mantissa bits used in the single precision floating-point format. CR requires increasing the unit of least precision by one. The error made is equal to 10111b / 10000000b = 0.18d ULP. FR alternatively allows a truncation which causes an error of 1101001b / 10000000b = 0.82d ULP.

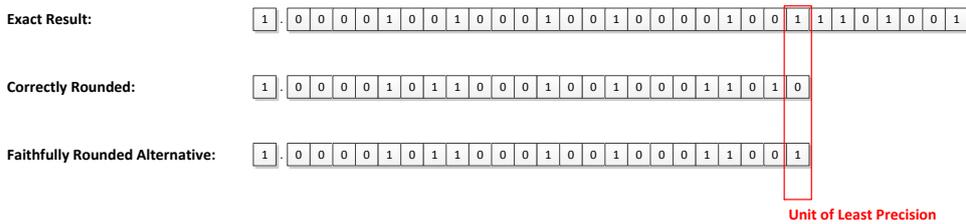


Figure 2.15: Faithful rounding of a floating-point mantissa

The improved accuracy of CR comes at a significantly increased hardware cost which may not be restricted to the rounding logic itself. For example, in multiplicative division methods it is often not possible to directly compute a CR result. Instead, an FR result is returned and

resource and latency intense post-processing is required to actually perform CR.

Note that while the *maximum error* doubles when FR allowed, the same does not automatically apply for the *average error*. For example, the division units presented in this work contain multiple error sources. In worst case these errors are additive, but in other cases they may cancel each other out. Therefore, the error is not evenly distributed over the allowed range of 1 ULP.

2.3 CONVEX OPTIMIZATION

This section gives an overview of convex optimization and the CVXGEN solver generator. It was previously published in a similar form in [8].

Convex optimization is a subclass of mathematical optimization which in general may be defined as follows:

Given an objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, m constraint functions $g_i : \mathbb{R}^n \rightarrow \mathbb{R}, i = 1, \dots, m$ and m constraints $b_1, \dots, b_m \in \mathbb{R}$:

$$\begin{aligned} & \mathbf{minimize} && f(x) \\ & \mathbf{subject\ to} && g_i(x) \leq b_i, i = 1, \dots, m \end{aligned}$$

The vector x is also called the optimization variable in this case. A function $h : \mathbb{R}^n \rightarrow \mathbb{R}$ is called a *convex* function if it satisfies the inequality $f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$ for all $x, y \in \mathbb{R}^n$ and all $\alpha, \beta \in \mathbb{R}_0^+$ with $\alpha + \beta = 1$ [17].

A (mathematical) optimization problem is called a convex optimization problem (or convex minimization) if the objective function f and all constraint functions $g_i, i = 1, \dots, m$ are convex functions. A convex optimization problem is defined as follows: given a real vector space \mathcal{X} and a convex function $f : \mathcal{X} \rightarrow \mathbb{R}$ defined on a convex subset \mathcal{X} of X . The problem is to find any point x^* in \mathcal{X} such that $f(x^*) \leq f(x)$ for all $x \in \mathcal{X}$.

Many optimization problems can be transformed to a convex optimization problem. E.g., maximizing a concave function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is equivalent to the convex minimization of $-f$.

The convexity property makes such problems easier to solve than the general case [17], and allows the creation of efficient automatic solvers [45, 76]. CVXGEN, which targets problems that can be modeled as convex quadratic programs, is one such tool which has shown superior performance over competing approaches [78].

In contrast to familiar solvers, e.g., from the (I)LP domain, CVXGEN does not solve a specific problem instance. Instead, it *generates* a solver for the problem (formulated in an abstract DSL), but with the parameters (e.g., the b_i above) still remaining variable. The actual values for the parameters are set only at run-time of the solver code.

```

...
work.v[268] = target[31]-work.L[249]*work.v[269];
work.v[269] = target[104];
work.v[270] = target[105]-work.L[254]*work.v[269];
work.v[271] = target[106]-work.L[255]*work.v[269]-work.v[270];
work.v[272] = target[107]-work.L[257]*work.v[269]-work.v[271];
work.v[273] = target[108]-work.L[260]*work.v[269]-work.v[272];
work.v[274] = target[32]-work.L[264]*work.v[118]-work.v[273];
work.v[275] = target[33]-work.L[274]*work.v[122]-work.v[274];
work.v[276] = target[346];
work.v[277] = target[347];
...

```

Figure 2.16: Typical excerpt from the generated solver code

With the problem instance now fully described, the solver can compute the optimization variable.

The generated C code for the solver consists of many floating-point operations, is almost branch-free, and does not make any library calls. It is thus suitable for stand-alone execution in embedded systems lacking complete library support. The solver code relies on three key data structures: the values of the input parameters, the optimization variables, and a work area for temporary intermediate values. They are realized as C structures that contain several arrays of double-precision values. In almost all cases, data is addressed directly with only very few pointers being used. These properties also make the code attractive for compilation to reconfigurable hardware. Figure 2.16 shows a typical part of the solver code.

The solver computations have a high degree of instruction-level parallelism, but have also long chains of data-dependent operations. A simplified example of such data dependencies is shown in Listing 2.1).

```

x[1] = a*b + c*d;
x[2] = e*f + g*x[1];
x[3] = h*i + k*x[2];

```

Listing 2.1: Solver computation structure (simplified)

These dependency chains form a (potentially long) path through the algorithm’s data flow graph, shown in Figure 2.17 for the previous example, with the critical path marked by bold red edges. Besides chains of multiply-adds, the code also contains many floating-point divisions, which in some cases also form long dependency chains together with other operations.

Section 7 focuses on the hardware acceleration for convex solvers using many of the techniques presented in this thesis.

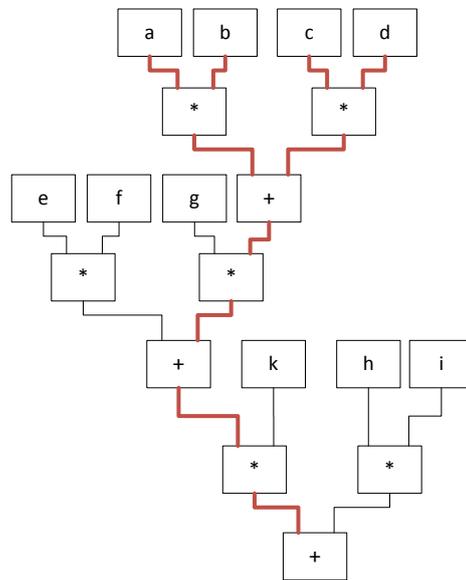


Figure 2.17: Critical path of code in Listing 2.1

HIGH LEVEL SYNTHESIS FOR APPLICATION-SPECIFIC COMPUTING

In Section 2.1.2.5, the synthesis flow from a hardware description in a HDL to a configuration file was presented. As already stated, writing such HDL code manually is a time and cost intensive task. Therefore, the synthesis flow can be extended to a higher level. The high-level synthesis starts with a higher-level description of the desired system behaviour, and *automatically* generates the HDL.

3.1 WHY USE HIGH-LEVEL SYNTHESIS

There are four major reasons for the use of HLS in the design process:

REDUCING MANUAL WORK REDUCES THE NUMBER OF ERRORS. Manual process steps usually involve the risk of human error. Errors discovered in the development cycle are already costly due to the nature of hardware debugging. Checking hardware in a simulation environment requires specialists, and is very time intensive, especially for large hardware designs. But errors not discovered before production is even more problematic. In ASIC-design, an error in the final product are usually irreversible, which means that the chip (or whole product) must be refabricated. In FPGA-designs, the error could be corrected in the field, but if the product is not shipped with an update function, the correction may be very expensive as well.

DESIGN SPACE EXPLORATION GETS MUCH EASIER. In many cases, the designer has a choice between a faster and bigger, or a slower and smaller solution. Designs written in a HDL may require at least a partial rewrite if it turns out that they do not reach the desired performance goals. Good HLS tools can create multiple designs from the same input code, providing different compromises between size, performance and power.

ALLOW NON-EXPERTS TO WORK IN HARDWARE DESIGN. While writing HDL requires deep knowledge of the hardware, the high-level description is meant to be easier to work with. For example, in many cases, the high-level description is given in (a subset of) C, which is already known to many software developers. Understanding a high-level description should be much easier, even for hardware experts. In some cases, even more abstract descriptions are possible using *domain-specific languages* (DSLs).

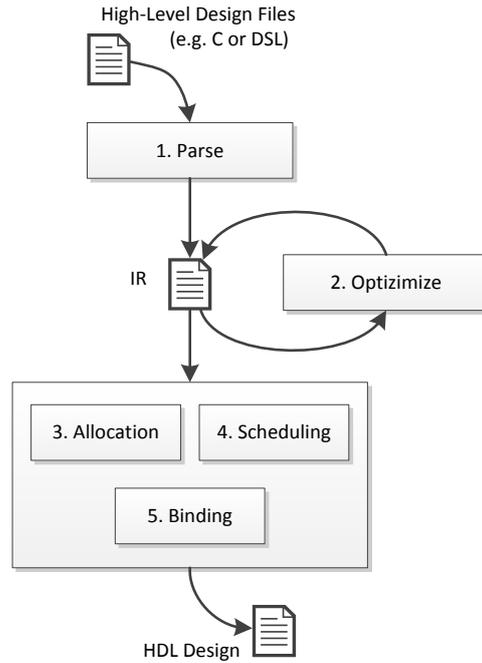


Figure 3.1: Steps of High-Level Synthesis Flow

SHORTENING THE DESIGN CYCLE reduces time to market and development costs. As the automated tool usually operates much faster than a human hardware designer, it should provide results much earlier. This will usually lead to an earlier product release. Thus, the company has a better chance to hit the market window. Not only is the duration of the design cycle reduced, but also total personnel effort. Because chip design is often a major part of development costs, further automation of this process can have a significant effect on the total development costs.

3.2 WORKING PRINCIPLE

The goal of HLS is to convert a high-level description of a design to a Register-Transfer Level (RTL) netlist described in a HDL. The RTL netlist describes the design as a set of hardware registers connected by logical operations. Figure 3.1 shows the typical steps in high-level synthesis.

In a first step, the input file(s) are parsed and converted into an Internal Representation (IR). This step is the same for hardware generation as for software compilation. Therefore, many HLS tools use the front-end and IR of software compilers like GCC [103] or LLVM [74] for this task.

In the second step, the IR can be subject to several optimization tasks. They are working directly on the IR. Often, optimizations of software compilers can be useful for hardware generation, as well. For example, any step that removes unnecessary memory accesses will re-

duce the required memory bandwidth for both hardware and software solutions. In [53], many LLVM optimization passes are evaluated regarding their benefits for HLS.

Afterwards, the IR or a Control Data Flow Graph (CDFG) constructed from the IR is fed into the three major steps of high-level synthesis, which are allocation, scheduling and binding. Some HLS tools perform some of these task together or even iteratively to find an optimal result. In [53], many LLVM optimization passes are evaluated regarding their benefits for HLS.

Figure 3.2 shows the construction and optimization of the IR and the CDFG using a small example of C code. During the optimization (-O1 is used), many load and store operations are removed and the data is stored in registers instead.

3.2.1 Allocation

Allocation defines which and how many hardware resources are available for use, often in respect to user constraints. Such constraints can include an upper limit for the number of instances of functional unit (e.g., the maximum number multipliers), or a clock frequency that the generated hardware should achieve.

For each operation in the IR, multiple hardware implementations (or functional units) may exist in the tools library, each targeting different area/latency/throughput/power trade-offs. Such a library may contain technology-dependent and technology-independent hardware implementations.

Some HLS tools may perform *additional* allocation during scheduling and binding tasks [30].

3.2.2 Scheduling

In scheduling, each operation in the program is assigned to a particular clock cycle. The assignment must not violate the dependencies of the CDFG, e.g., for the computation $A * B + C$, the addition may not be scheduled to a cycle in which the multiplication has not finished. Furthermore, if hardware resources are limited, there may not be more operations executed in parallel than allowed. Scheduling algorithms can be classified into "time constrained" and "resource constrained" methods. In time constrained scheduling, the number of FUs is minimized for a fixed number of clock cycles, while resource constrained scheduling minimizes the number of cycles for a fixed resource limit (e.g. number of FUs).

```

if (x < 10)
    x++;
else
    x = y / 2;
x = x % 77;

%0 = load i32* @x, align 4
%cmp = icmp slt i32 %0, 10
br i1 %cmp, label %if.then, label %if.else

if.then:
%1 = load i32* @x, align 4
%inc = add nsw i32 %1, 1
store i32 %inc, i32* @x, align 4
br label %if.end

if.else:
%2 = load i32* @y, align 4
%div = sdiv i32 %2, 2
store i32 %div, i32* @x, align 4
br label %if.end

if.end:
%3 = load i32* @x, align 4
%rem = srem i32 %3, 77
store i32 %rem, i32* @x, align 4
    
```

a) Input code in C

b) LLVM IR

```

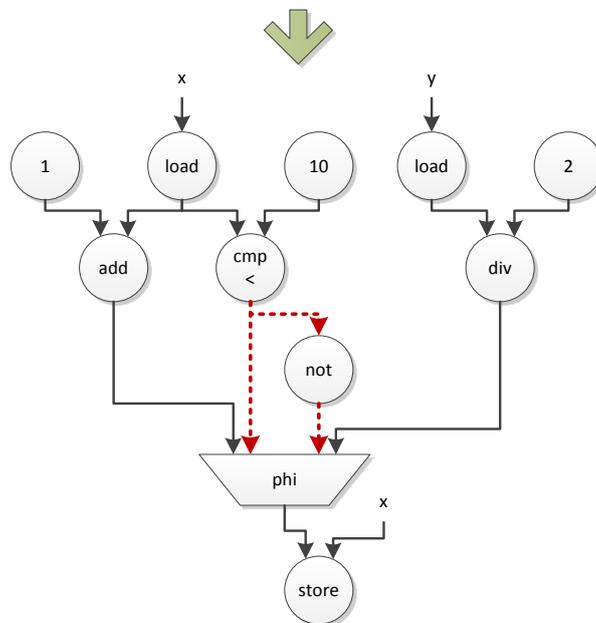
%0 = load i32* @x, align 4
%cmp = icmp slt i32 %0, 10
br i1 %cmp, label %if.then, label %if.else

if.then:
%inc = add nsw i32 %0, 1
br label %if.end

if.else:
%1 = load i32* @y, align 4
%div = sdiv i32 %1, 2
br label %if.end

if.end:
%storemerge = phi i32 [ %div, %if.else ], [ %inc, %if.then ]
%rem = srem i32 %storemerge, 77
store i32 %rem, i32* @x, align 4
    
```

c) Optimized LLVM IR (using -O1)



d) Resulting control data flow graph (control edges shown in red)

Figure 3.2: Example for the construction and optimization of LLVM IR and CDFG

3.2.3 Binding

Binding involves two steps: First, each operation in the program being synthesized must be assigned to a *specific* functional unit. Second, each program variable must be assigned to a register. When a hardware unit is used for multiple operations, it is called a *shared* resource. The same applies to registers that are reused for multiple variables. Register and functional unit binding directly affects connectivity binding, as connections for data transfers are required from component to component, such as a bus or a multiplexer [30].

Sharing aims to reduce the resource/area requirement of the circuit. However, in both cases, multiplexers are usually required to provide the correct input value to the shared resource. These multiplexers may consume a considerable amount of area, especially on FPGAs.

3.2.4 Loop Pipelining

Unrolling loops allows to increase the degree of parallel execution, as long as sufficient functional units can be allocated. However, even non-unrolled loops may allow some kind of parallelism. Similar to a processor pipeline, loop pipelining can start the *next* iteration of a loop before the *previous iteration* is finished. This way, the execution of the iterations will partially overlap.

Figure 3.3a shows the execution of a small loop (see Listing 3.1) without loop pipelining. Each iteration starts after the previous has iteration been finished, so there is no parallel execution. If loop pipelining is enabled, the next iteration starts as soon as possible. Ideally, a new iteration starts with every clock cycle as shown in Figure 3.3b. In this case the, Initiation Interval (II) is one cycle.

Listing 3.1: Small example loop used in Figure 3.3

```
for (i=0; i<N; i++)
  a[i] = b[i]+3;
```

Pipelining is not always possible. Dependencies between the iterations may prevent the HLS tool from generating a pipelined architecture. This includes data dependencies, memory dependencies, and control dependencies. Furthermore, limited hardware resources may prevent pipelining from happening, e.g. if a loop contains three additions but only one adder unit may be used.

3.3 OPEN PROBLEMS

While HLS offers major advantages, there is also a set of open problems which is subject for ongoing research.

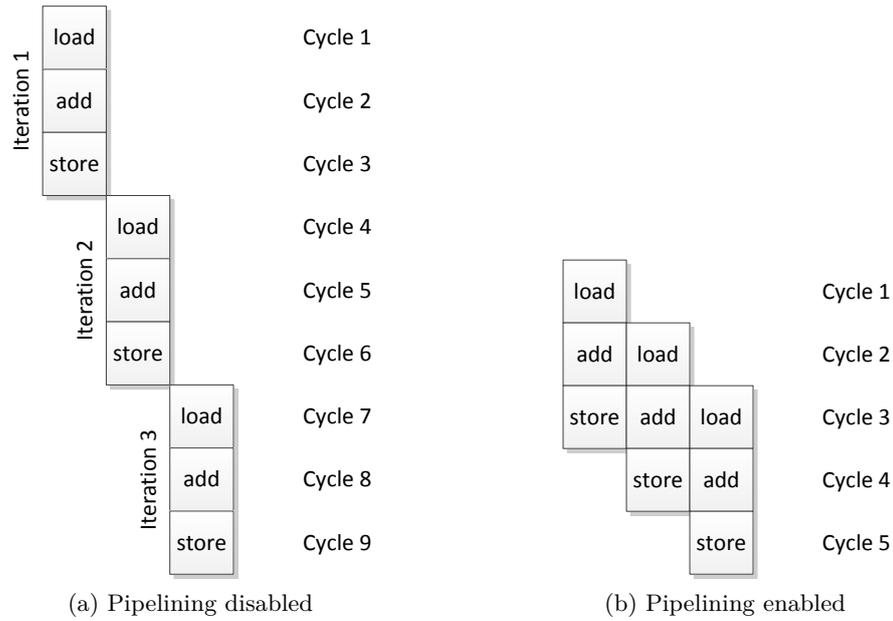


Figure 3.3: Loop execution with and without pipelining

First, most HLS-tools are still not usable for non-experts as stated above, because they do not address the complete hardware/software design problem. They focus on the synthesis of a C-function to hardware. But this C-function is often only a small (but compute-intensive) part of a larger program. When the synthesis is done, the user is left alone with the initial C program, and a piece of hardware description, also called the hardware kernel. But without deep knowledge of hardware design, it will be very difficult for a user to create an efficient interface between the hardware kernel and the software program. Thus the goal of allowing non-experts to work in hardware design is not reached. Therefore, hardware-software *co-compilers* are required, which automatically provide the required glue logic for the hardware-software interaction. Nymble [4] is one such co-compiler.

Furthermore, the handling of floating-point computation is very limited. A survey from 2016 lists 33 industrial and academic HLS tools, of which only 13 can handle floating-point computations [85]. Looking closer at the results, it turns out that some of these compilers only allow floating-point to fixed-point conversion, e.g., AccelDSP [115]. Others depend on floating-point units manually integrated into the design flow by the user, e.g., ROCCC [113]. But even the remaining ones often rely on more or less strict IEEE 754 conforming computations, whereas relaxing this conformity can bring significant benefit, as shown for the fused data path [72]. Automatic conversion from floating-point to fixed-point arithmetic may be a suitable approach for some applications, if the large dynamic range of floating-point format is not required. However, such conversion must be sensitive to input data, and the requirements

of the application, to determine the optimal fixed-point format for each variable in the program.

Finally, the experiments made in this work have discovered big problems most existing tools have with the synthesis of *large* code blocks. Even leading industrial tools have been observed to produce inefficient or non-synthesizable hardware when it comes to high degree of resource sharing. As resource sharing on FPGA may be counterproductive in some cases, some tool designers argue completely against sharing of integer additions or registers [23]. However, for programs consisting of ten-thousands of such additions and registers, the fully spatial approach is not an efficient solution.

PRIOR AND RELATED WORK

This chapter discusses previous work and approaches. In the opening sections, state-of-the-art HLS tools and approaches for efficient resource sharing will be discussed. Later sections will include a recapitulation on related work on division and FMA computation is recapitulated. Parts of this chapter have been published before in [6–8].

4.1 HIGH LEVEL SYNTHESIS FROM C

An increasing number of HLS tools, originating both from industry and academia, is capable of translating (often differing) subsets of C into synthesizable RTL HDL code.

Commercial tools include Mentor Catapult[40], Xilinx Vivado HLS [118] and Synopsis’ Symphony C Compiler [106]. All of these tools have in common that they provide only hardware synthesis. They do not consider hardware/software-co-execution and interfacing in a heterogeneous system such as an ACS. Hardware/software co-synthesis is supported in academic tools, e.g., in Nymble [4], ROCCC [113], COMRADE [41], and Garp CC [22]. LegUp [23] also supports hardware/software co-synthesis and has recently changed from an academic to an industrial tool.

Garp CC [22] is based on the SUIF compiler infrastructure [13] and uses a modified GCC tool chain. In addition to high level synthesis, it offers automatic partitioning to execute a C program on a MIPS processor augmented with a Coarse-Grained Reconfigurable Array (CGRA). Such CGRAs consist of a large number of functional units and register files which are interconnected by a mesh style network. In theory they are in some cases more efficient, but less flexible than FPGAs. However, CGRAs are currently not available for sale and are often emulated on FPGAs.

COMRADE, based on SUIF2 [68], focuses on the compilation of control-intensive C code into dynamically scheduled accelerators [41]. This means the hardware functional units do not start at a predefined cycle or stage as it is the case with static scheduling (as described in Section 3.2). Instead, the computation of each unit starts whenever all inputs and dependencies are ready. The proposed compute model includes very finely granular hardware/software partitioning with the proposed architecture allowing master-mode accesses to memory shared between the reconfigurable compute unit (RCU) and the software-programmable processor (SPP). However, resource shar-

ing and floating-point arithmetic are not supported COMRADE and neither COMRADE nor SUIF2 are currently developed any further.

Nymble [4] extends many concepts of COMRADE, e.g., it continues to offer finely granular hardware/software partitioning. It uses the LLVM compiler framework [74] as front-end and for target-independent optimization. The generated hardware kernels and software parts execute together in a shared memory architecture in the same address space (allowing transparent passing of pointers between software and hardware). Initially, the MARC II configurable memory system [70] is used to perform multiple concurrent read/write accesses. In contrast to COMRADE, Nymble creates pipelined statically scheduled accelerators, moving the focus away from control-intensive applications to data-driven computation.

The C to hardware compilers ROCCC and LegUp are also based on the LLVM compiler framework. While ROCCC lacks support for many commonly used C constructs, e.g., pointers and variable-distance shifts, LegUp supports a large subset of C code. ROCCC can be seen as a specialized tool, providing good results on a smaller subset, while LegUp does well even in the general case.

A limitation that LegUp shares with many other compilers is the partitioning granularity: only complete functions are translated into hardware. Nymble is more selective. It can translate just part of function to hardware and even exclude sections within that area, moving them back to software, as required.

4.2 RESOURCE SHARING

Ideally, reconfigurable computing is performed in a fully spatial model, associating an individual hardware operator with each operation in the algorithm (input program). This approach allows hardware-accelerators to often exceed the performance of SPPs which reuse (time-multiplex) the same limited number of compute elements for all operations.

However, the fully spatial approach quickly becomes infeasible on current-generation reconfigurable devices when floating-point operations are considered. On most FPGAs, appropriate hardware operators have to be composed from many low-level primitives, requiring significantly more chip area than integer operators. In practice, large numbers of such operators cannot be implemented with high throughput and low latency.

Thus, even for reconfigurable computing, resource sharing must be considered for costly floating-point operators. However, a number of aspects closely tied to the underlying FPGA architecture have to be considered.

First, the fully spatial approach allows the use of operators specialized for each operation (e.g., in terms of bit widths, number formats, or constant inputs). When attempting to reuse operators, this specializa-

tion can often only be performed in a more limited fashion and needs special care. One technique extracts recurring operation patterns from the original CDFG to execute on the same operator and accepts small variations in bit widths for a greater degree of reuse [27, 28]. However, the shared operator will then have the maximum of the bit widths required by the operations and may end up being slower than specialized operators.

Secondly, reusing operators requires wide multiplexers (64b for double-precision data) on their inputs to connect them to the different data sources. Such multiplexers require significant area and delay, depending not only on the FPGA architecture but also the interface of the shared operator itself [46]. In [26], an integer linear programming based approach is shown to reduce the number of multiplexers. However, it is stated that a solution may not be reached within a reasonable time for larger problems.

Thirdly, the multiplexers must be controlled over time, establishing the connections required by the execution schedule. In the fully spatial paradigm, a commonly used controller architecture uses a Petri net-like N -hot approach, where the activation state of each operator is controlled by an associated flip-flop and multiple of these flip-flops may be active in parallel (easily supporting pipelined execution).

However, when attempting to do aggressive resource sharing, this established approach can be inferior to the use of microcoded controllers. Experiments presented in this thesis show that the N -hot approach for large loop bodies can cause resource demands that exceed the capabilities of the target FPGAs. Microcode has often been used to drive hardwired compute elements, but mainly in HLS for ASICs and ASIPs [16]. In Section 5.3, a refined scheme better suited for mapping to FPGAs is presented.

4.3 FLOATING-POINT UNITS FOR FPGAS

In this work, multiple floating-point units are presented, all specially designed and optimized for FPGAs. Two dividers and two FMA units are presented as well as a floor and ceiling unit. While for the latter no FPGA-specific work was found in literature, divisions and multiply-add units have been subject to intense research in the past.

4.3.1 *Algorithms for Digital Division*

Methods for (floating-point) division can be divided into three categories: digit recurrence (or subtractive), lookup table based, and multiplicative (or iterative) methods.

Digit recurrence algorithms compute one digit of the result per clock cycle and thus achieve linear convergence. For higher performance a radix higher than two can be chosen, allowing to compute more than

a single bit of the result per cycle at the cost of increased area requirements. One example for a digit recurrence algorithms is the SRT algorithm [65].

Lookup table based methods are used to compute the reciprocal of the divisor and use a later multiplication to actually compute the quotient. For small word widths it is possible to use the entire divisor as address for a single table lookup. However, the exponential growth of the table size prevents larger lookups. For larger word widths, bipartite tables [101], or piecewise polynomial approximations can be used. In piecewise polynomial approximations, the reciprocal function is divided into 2^m equally sized partitions. For each partition a polynomial approximation is performed, with the polynomial coefficients required for each partition being stored in a lookup table with m entries. The m most significant bits of the divisor are then used as address to retrieve the coefficients. Afterwards, a *polynomial approximation* of the division can be computed using only multiplications and additions.

Two examples for popular **multiplicative methods** are the Newton-Raphson [132] and the Goldschmidt [43] algorithms. While Newton-Raphson computes the reciprocal of the divisor, again requiring a final multiplication, Goldschmidt performs a direct computation of the quotient. Both methods offer quadratic convergence, making them especially attractive for computations that require a high accuracy (e.g., double-precision), but usually requiring multiple iterations to reach the desired accuracy. This number of iterations can be reduced significantly by providing a good initial approximation which can be computed using another division method, e.g., a lookup table based method. Using Goldschmidt's approach for the computation of $Q = Y/X$, a sequence is computed such that

$$\frac{A_{i+1}}{B_{i+1}} = \frac{A_i * Z_i}{B_i * Z_i}$$

with $A_0 = Y, B_0 = X$ and $Z_i = 2 - B_i$. In this sequence, A_i converges to Q .

4.3.2 FPGA Divider Implementations

A number of division methods have been adapted for FPGA implementation of double-precision division in the past. The FloPoCo floating-point library uses a Radix-4 digit recurrence method [32]. As in most digit recurrence methods, it allows the computation of the CR result. However, performance for larger word widths is limited due to the linear convergence of digit recurrence methods.

The VFLOAT library implements a lookup table based Taylor approximation to compute the reciprocal of the divisor [38], which is then multiplied with the dividend. The result may differ from the CR result computed by IEEE 754 conforming dividers (such as a Xilinx

IP core), which led the authors to classify their unit as 1-ULP accurate. VFLOAT division was shown to be faster than the corresponding Xilinx IP core.

An even faster FPGA division unit was presented by Pasca [90]. It uses a piecewise polynomial approximation of second degree as seed value for a single Newton-Raphson iteration which allows the complete double-precision divider to compute a FR result. Possible extensions to compute the correctly rounded result are discussed at the example of a single precision division unit. It offers superior performance when compared to other state-of-the-art FPGA division units. However, as the Newton-Raphson method only computes the reciprocal, an additional multiplication must be performed afterwards. In contrast, the Goldschmidt method allows this multiplication to be performed in parallel, and thus could further reduce the overall latency.

In [47] and [92], two multiplicative division methods for double-precision mantissa division are implemented on *ASICs*. While [47] proposes a single table look-up followed by three Goldschmidt iterations, [92] starts with an polynomial approximation and computes the final result using a single Goldschmidt iteration. In both approaches, a modified version of the Goldschmidt [77] algorithm is used.

The difference between the traditional Goldschmidt method and the modified version is shown in Table 4.1. The computation of A_i leads to the same result in both versions. However, with growing i , the required word width of R_i is less in the modified version than in the original. Furthermore, the squaring operation performed for R_i requires fewer resources than a full multiplication.

Table 4.1: Traditional vs. modified Goldschmidt method

method	iteration step	initial value
Traditional	$A_{i+1} = A_i * (2 - B_i)$	$A_0 = Y * \text{reciprocal}[X]$
Goldschmidt	$B_{i+1} = B_i * (2 - B_i)$	$B_0 = X * \text{reciprocal}[X]$
Modified	$A_{i+1} = A_i * (1 + R_i)$	$A_0 = Y * \text{reciprocal}[X]$
Goldschmidt	$R_{i+1} = R_i^2$	$R_0 = 1 - X * \text{reciprocal}[X]$

Both approaches target low latency division and compute a faithful rounded result. In this thesis, the two approaches are compared and optimized for use on recent FPGAs.

4.3.3 Tiling and Truncated Multiplication

To reduce the size of the division unit's internal wide fixed-point multipliers, the use of *truncated* multipliers is suggested in [102]. The truncation profits from the fact that in fixed-point multiplication and floating-point mantissa multiplication, the result is often rounded to avoid a

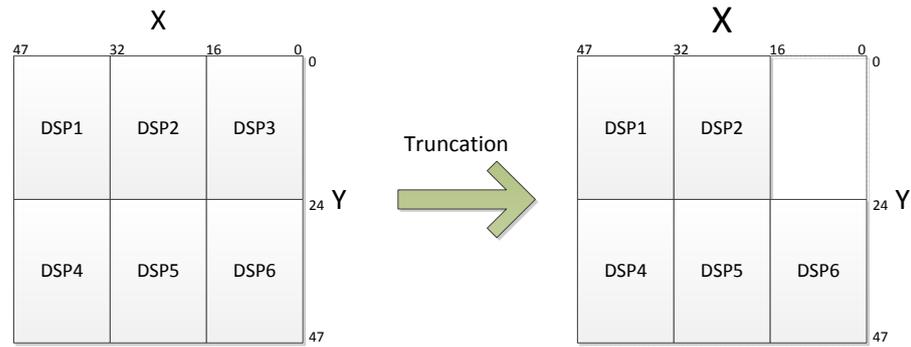


Figure 4.1: DSP tiling and truncation of a 48 bit multiplier

growth in word length. A truncation of the least significant bits during the multiplication is proposed which reduces the hardware cost by 25...35% introducing only a small computation inaccuracy. To partially compensate for this, and limit the maximum error, a correction-constant is added to the final result.

On FPGAs, multiplications larger than those natively supported by the FPGA DSP blocks are composed of multiple DSP blocks (which is called tiling). The multiplication is thereby divided in multiple smaller multiplications which in the device's DSP block. The results of the smaller multiplications are then added to compute the final product. For example the 48 bit multiplication $X * Y$ can be computed in 6 parts as shown in the following equation:

$$\begin{aligned}
 X * Y = & X[16 : 0] * Y[23 : 0] & + X[32 : 17] * Y[23 : 0] * 2^{17} \\
 & + X[4 : 33] * Y[23 : 0] * 2^{33} & + X[16 : 0] * Y[47 : 24] * 2^{24} \\
 & + X[32 : 17] * Y[47 : 24] * 2^{24+17} & + X[4 : 33] * Y[47 : 24] * 2^{24+33}
 \end{aligned}$$

In [15], Banescu adapted the concept of truncated multiplication for DSP tiling on FPGAs. Instead of truncating all bits in the adder tree less significant than position x , Banescu truncated/omitted complete DSP blocks, or replaced them by smaller LUT-based multipliers with reduced input word width. Figure 4.1 shows such a truncation for a 48x48 bit multiplication which is composed of multiple 16x24 bit DSP blocks. DSP3, which computes the least significant bits of the result, is truncated completely. Assuming that only the upper 48 bits of the result are used, the truncation error may be acceptable, as it is small compared to the error made when dropping the lower 48 bits of the 96 bit result.

Truncated FPGA multipliers were also used in the division unit presented in [90] as well as in the division units presented in this thesis.

4.3.4 Carry Save Adder

Carry Save Adders (CSAs) have long been used for fast constant-time addition [89], especially inside multiplication units. They are used to compute the sum of three or more n -bit numbers in binary. However,

they differ from other adders in that they output two n-bit numbers as result, e.g., a 3-to-2 CSA computes $A + B + C = D + E$. To compute the final result in binary format, the two numbers must be added by traditional adder.

The representation of a value as a sum of two binary encoded numbers is called Carry Save (CS) format in this work. It departs from conventional binary format by allowing the values **0**, **1**, **2** for each digit. While the CS format allows fast addition, it has to deal with non-unique representations for numbers, complicating, e.g., comparison operations. Please see Section 6.1.5 for a discussion of some of these details. Figure 4.2 shows multiple different representations for the same value in binary and CS format.

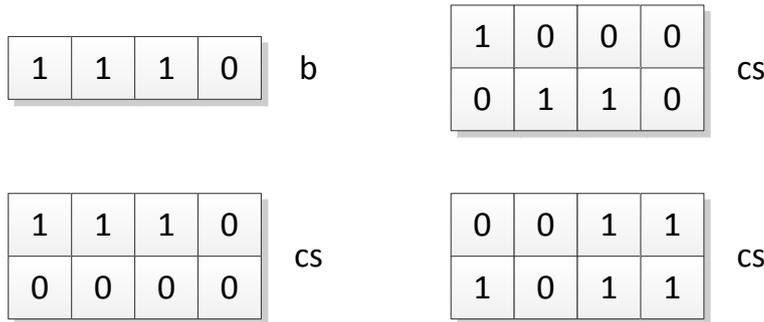


Figure 4.2: Binary (b) and multiple carry save (cs) representations of the decimal number 14

A Partial Carry Save (PCS) format is a mix between binary and carry save format, in which the additional carry bit is not available for each digit. Figure 4.3 shows multiple different representations for the same value in binary (b) and carry save (cs) format.

Automatic inference of CS arithmetic in synthesis has also been subject to prior research [111]. However, it has focused on the general synthesis of CS structures, not their selective use to accelerate floating-point operations. Other approaches use CS arithmetic internally to individual operations, but not between them [34, 112].

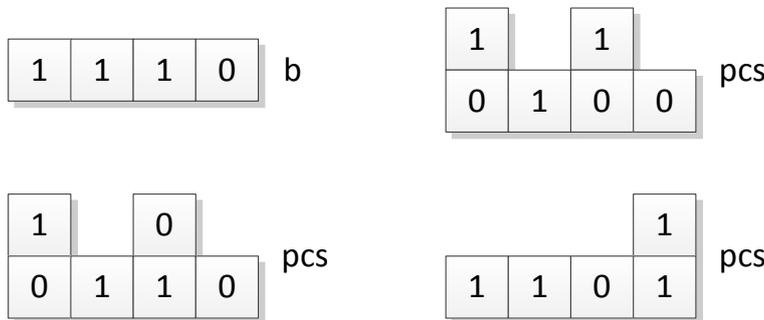


Figure 4.3: Binary (b) and multiple partial carry save (pcs) representations of the decimal number 14

4.3.5 *Fused Multiply Add*

The multiply-add fused unit, which was later referred to as FMA, was first proposed in 1990 [51]. More recent works introduce improved FMA architectures, but often target stand-alone ASICs or units integrated into CPU pipelines. Thus, they use IEEE 754-conforming representations for all input operands as well as the result [20, 63, 95]. [96] gives a survey of the wide spectrum of FMA architectures developed from 1990 to 2007.

The principle of fused operators has also been applied to other computations, such as fused dot products [98, 105], again having standard-conforming interfaces.

The application-specific use of non-standard formats for improved numerical accuracy has been proposed for FPGAs, e.g., in [15]. The use of non-standard formats to improve performance is presented in [34] for the use of a multiply-accumulate unit. It uses a PCS representation to achieve low latency at the addition stage but relies on application-specific knowledge of the input and output value ranges.

Implementations of Radix 4 and 16 exponents showed improved addition speed but slower multiplication [24].

Many existing floating-point libraries for FPGAs omit subnormals (which only marginally extend the representable number range) to improve performance [32, 131]. The units developed in this thesis will also follow this approach.

In contrast to the publications discussed above, others focus on the assembly of complete datapaths from individual operators. FloPoCo [36] exploits a language mixing features from VHDL and C++ to describe pipelines of floating-point operators. However, it does not automatically perform operator fusion. Langhammer et al. developed a floating-point datapath compiler which can generate fused floating-point operations from a subset of C. The generated datapaths have standard IEEE 754 inputs and outputs [72, 73]. Both of these prior works cannot handle control flow.

The approach presented here extends these prior works by the selective use of (partial) carry save number formats and by the integration in a C-to-HDL Compiler.

HARDWARE SYNTHESIS OF LARGE, IRREGULAR C CODE

FPGAs have been shown to be beneficial for the fast and efficient implementation of control applications. However, many HLS tools aim at the translation of relatively short input programs to hardware. Often, these algorithms (e.g., from signal and image processing applications) also have a very regular control flow, consisting of one or more small-bodied loop nests that are often amenable for optimizations such as unrolling and pipelining.

However, code from domains such as control engineering may have a considerably different structure, with large loop bodies holding (tens of) thousands of individual operations (often floating point), e.g. CVX-GEN generated C code (compare Section 2.3). Compilation of such codes not only requires the sharing of hardware operators, but also the efficient storage and forwarding of a multitude of intermediate results. Both academic as well as industrial synthesis tools have great difficulty coping with such input programs.

Thus, this work concentrates on creating a HLS system capable of translating this type of irregular “C”-code into accelerators executing on FPGAs. Using LLVM as the front- and mid-end (providing target-independent optimizations), and based on the HLS back-end Nymble [4], this chapter presents Nymble-RS, an HLS engine aggressively exploiting resource sharing to fit the required computations on FPGAs. As a target architecture, it aims for a platform containing one or more software-programmable processors tightly coupled to reconfigurable logic. Such chips are readily available from multiple manufacturers (e.g., Xilinx Zynq, Altera SoCs etc.).

Section 5.1 gives an introduction to the Nymble compiler, which was used as *starting point* for this work, so this section does not present novel contributions. Afterwards, Section 5.2 explains some of the new features that have been integrated in Nymble-RS in the course. Parts of this section have been published in [4, 8, 9]. Finally, Section 5.3 and Section 5.4 present two of the main contributions of this paper, the synthesis of a distributed micro code architecture with advanced intermediate handling. They have been previously published in [8].

5.1 THE NYMBLE COMPILER

As stated above, Nymble-RS is based on the Nymble compiler. The intention of this section is to give an overview of structure of the Nymble compiler.

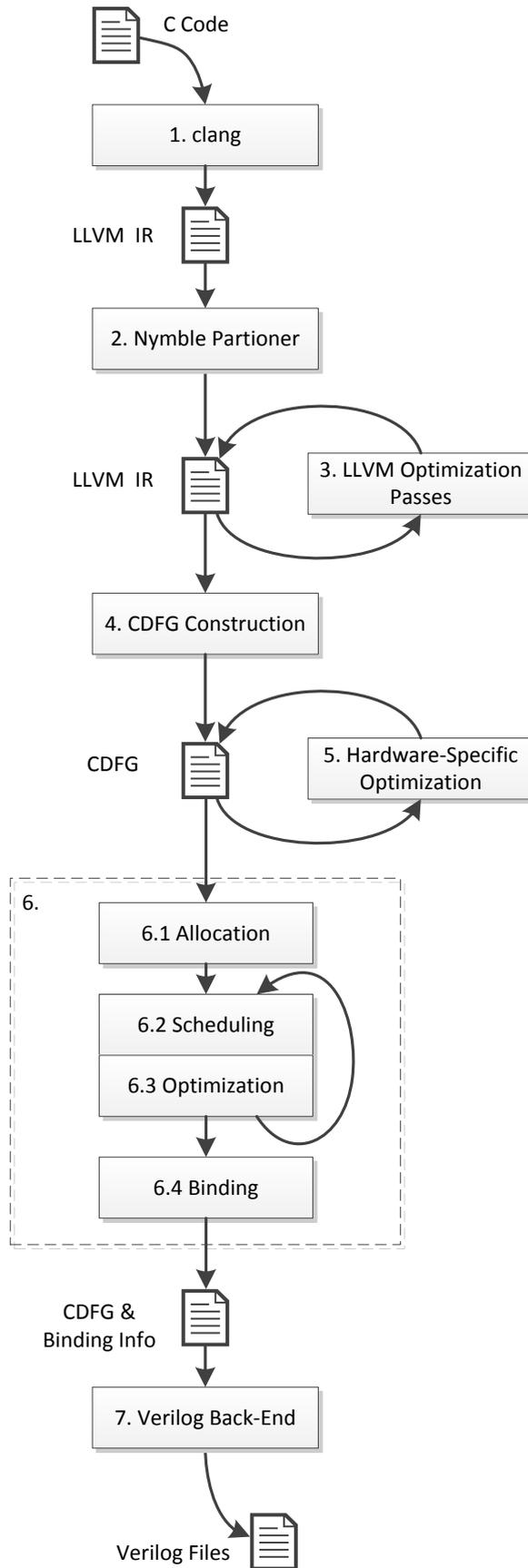


Figure 5.1: Compile Flow of Nymble/Nymble-RS

Nymbble is a hardware/software co-compiler for adaptive computer systems. In contrast to COMRADE [41], it creates *statically* scheduled hardware accelerators. In contrast to most industrial compilers, it does not only generate the hardware, but also creates modified software, including the glue code required for hardware-software communications. Furthermore, Nymbble is not limited on the translation of a complete function. Instead, it offers a finer partitioning granularity which allows it to accelerate only *parts* of a function.

The compile flow is shown in Figure 5.1. The starting point is a software program written in C. In the C code, the hardware region is defined using pragmas.

In the first step, the C code is parsed using the LLVM [74] C/C++ front-end `clang`. It translates the input program into the LLVM IR. A modified `clang` is used which accepts Nymbble-specific hardware region pragmas and converts them into special marker instructions at the region entry and all of its exits. An example for the transformation of C code into LLVM IR can be found in Section 3.2.

In the second step, the IR is passed to the Nymbble partitioner. It automatically extracts the code marked as hardware region into a separate function. Calls within this region are automatically inlined. Variables required from outside the accelerator are passed as arguments to the hardware function. Likewise, variables modified in the accelerator function become "out" arguments if they have uses in the software part of the program.

Next, multiple LLVM optimization passes are run on the IR. The passes are shown in Table 5.1. Further passes may be added by the user of the compiler, depending on the application needs. For example, the synthesis results presented later in this work have been created using the additional passes `-indvars`, `-mem2reg`, `-loop-rotate`, and `-loop-unroll`. While `mem2reg` helps to reduce the number of memory accesses, `indvars` and `loop-rotate` are required by `-loop-unroll` to successfully unroll small loops.

In the fourth step, the optimized LLVM IR is the input for the construction of the Nymbble IR, a CDFG hierarchy (see Section 3.2 for any example of a CDFG construction). To be exact, a Control Memory Data Flow Graph (CMDFG) is constructed for each loop, which is a CDFG with additional edges for memory dependencies. The loop's operations are the nodes of the CDFG. Dataflow dependencies are represented by the data edges. Furthermore, all control flow is converted to conditional dataflow (multiplexer) and conditionally executed operations, e.g., memory writes and inner loops. These conditional operations are controlled by control edges. In addition, memory dependency edges are inserted between all memory accesses that are not marked as independent by LLVMs alias analysis. The remaining steps of the high-level synthesis are then executed on this CMDFG.

Table 5.1: LLVM passes used in Nymble (table taken from [4])

Name	Description
-simplifycfg	Removes dead or unnecessary basic blocks.
-lowerswitch	Transforms switch instructions to a sequence of branches.
-loop-simplify	Guarantees that natural loops have a preheader block; their header block dominates all loop exits, and they have exactly one backedge.
-sccp	Sparse conditional constant propagation.
-instcombine	Algebraic simplifications.
-dce	Dead code elimination.
-mergereturn	Transform function to have at most one return instruction.
-basicaa, -scev-aa	Alias information for load and store instructions based on program independent facts and scalar evolution analysis.
LoopInfo	Mapping of basic blocks to natural loops.
DominatorTree	Dominance relation for basic blocks.

In step five, hardware-specific optimizations are performed. Such optimizations include dead tree elimination, which removes unused operations from the graph, and operator chaining, which reads the operation latency from a configuration file and allows two or more low-latency operations to be executed in the same cycle. Furthermore, the automatic insertion of BlockRAM scratch pad memory, a contribution of this work discussed in Section 5.2.4 takes place at this point.

Afterwards, in the sixth step, the core operations of the high-level synthesis are performed. Nymble formerly only supported the generation of fully spatial hardware using As Soon As Possible (ASAP) scheduling, or experimental modulo scheduling (without support for resource sharing). In this work, allocation, binding and iterative resource-limited scheduling is added (see Section 5.2).

Finally, the scheduled CDFG is passed to the Verilog back-end. The back-end was also modified in this work to support resource sharing and the Zynq target platform (see Section 5.2.5). Furthermore, the microcode described in Section 5.3 is generated here.

5.2 NYMBLE-RS IMPROVEMENTS

In this work, the baseline compiler Nymble was extended with resource sharing capabilities, yielding a new version called Nymble-RS. During this extension, multiple features have been added to the compiler.

5.2.1 Allocation

Creating a resource shared hardware requires the allocation of functional units as discussed in Section 3.2. Nymble-RS supports three different ways of handling the allocation step.

First of all, Nymble-RS can generate hardware without any allocation constraints. In this case, operators and registers are automatically allocated as required during the binding step. The number of functional units used depends on the number of parallel operations of the same type scheduled in the same stage or, if pipelining is possible, the number of operations of the same type scheduled in stages that are active simultaneously.

As unlimited allocation may lead to excessive resource usage, especially when floating-point arithmetic is translated, constraints are generally imposed. To simplify the constraint definition, Nymble-RS is equipped with a heuristic that enables the HLS tool to *automatically* make a sensible, though not necessarily optimal choice. The only user-set constraint is the total number of Floating-Point (FP) units to be used, which serves as a general limit on the hardware area required by the accelerator.

The simple heuristic aims to reflect the mix of FP operations in the CDFG with the mix FP operators in the hardware, subject to the constraint that at least one FP operator is available for each of the required operation types. The initial minimum number of FP operators of each type is computed by determining the fraction for each operation type of the total user-set number of FP operators, based on the relative static frequency these operation types occur in the CDFG, and rounding down. The initial solution is then incrementally refined by determining the operators farthest away from their ideal ratio, and adding another operator of this type. This happens until the user-set upper bound on the total number of FP operators is reached.

While hardware built using this heuristic already delivers good performance and energy efficiency (see Section 8), it could serve as a initial solution for more intelligent iterative refinement, e.g., by simulated annealing, which could then also consider individual operator areas (see Section 9.2).

As a third option, Nymble-RS allows the direct definition of individual operator limits via command line parameters. This way, the user gains maximum control over the instantiated operators.

Listing 5.1: Single Scheduling Iteration

```

1   alapSchedule = getAlapSchedule( CDFG );
2   while notFinished() begin
3       opsReady = CDFG.
      getAllOperationsWithPredecessorsScheduled();
4       op = getOpWithHighestPriority( opsReady, alapSchedule );
5       op.stage = getEarliestPossibleSchedule( op );
6   end

```

5.2.2 Iterative Scheduling

Originally, Nymble offered only ASAP scheduling and an experimental modulo scheduling pass [97]. Modulo scheduling computes the optimum II, but it cannot be used for large loop bodies due to its high runtime for larger input graphs. On the other hand, the existing ASAP scheduling did not support any resource limits. Therefore, Nymble-RS is extended with an iterative scheduling pass.

A single scheduling iteration is similar to list scheduling [12] using "longest path" as priority with two major differences. Listing 5.1 shows a simplified pseudo-code implementation of the scheduling algorithm.

At first (line 1), an As Late As Possible (ALAP) schedule is computed, which is used as priority function by the following steps. While most iterations uses a unlimited ALAP schedule as priority, the last iteration rely on a resource-limited ALAP schedule which can significantly differ from "longest path" priority.

The scheduling algorithm works on a list of operations that are "ready" for scheduling, which means that all their inputs (and other dependencies) are already scheduled (line 3). Out of this list, the operation with the highest priority (with the smallest ALAP schedule) is chosen and assigned to the earliest possible stage (line 4 and 5). In this point, it differs from list scheduling, which iterates over the stages and selects the operation with the highest priority that can be places in the current stage. The earliest possible stage is thereby the lowest stage in which all inputs are ready, all dependencies are ready, and a hardware operator is free (if a limit constraint was defined).

Nine iterations of the scheduling algorithm are performed sequentially. Some new optimization passes depend on scheduling information, such as the schedule dependent tree-height optimization described in Section 5.2.3. They are executed *inside* the iterative scheduling, between the iterations.

Afterwards, a two step optimization is performed to reduce the required amount of registers for intermediate storage. In the first step, the whole schedule is traversed from back to front. Thereby all operations except store, output and input operations are moved backwards as far as possible (ALAP scheduling). In the second step, all operations with two or more inputs are moved up again (ASAP scheduling). Figure 5.2 illustrates the effect on a small example graph, where it reduces the

registers required for intermediate storage from two to one. In Section 5.4, further methods for reducing the number of registers are proposed, which are applied after the scheduling.

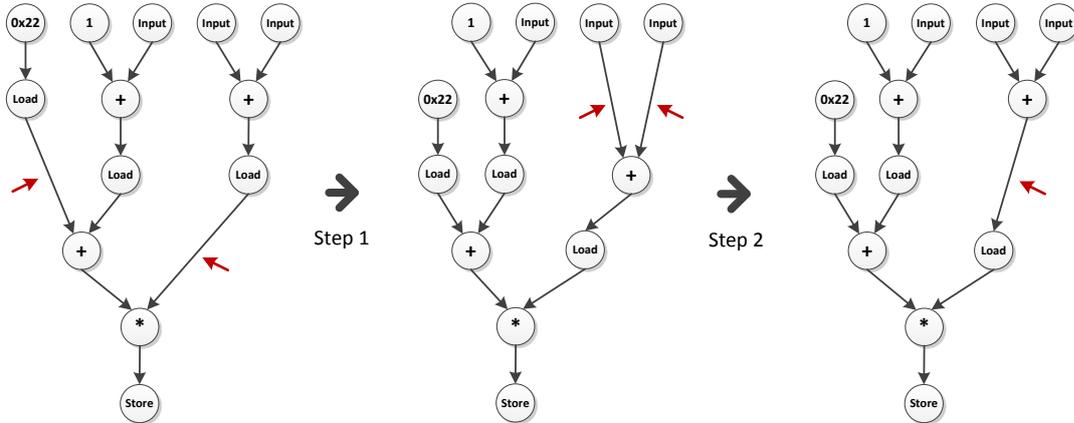


Figure 5.2: Schedule optimization performed to reduce register amount (red arrows indicate intermediate values that must be stored in a register)

Note that there is no strict separation between allocation, scheduling, and binding. For example, the allocation does not always happen before scheduling. If no limit is given by the user, the actual number of functional units required may be settled during the scheduling by the number of operations scheduled for parallel execution.

5.2.3 Schedule-Dependent Tree Height Optimization

Schedule-dependent tree-height optimization [48] has proven useful to reduce the height of trees of *identical* operations by taking into account the availability time of input signals. Without knowledge of the schedule, tree-height optimization must create a balanced tree. However, if some of the inputs to that tree are available later than others, a balanced tree may not be an optimal solution (see Figure 5.3a). Schedule-dependent tree-height optimization can find a more optimal solution by starting the computation with those operations that are available first (see Figure 5.3b). It will in most cases shorten the schedule length and in some cases even reduce the amount of required hardware operators (e.g., in Figure 5.3b only one adder is required after the optimization).

In this work, the technique is extended beyond the usual arithmetic (e.g., add, multiply etc.) to also recognize specific *patterns* of comparators and multiplexers in the CFG as **min** and **max** operations, which can then also be subjected to this optimization. An example is shown in Figure 5.4: (a) shows the original CFG consisting of compare operations and multiplexers. Our algorithm now recognizes that these combinations of compares and multiplexers actually represent min/-max operations (b). They can be subject to schedule-dependent tree-

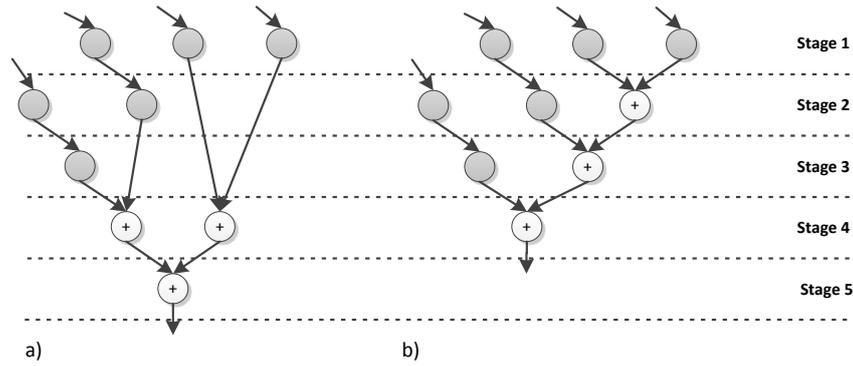


Figure 5.3: Generic and schedule dependent tree height optimization at the example of a three adder tree

height optimization which allows a rearrangement as shown in (c). The schedule is shrunk from five down to four stages in this example.

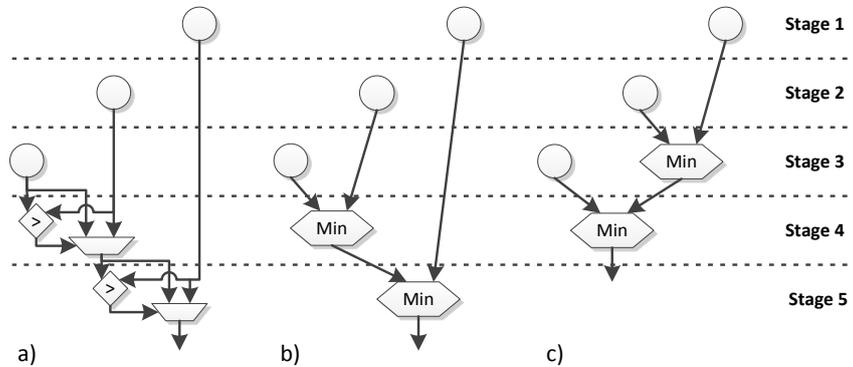


Figure 5.4: Detection and optimization of minimum / maximum computation trees

5.2.4 BlockRAM-based Scratch Pad Memory

Nymble memory accesses are cached on all supported target platforms (see Figure 5.7 in Section 5.2.5 and [4]). However, to reduce memory bandwidth demands, the system also supports automation for the use of BlockRAMs as scratch pad memory, instead of accessing the cached memory hierarchy.

To enable this feature, the user can explicitly mark arrays for BlockRAM storage. The compiler will then automatically create BlockRAM read and write operations instead of main memory accesses. Note that the automation provided by Nymble even extends to arrays that are *shared* between SPP (software) and RCU (hardware): specialized data-movement units are synthesized by the compiler which, when entering or exiting the accelerator, copy the contents of the local arrays from/to the main memory shared with the SPP.

Since BlockRAMs on the target devices (currently Xilinx Virtex-5, Virtex-6 and Zynq) have only two ports, multiple instances will be

replicated if more than one read port is allocated. Each instance will offer a dedicated read port. The write ports are all connected to ensure that each write is stored in all BlockRAMs. The RAM design uses the simple dual port design pattern to enable the automatic use of Distributed RAM instead of BlockRAM if the only a small RAM is required.

Pointers present a difficult problem: once static LLVM alias analysis has determined that a pointer points only to a *single* BlockRAM-stored array, each access through that pointer will be mapped to that BlockRAM, too. If a pointer can point to *multiple* arrays marked for BlockRAM-storage, all of these arrays will be stored in the same BlockRAM (start addresses are automatically assigned by the compiler).

Note that, the pointer support is limited to pointer variables that are assigned in hardware part. Pointers assigned in the software and "used" in the hardware not supported. If such a pointer can possibly access an array for which BlockRAMs storage was requested, the request is ignored. Furthermore, if a pointer can point both to main memory and to arrays for which BlockRAMs storage has been requested, the latter request is also ignored, too. In both cases, the array data stays (and is accessed) in the main memory hierarchy.

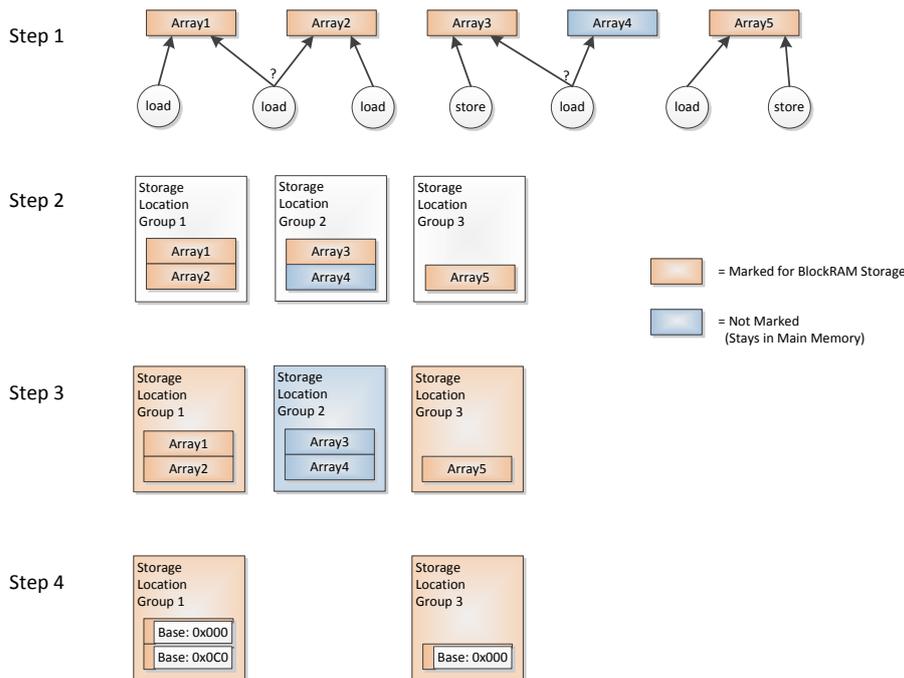


Figure 5.5: Collecting arrays marked for BlockRAMs storage

Figure 5.5 shows an example of the working principle. In a first step, all accesses to the arrays marked for BlockRAMs storage are collected. In the example, eight accesses are found. Two of them are pointer accesses of which can potentially access two different arrays.

Second, all arrays are assigned to a storage location group. If two arrays accessed by the same pointer access, they are merged into the

same storage location group. All other arrays are assigned to separate storage location group.

In the third step, storage location groups are assigned to BlockRAM or main memory storage. If a group includes main memory arrays and arrays marked for BlockRAMs storage, it is set to set to main memory storage (e.g., storage location group 2 in the example in Figure 5.5). All other storage location groups inherit the the storage assignment from their arrays.

Finally, base addresses are assigned for each array in the storage location group that target BlockRAM. The first array is placed at the beginning of the BlockRAM, so it's base address is set to zero. The base address is incremented by the size of the previous array. For example, assuming that `Array1` is 192 bytes wide, the base address of `Array2` is set to 192 (= 0x0C0).

After the final storage location was determined, the top-level CDFG is modified (see Figure 5.6). Originally, it contains the main memory base address of each array as an input node. In step 1, input nodes get replaced by the base addresses are assigned by Nymble-RS. In step 2, main memory accesses are converted to BlockRAM accesses, if they access an array marked for BlockRAMs storage. Finally, data-movement units are created at the hardware entrance and the hardware exit of the top-level CDFG. They copy the data from the original base address in main memory to the Nymble-assigned base addresses in the BlockRAM and and vice versa. In the CDFG, data-movement units are variable latency operations. Thus the execution of the hardware is stalled while a data-movement unit is active.

Note that a command line option exists to combine all BlockRAM arrays into a single storage location group. It can be used to reduce the BlockRAM utilization. Furthermore, the creation of data-movement units can be selectively suppressed via command line to avoid unnecessary data transfers.

The BlockRAM instantiated uses the output register available on Xilinx devices. Thus, each access requires two clock cycles. The output registers allow operation above 200MHz even if the output is connected to large multiplexers without any registers in between.

5.2.5 *Support for Zynq / AXI as Target Platform*

The original Nymble targeted a Virtex-5 based architecture using the MARCII cache system [4, 70]. However, MARCII did not allow operation above 110 MHz. Furthermore, in the original Nymble, every load operation caused a two cycle pipeline stall, even if the data was available in the cache. Finally, the Virtex-5 is a deprecated chip. Therefore, in Nymble-RS, the Xilinx Zynq-7000 SoC [129] is integrated as a modern target platform.

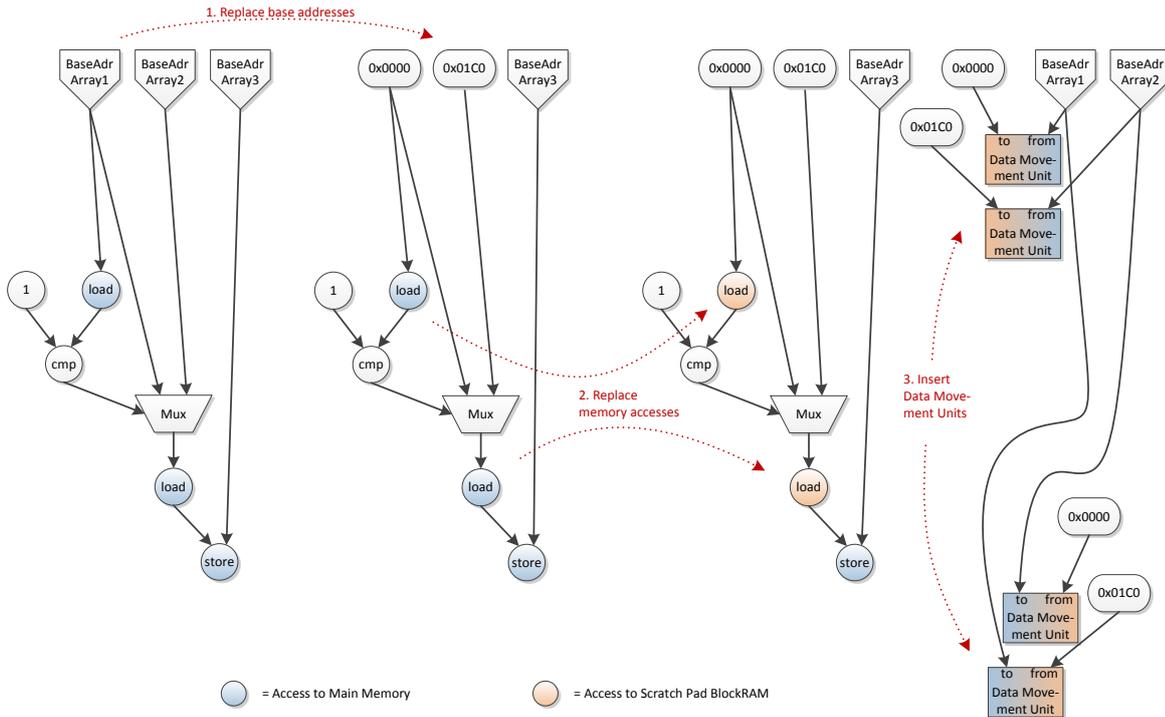


Figure 5.6: Necessary CDFG modification for arrays stored in BlockRAM

The Zynq SoC is composed of a dual-core ARM Cortex-A9 [10] Processing System (PS) and an integrated Programmable Logic (PL). PS and PL are connected to each other via multiple AMBA AXI interfaces [11]. Figure 5.7 gives an overview about the available connections.

From the PL, four 64-bit/32-bit configurable AXI slave high-performance ports allow direct access to the DDR memory and the on-chip memory. Furthermore, two general purpose 32-bit slave ports allow the PL to access the central interconnect of the PS, thereby giving access to all I/O peripherals of the SoC, e.g., CAN controller, UART, SPI etc. Finally, a 64-bit AXI slave Accelerator Coherency Port (ACP) allows cache coherent and low-latency access to CPU data in L1 and L2 caches. It is directly connected to the Snoop Control Unit (SCU) of the ARM processors. From the PS, the PL is accessible via two 32-bit general purpose AXI master ports (GP0 and GP1).

The base design proposed in this work is shown in Figure 5.8. To create a tight coupling between the accelerator and the processor, and to avoid the need for cache flushes, this initial design uses the ACP for cache-coherent and low-latency memory accesses.

Furthermore, the generated hardware kernel is accessed by the software using the AXI master GP0 port. The registers that control the hardware kernel are mapped into the ARM core's address space. The software can write input data, start the execution and read output data from the kernel. All required software and hardware for this interaction is automatically generated.

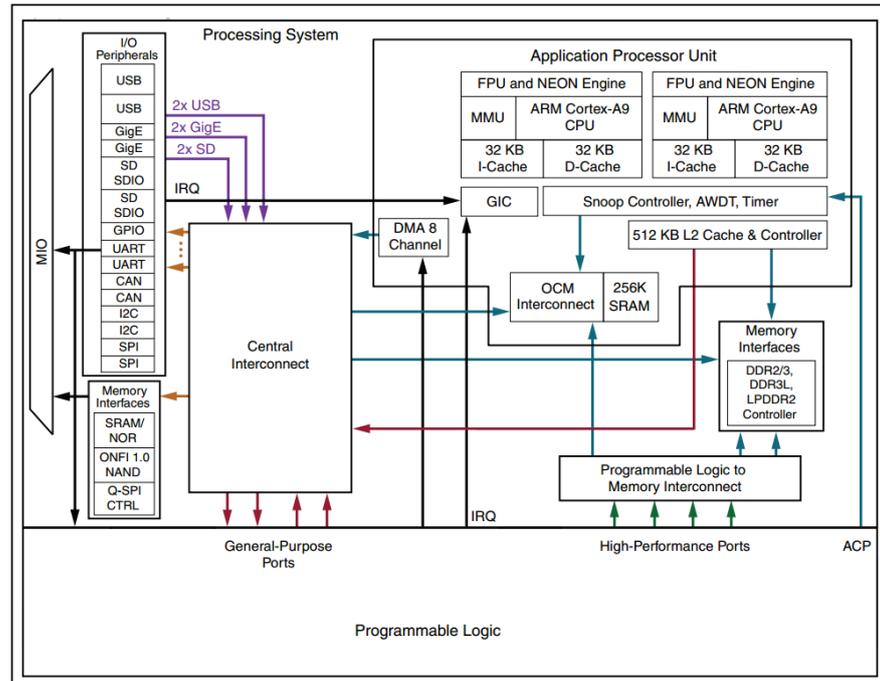


Figure 5.7: Zynq-7000 SoC Architectural overview (image taken from [129], some details are hidden to direct the focus on the AXI connections)

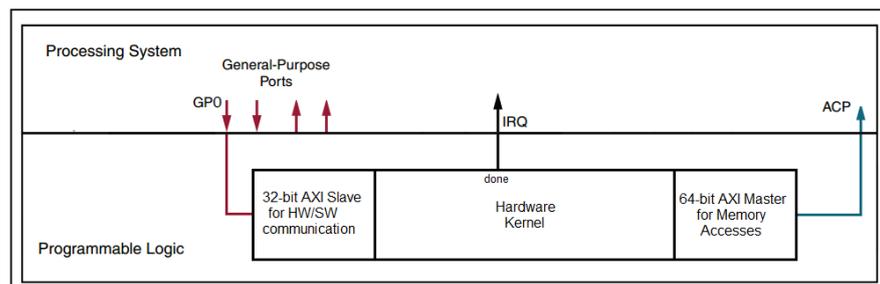


Figure 5.8: Proposed base design for the Zynq-7000 SoC target

The compiler was modified to generate memory accesses through the AXI port. In the original Nymble, load and store operations were so-called variable latency operations. They were statically scheduled with a latency of a single cycle, but they could dynamically stall the pipeline if the access takes longer.

In Nymble-RS, when the Zynq target is selected, load and store operations instead are divided into *two* parts: a *load-start* and *store-start* operation that starts the memory access by sending the correspond AXI command and a *load-finish* and *store-finish* operation that waits for the operation to finish if necessary. Only the two "finish" operations will stall the pipeline if the memory access is not completed in time. However, a proper scheduling of both parts ensures that the pipeline stalls occur only in unavoidable situations, e.g. in case of a cache-miss. In the current design, the minimum distance between *load-start* and *load-finish* is set to 15 cycles, while the minimum distance between

store-start and *store-finish* is set to 18 cycles for operation at 200 MHz. These operation latencies have been observed in the running hardware kernel using chip scope.

The two "start" operations will stall the pipeline only if the AXI port is not ready to receive a new command. To avoid stalls due to non-ready AXI ports, a "start" operation may only be executed every second cycle.

To distinguish between concurrent memory accesses, the AXI ID is used. To each pair of "start" and "finish" operation, an ID is assigned. It is sent by the "start" operation together with the load or store command. When the answer returns, it's ID allows to recognize to which command it belongs. On the Zynq ACP, the ID is three bits wide, thus up to $2^3 = 8$ accesses can be in-flight simultaneously. If a loop contains more than eight load or store operations, multiple operation-pairs must use the same ID. In this case, special care is taken so that the activity of these operation-pairs does not overlap.

5.3 OPERATION MULTIPLEXING USING DISTRIBUTED MICROCODE

Branch-free code like that generated by CVXGEN is very promising for hardware synthesis as the degree of parallelism is now only limited by data dependencies and available resources (no control dependencies exist). However, the relatively large chip area required for individual floating-point operators, combined with the large number of operations which occur, for example, in typical auto-generated solvers, require *sharing* of operators among multiple operations. This section presents some of the techniques Nymble-RS uses to generate resource-shared hardware implementations. As will be seen, fitting the long irregular (not dominated by small-bodied loops) computations at all into mid-size FPGAs (80-240K LUTs) requires significant effort for the tools. Nymble-RS uses static scheduling, but is able to handle variable-latency operations such as cached memory accesses by stalling/restarting the entire datapath. In this thesis, the statically determined execution order is referred to as *stages*, which may differ from the actual clock cycles due to cache misses and other main memory latencies, which basically "stop the world" from the perspective of the datapath.

Sharing a hardware *operator* between N different *operations* in the CDFG usually results in the creation of a N -to-1 multiplexer for each input. Two aspects have to be considered for this approach: first, especially for integer arithmetic operators, the size of the multiplexer often exceeds the size of the actual operator. Most hardware compilers thus multiplex only large operators when reusing them saves a lot of resources (e.g., floating-point operations, cache ports to main memory etc.). Secondly, increasing degrees of sharing lead to denser interconnections between data sources (intermediate value storage, outputs of other operators). However, once a connection between a data source and an

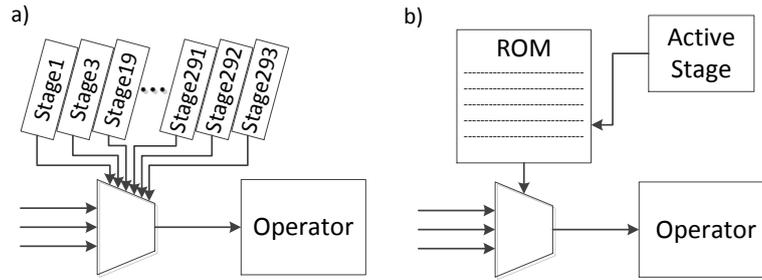


Figure 5.9: Example for time-multiplexing operators

input multiplexer has been created, the connection can be *reused* for all data originating from that source. Thus, even though an operator is reused for different operations in different stages, the multiplexer width no longer grows. At most, all inputs of all operators are connected to all data sources, leading to a worst case of $O(n^2)$ connections for n operators. While the hardware of each multiplexer is relatively simple even with 64b-wide inputs, the quadratic growth of the interconnection density can lead to place-and-route problems later. Note that intermediate registers (see below), which also act as operators in this model, also contribute to the growth of interconnection density and multiplexer size.

In Nymble-RS, the operations are scheduled as described in Section 5.2.2. After scheduling, the minimal II is determined. It is set to the lowest number that avoids memory dependency (or resource) conflicts between successive iterations. A minor improvement of the II of the computation could be achieved by using modulo scheduling. However, modulo scheduling is slow when applied to a graph containing thousands of operations and would lead to excessive run-times of the Nymble-RS tool flow.

The translation of complex irregular “C” code may result in a CDFG with thousands of stages which would induce a correspondingly large number of FSM states in the controller. By itself, that would be manageable, as thousands of states are still efficiently mappable to one-flipflop-per-state controller implementations (note: not necessarily one-hot encoded!). For a resource-shared datapath, however, the true difficulty (sketched in Figure 5.9.a) lies in the extreme number of *fan-ins* each hardware operator, shared between multiple stages, would need to accept from the controller. Typically, these would be one-hot input multiplexer select signals from *each* associated controller state. This excessive number of signals is not efficiently implementable, even when the logic synthesis and mapping tools attempt to reduce routing congestion by logic/register replication (see Section 7.3 for a discussion).

As an alternative, the use of *partitioned distributed microcode* is proposed. A binary-coded global micro-code program counter (μ PCg) keeps track of the global execution state of the datapath. Each operator is provided locally with a dedicated micro-code ROM that controls

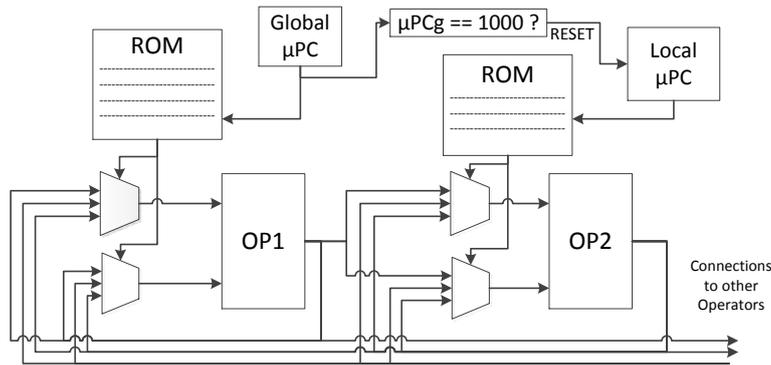


Figure 5.10: Using a shallower microcode ROMs, locally addressed by μPCl

their *individual* enable and select signals. By *distributing* the control signal sequences to the operators, the fan-in per-signal reduces to just one (the operator micro-code ROM, as shown in Figure 5.9.b).

When this scheme is applied in its most basic form, each operator ROM would need to have a depth equal to the global number of stages. However, some operators are active only in parts of the schedule, most of the entries in their ROMs would be zeros. As an alternative, shallower ROMs can be created which just encompass the range between the first and last stages an operator is active (rounded up to the next power-of-2) and which are addressed using *local* program counters μPCl . Each μPCl is started when the μPCg reaches the start of the operator's active stage range. Figure 5.10 shows this for OP2 which is assumed to be active only in stages 1000...1045 and thus more efficiently controlled by a 64-entry microcode ROM.

When switching from per-stage state-registers in the controllers to sequential microcode, the capability for pipelining would normally be lost. While this would not have a major impact on many use-cases, as heavily-resource shared micro-architectures have only limited potential for overlapped execution of iterations, a solution was devised to lift even this limitation: *partitioning* the address spaces of the microcode ROMs leads to a μPC value of p activating the enable and select signals for both the stage p , as well as for the stage s where

$$(s < p) \quad \wedge \quad (s \bmod \Pi = p \bmod \Pi).$$

As an example, for $\Pi=600$, an $\mu\text{PC}=1011$ would activate both stages 1011 and 411 simultaneously. Note that computations for the partitioned microcode address spaces are all performed at compile time and affect only the ROM contents; no additional hardware is required at execution time. In the case studies presented later in this thesis, this actually allows a small but significant overlap of the execution of two iterations.

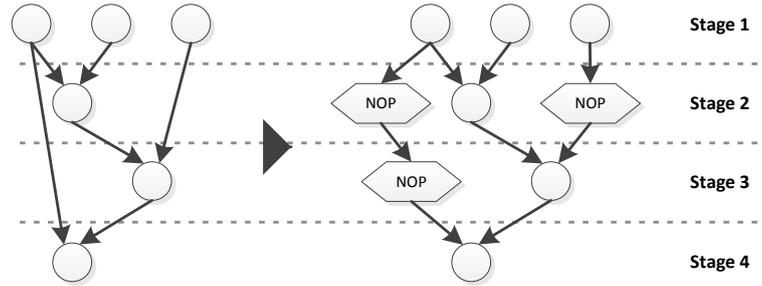


Figure 5.11: NOP chain of length two to delay intermediate results from Stage 1 to Stage 4

5.4 REGISTER REDUCTION FOR INTERMEDIATE VALUE STORAGE

An intermediate result computed earlier in the schedule has to be retained until the last time (indicated by the number of the schedule step, here called *stage*) it is used, thus defining its *lifetime*. In larger programs, such as CVXGEN generated code, a high number of intermediate results must be stored simultaneously. Thus, the way these intermediate results are stored significantly affects the required chip area. In a spatial approach each intermediate result would have a dedicated register connected to its source operator for buffering the value. But this is not practical for larger programs or loop bodies containing thousands of floating-point operations, as it would require *thousands of 64 bit registers*, which would then cause a massive growth in the size of operator input multiplexers.

5.4.1 Individual Delay Registers

As a simple solution intermediate values can be stored-and-forwarded from stage to stage in *individual registers*, which could be modelled as No-Operations (NOPs) in the CDFG. Note that these NOPs are a type of *operation* (similar to Add, Mul, Div etc.) which can *share* actual hardware registers (which act as *operators* in this case). In Nymble-RS a fast linear-scan register allocator [93] is used to map NOPs to registers. While feasible, using just individual delay registers would still require too much chip area: early experiments determined that in some test cases the datapaths have long value lifetimes and would result in more than 75% of all operations being NOPs.

5.4.2 Shift Registers

Instead of individual registers *shift registers* can more efficiently hold values for longer time intervals. In the CDFG they are expressed as multi-stage NOP operations and can be mapped very efficiently to FPGA primitives such as SRL16 shift registers etc. A shift register

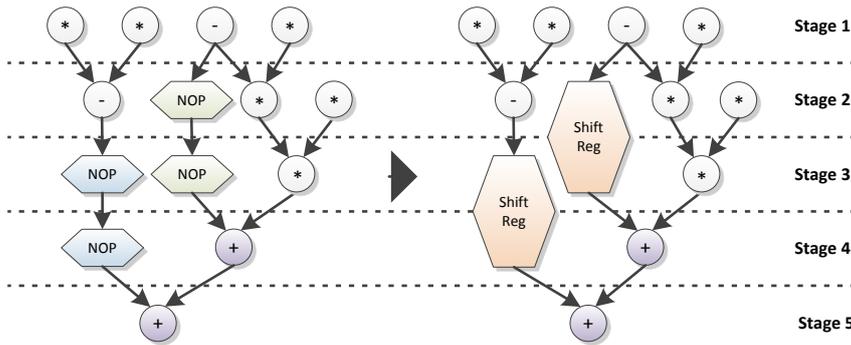


Figure 5.12: Reducing the number of NOPs using shift registers

of depth k can delay an intermediate value for k stages. Shift registers can be reused to delay values from multiple data sources: as long as their inputs and outputs are accessed in *different* stages, the actual lifetimes may overlap.

In Figure 5.12 the left example uses NOP-chains to store the intermediate values. As the lifetime overlaps, at two different hardware registers must be used (coloured blue and green in the example). Assuming that both additions in stage 4 and 5 are executed on the same addition unit, the left input of this addition unit must be connected to both registers using a multiplexer. Alternatively, a single two-stage shift register can be used to store the two intermediate values, since they are enqueued/dequeued at different schedule stages. This way, no multiplexer is required on the left input of the addition unit.

In this work powers of two are chosen as the depths of the predefined shift registers, thus allowing the easy composition of longer delays with only few shift register operators.

5.4.3 Spilling to Scratch Pad Memories

If *many* values have to be retained for long lifetimes, even the use of shift-registers requires too much area (see Table 7.2). Instead, similar to compiling into machine code for a register machine, excess values from registers can be *spilled* to on-chip BlockRAM banks.

Nymble-RS can use BlockRAM-based on-chip memory as scratch-pad memories to store arrays (see Section 5.2.4 for details). The same infrastructure can be employed to delay values for longer periods of time, as long as these times are shorter than the II of a loop. If that bound were exceeded, a value from the next iteration would overwrite one still needed in the current iteration. For many irregular computations (such as the complex solver codes examined later in this work), the II of outer loops will often be very long (close to the total schedule length). This is actually amenable for a resource-shared microarchitecture, as the reuse of operators within the same iteration leaves them unavailable for computing data for the next iteration.

Figure 5.13 shows the use of spilling on the same example which was already used above. The four NOPs are replaced by load and store operations in the CDFG. Again, load and store for both paths happens at different schedule stages, allowing to use the same load and store units (and the same scratch pad memory). Thus, also in this case the multiplexer of the left adder input can be saved.

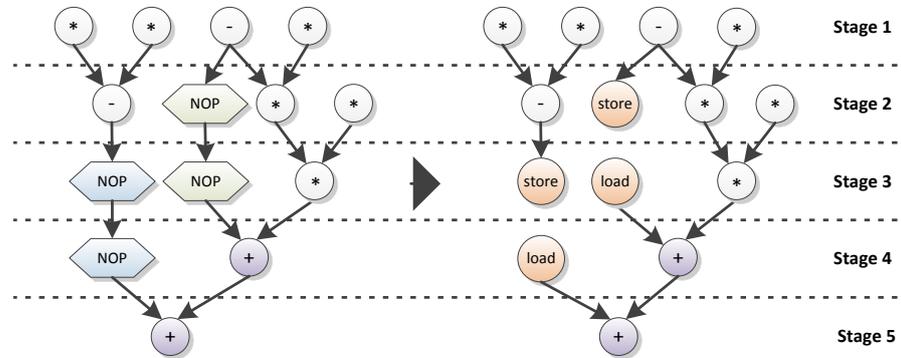


Figure 5.13: Two NOP-chains are replaced by store (S) and load (L) operations

All scratch pads are operated in a simple dual-port mode (one read and write in parallel), which allows their implementation in either BlockRAM or Distributed RAM on Xilinx FPGAs, as Distributed RAM does not support true dual-port operations. Thus, a single scratch pad can accept only a single value per stage and also produces only a single value per stage. As before (in Section 5.2.4), this approach is explicitly chosen over true dual-port operation, as using simple mode gives the logic synthesis tool the freedom to flexibly pick the best implementation (BlockRAM or LUT-based DistributedRAM) for each scratch pad. Furthermore, on recent Xilinx FPGAs the BlockRAM data port width doubles from 36 to 72 bits when using simple dual-port mode instead of true dual-port mode [123].

Datapaths of large C code blocks often need to delay many values, some of which may be read or written in parallel. In this case, multiple scratch pad memories are employed automatically. However, in the first step, Nymble-RS always tries to re-use existing scratch pad memories to store the intermediate values. Scratch pads used to store arrays are extended for this purpose, so that the intermediate values are stored *behind* the array data in the same BlockRAM. Only if all existing scratch pads are busy, new scratch pads are generated by Nymble-RS for the remaining intermediate values.

The (remaining) intermediate values are assigned to a scratch pad by solving a graph colouring problem that assigns all accesses occurring in the same stage into separate banks. To solve the colouring problem, the DSATUR (Degree of Saturation) algorithm is used [19].

Alternatively, for small problems with less than 1000 nodes Brown's algorithm is executed [21] with modifications suggested by Brèlaz [19].

The number of scratch pads is equal to the number of colours required to solve the graph colouring problem. The size of each scratch pad is equal to the number of values that must be stored in parallel. New scratch pads are created and sized automatically as needed; they do not need to be predefined by the user.

5.4.4 *Recomputation vs. Storage*

The last approach to reduce the amount of registers is to *recompute* values as needed instead of storing them. This can be done only if two preconditions are fulfilled.

First, the inputs to the computation are already available in the correct stage *without* introducing the need to store *additional* intermediate values. For example, constants, loop-inputs or intermediates that are stored anyway can be considered "available".

Secondly, a hardware unit must be available in the target stage for the required operation. The benefit of saving a 64-bit registers could be eliminated totally by the hardware cost of an additional hardware unit. However, adding small units such as integer adders can be beneficial.

If both conditions are fulfilled, the CDFG is modified by inserting the operations to recompute the value when required. This will avoid the need to store the intermediate value, assuming it is not required elsewhere.

5.4.5 *Combination of Methods*

Nymble-RS exploits a combination of multiple of these methods to reduce area. As will be shown in Section 7.4, the most efficient results are achieved by spilling longer lifetimes to scratch pads (spanning more than three stages), and then using individual registers allocated using linear scan to create the few remaining, very short NOP chains.

Floating-point arithmetic both in software as well as in hardware is often implemented following the standard IEEE 754 [55]. For many applications, single-precision computation suffices, which is preferable both for the reduced storage size as well as increased memory bandwidth. But some applications require double-precision operations for numerical stability. On SPPs, which commonly use the same Floating-Point Unit (FPU) for both single and double-precision, arithmetic operations often show similar performance, regardless of the precision used. On FPGAs, however, double-precision operations often have a significantly higher latency than single precision [131].

In this chapter, multiple double-precision floating-point units are proposed, most of which do not strictly conform to IEEE-754. Large parts of this chapter have been presented before in [6] and [7]. In some cases, synthesis results for more-recent FPGAs than in the original publications have been added.

6.1 CARRY SAVE FUSED MULTIPLY ADD

Many signal processing and control engineering applications have large numbers of floating-point multiply-add operations at their core. When considering the use of reconfigurable compute units to speed-up these algorithms, the implementation of fast multiply-add units often becomes crucial.

Orthogonal to the performance of individual units is the system-level performance vs. area vs. energy balance. To allow system-level evaluations to access a wide range of benchmarks, the *automatic* use of the new units by system-level design tools, such as high-level language to hardware compilers is required. In general, attempting to realize *all* required multiply-add operations by very fast (low latency, high throughput) implementations is not efficient, as the area (and possibly energy) overhead can quickly become prohibitive. It is thus worthwhile to employ strategies that employ the fast multiply-add units only selectively on the critical path of a computation.

Entire *chains* of multiply and add operations are typical for datapaths of some applications, including convex optimization problem solvers. To reduce the application-level latency, it is required to reduce the latency passing through the complete FMA unit, starting at the multiplier input and ending at the adder result. This eliminates the MAC unit proposed in [34] from consideration, as it only exploits low

latency *addition*. However, the idea of a mantissa in PCS format can be extracted and used in FMA designs proposed in this thesis.

In this section, two FMA units are developed, both calculating $R = A + B * C$ using CS representations in their input and output format: One of them uses a PCS format and is portable to older FPGAs (e.g., Xilinx Virtex-5). The other one uses a Full Carry Save (FCS) format, and exploits special capabilities of recent FPGA generations (e.g., Xilinx Virtex-6 and later). For brevity, $\{C, A, B\}_M$ denotes the mantissas of C , A , and B respectively.

In the following section, we assume that only the adder input and one of the two multiplication inputs is on the critical path (see Figure 6.1). This scenario was often observed in the CDFG of multiple applications, e.g. CVXGEN generated code or code of signal processing filters. Therefore, in the following sections a custom number format is developed only for the inputs A and C , while B remains in the standard IEEE 754 format.

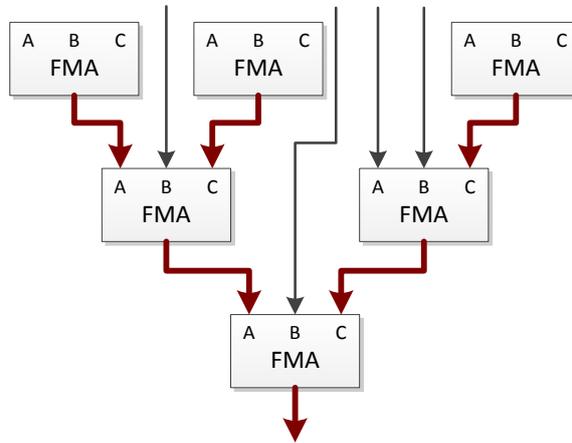


Figure 6.1: Example datapath with only the A and C input of FMAs in the critical path (marked red)

6.1.1 Classic FMA Architecture

Since one of the major means of latency reduction in this work is the avoidance of unnecessary normalization steps (see Section 2.2.1), the exploration of a classic FMA design [51] is chosen as starting point. This architecture, shown in Figure 6.2, is used as a baseline for further optimizations.

Adder and multiplier are fused into a single operation without an intervening normalization step. The multiplier result is instead provided in CS format (please see Section 4.3.4 for an introduction to the CS representation). Furthermore, the performance of the addition stage is improved by performing it partially in parallel with the multiplication $B * C$. The computation of the required shift-amount depends only on the exponents of the tree input values (block "Exponent Diff" in Figure

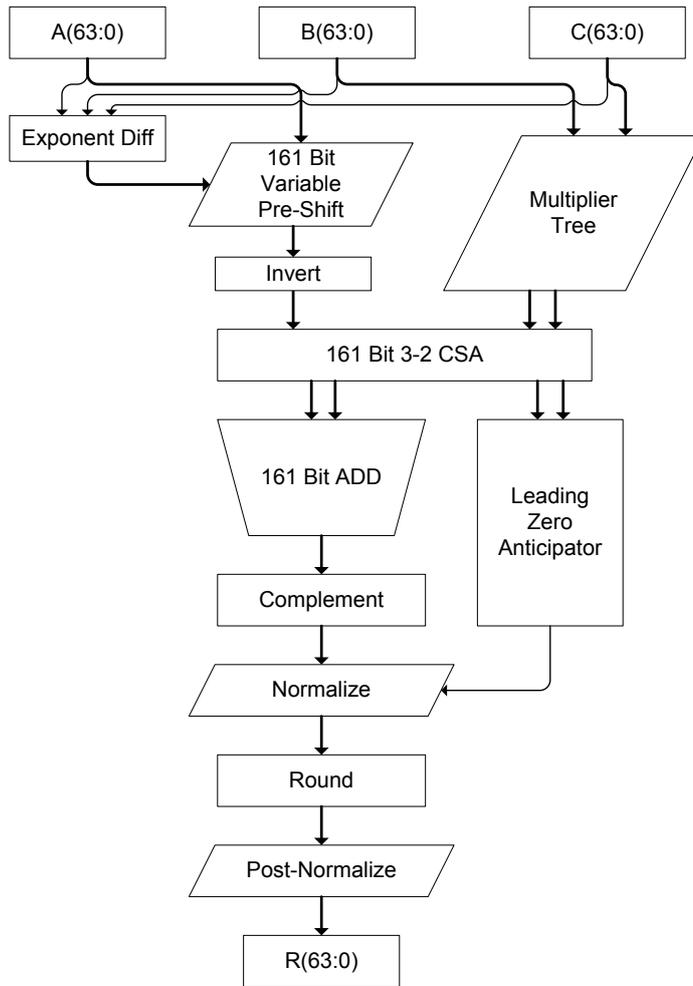


Figure 6.2: Classic FMA architecture [51] with IEEE 754-compliant operands and result (bit widths taken from [96], exponent and sign logic hidden to focus on mantissa computation)

6.2). Afterwards, the input is inverted if the sign of A differs from the sign of $B * C$. Then, a 161 bit variable pre-shift of the additive input A is done in parallel to the mantissa multiplication. Note that on the ASIC circuit presented in 6.2, the pre-shift is much faster compared to the mantissa multiplication.

The result of the pre-shift and the CS result multiplication are then fed into an 161 bit 3-to-2 adder. At this point the mantissa computation is basically finished, but the mantissa is not yet in the correct format.

Since the output of the classic FMA unit is in IEEE 754 format, the internal CS representation has to be converted to that plain binary format at the output. This is achieved by a 161 bit adder followed by a conditional complement block to handle negative numbers. The actual normalization (left-shifting to achieve the implied 1) is guided by a Leading Zero Anticipator (LZA) [51, 99], which computes the shift-distance in parallel with the addition. Rounding to the required preci-

sion, followed by a conditional one-bit right shift for post-normalization (to compensate for rounding overflow), is performed at the end.

6.1.2 Removing post-normalization

Even in its original form (normalization only after the adder), the classic architecture has potential for improvement by just slightly deviating from IEEE 754 (still using binary format, but with modified field widths): By adding an extra bit at the most significant side of the mantissa, the post-normalization right shift at the end can be skipped, because in the case of an overflow during rounding the additional bit is set to **1**). In other words, the additional bit increases the range of possible values of the mantissa m from $1 \leq m < 2$ to $1 \leq m \leq 2$ (compare Figure 6.3).

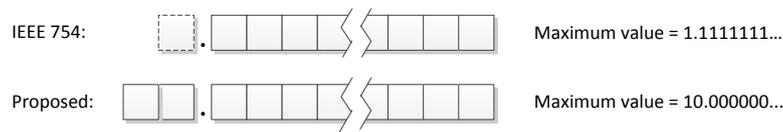


Figure 6.3: Mantissa bit width increased by one bit to eliminate post-normalization

Actually, this requires the use of two additional bits in the custom representation of the mantissa (now 54b), as the leading **1** can no longer be just implied. In practice, if targeting FPGAs with embedded DSP48E blocks (such as the Xilinx Virtex-5, -6, and -7 devices), the slight widening of the internal computation (from 53b to 54b, both including the leading **1**) does not require additional DSP blocks. Furthermore, in this approach of selectively employing custom number formats just on the critical path, only the A and C inputs (which are assumed to be the output of a previous FMA unit) needs to be widened. B can remain in standard format as there is sufficient time for its proper post-normalization.

Figure 6.4 shows how such a FMA unit could look like. Note that while the inputs A and C are increased by two bits, the actual mantissa width is only increased by one bit. Furthermore, the bit width of the multiplication result does not increase at all. Since the mantissa if B is still smaller than two, the result of the mantissa multiplication can be slightly larger than before, but it will always be less than four. In addition, the post-normalization of the input A can be done during the pre-shift at (almost) no cost. Therefore, the bit width of the adder does not increase.

This approach can eliminate the latency of the post-normalization completely at the cost of two additional mantissa bits. However, another way exists to eliminate post-normalization: the elimination of the rounding step. As post-normalization is required to compensate a

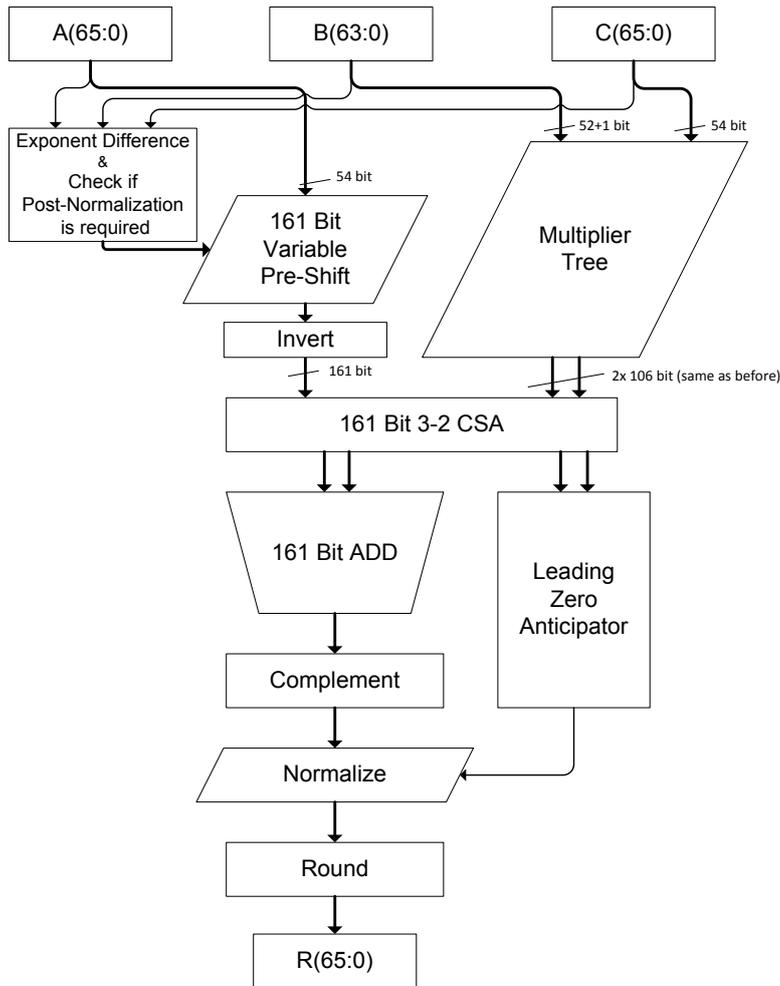


Figure 6.4: Modified FMA with increased mantissa width to eliminate post-normalization

rounding overflow, it can be dropped completely when rounding is not performed at the end of the operation. The next section will target this topic, therefore, the approach shown in Figure 6.4 is dropped at this point and *not* developed any further.

6.1.3 More efficient rounding

It is tempting to eliminate the rounding step entirely. However, while truncation may be acceptable for some applications, others will suffer from the increased rounding error. But by considering entire chains of FMA units during datapath assembly in high-level synthesis (as shown in Figure 6.1), the rounding step can be moved from the output of an FMA unit to the input of the succeeding unit.

The latency improvement is not obvious. However, it was stated above that the path of the A is not time critical because the pre-shift is much faster than the multiplier. Furthermore, this step allows the

integration of the C inputs rounding logic into the CSA tree of the multiplier (see Figure 6.5), adding at most one logic level to the critical path. To allow its execution in parallel with the multiplication, this approach performs the actual multiplication with the *truncated* value of C_M and then corrects an erroneous result afterwards by adding B_M to the product in the case that rounding would have increased C_M by one. Note that rounding in the next iteration requires again an increase of the mantissa width, but this time at the least signification side.

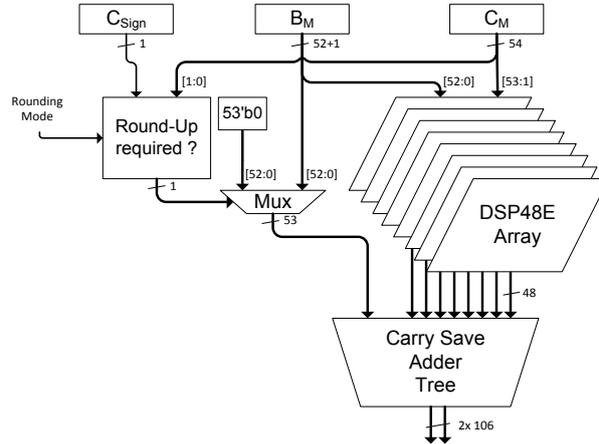


Figure 6.5: Internal structure of mantissa multiplier with integrated rounding unit

In IEEE 754 rounding mode "Round to nearest ties away from zero", the decision whether to round up or not solely depends on the last bit behind the rounding border. In that case, the previously added least significant bit (lsb) of the mantissa C_M is directly connected to the multiplexer input. Furthermore, since the multiplexer requires much less time than the multiplication inside the DSP, it is possible to use the free C-input of two DSP48E blocks for the conditional addition (see Figure 6.6).

Figure 6.7 shows the complete mantissa computation after the FMA was modified as described above. Two rounding units are added to the design: A one for A_M (running in parallel to the pre-shift distance computation), and a second one for C_M , integrated into the mantissa multiplier addition stage (CSA tree or free C-input)).

Note that if the rounding is computed in parallel to the pre-shift, it is no longer possible to perform a post-normalization using the pre-shift because it is not known in advance if the mantissa of A will overflow during rounding. The rounded mantissa of A can potentially take a value of two, which requires one more bit at the most significant side (54 bit wide mantissa, similar to the scenario discussed above). Therefore, the bit width of the adder must be increased by one bit to a total width of 162 bits.

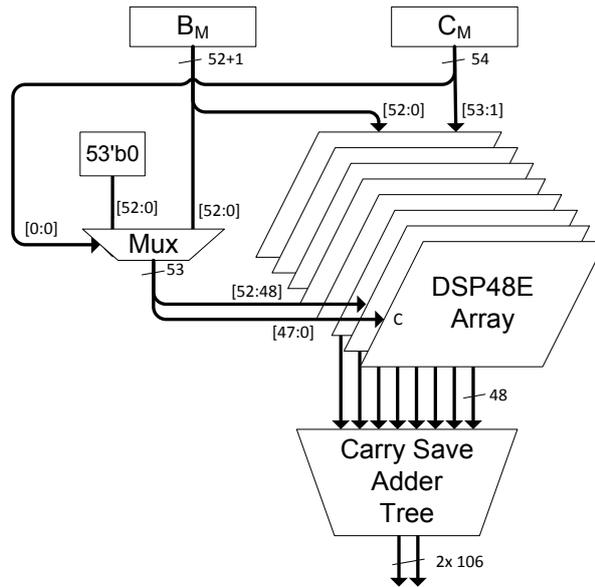


Figure 6.6: Alternative structure of mantissa multiplier with integrated rounding unit for the rounding mode "round to nearest ties away from zero"

6.1.4 Eliminating the variable-distance shift

In the last section, rounding and thereby post-normalization have been successfully removed from the end of the FMA unit. The last step remaining is the normalization which basically consists of a single variable-distance shifter. The number of leading zeros after the addition of two signed numbers can be anything from zero to mantissa bit width plus one (overflow). Large shifts especially occur when the adder input has another sign than the multiplication result, for example, in the computation $-98 + 9 * 11$. Such a situation is called partial annihilation (while $-99 + 9 * 11$ would be a total annihilation). The shifter thus must support distances from zero to the full width, so the msb of the result depends on *every single bit* of its input, that being 162b wide in the FMA unit see Figure 6.8. This is a high fan-in which requires many levels of LUT-logic in an FPGA implementation. Thus, a major improvement in latency could be achieved if this potentially very slow step could be eliminated.

To simplify the final shift, this work proposes replacing it with a multiplexer which is actually doing a shift in larger blocks of bits. To determine the block size, the requirements on the result must be considered and then worked backwards toward the width of the adder. In the result, at least the accuracy of IEEE 754 double-precision format with its 52b mantissa should be achieved. Since the implied **1** must be stored explicitly after this change, one more bit is required. Similarly, since this approach no longer uses an explicit sign bit but two's complement notation, an additional extra bit in the mantissa must be added.

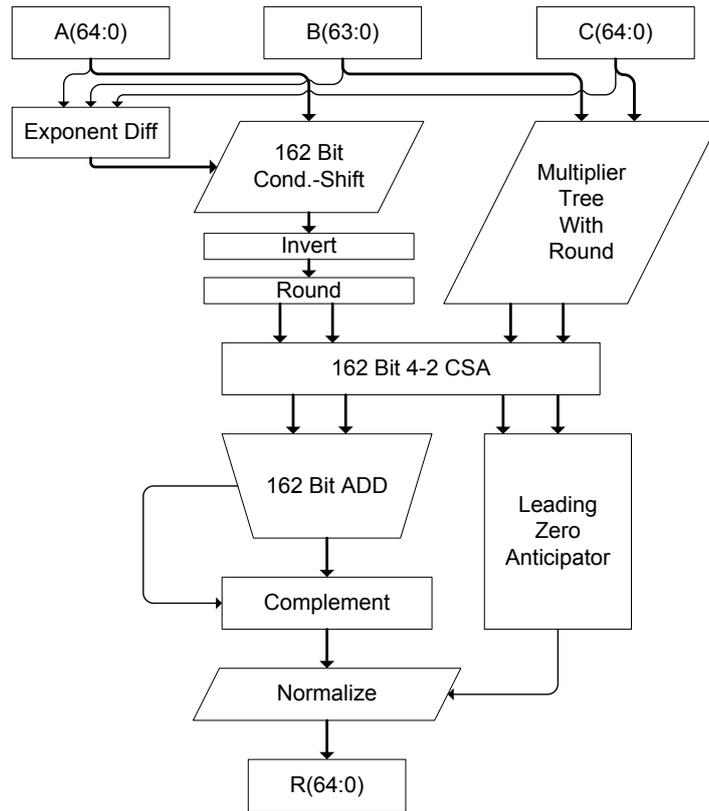


Figure 6.7: Modified FMA with rounding moved into the succeeding operation

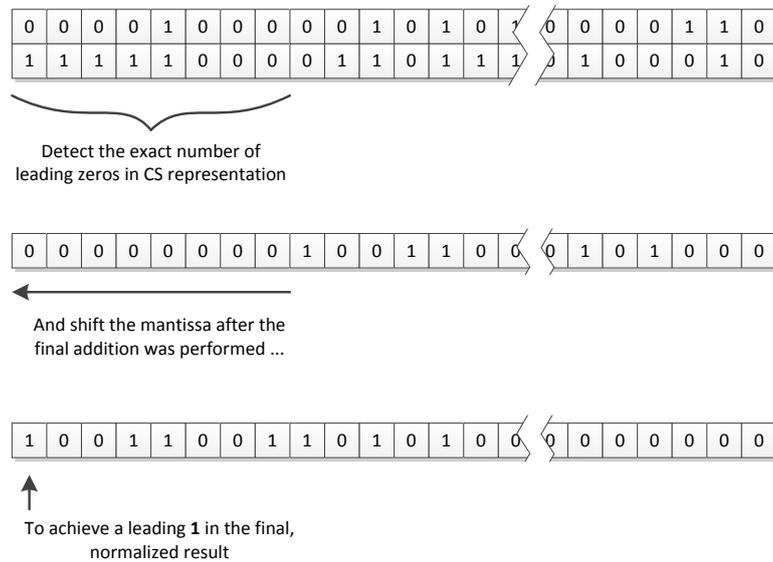


Figure 6.8: Example for a normalization shift

Finally, it is necessary to add a guard bit to catch a possible overflow in the mantissa *. This yields a total width of 55b as shown in Figure 6.9.

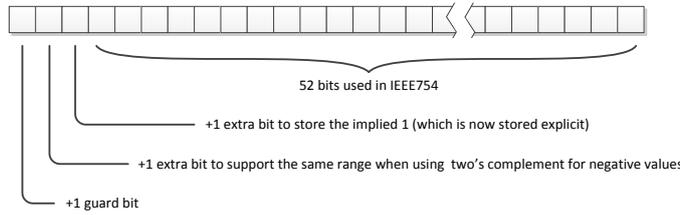


Figure 6.9: The proposed block width

The addition result (whose width is derived later in this subsection) is then converted into blocks of 55b. Since the number of leading zeros in the non-normalized result is unknown and generally not a multiple of 55b, the first non-zero digit could be positioned anywhere in the result. When shifting by multiples of 55b, the result mantissa must thus be composed of at least two blocks. As two blocks are selected, the final result mantissa is now 110b wide in total (see Figure 6.11). Figure 6.10 shows the selection step on the example of smaller 8 bit wide blocks.

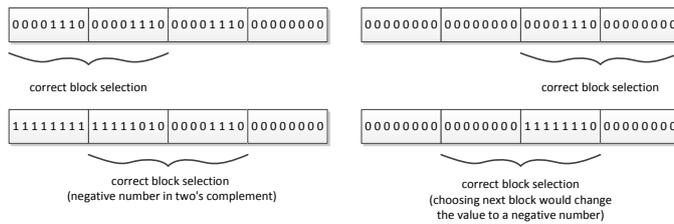


Figure 6.10: Examples for the block selection using 8 bit wide blocks)

After determining the result mantissa width to be 110b, the impact of this decision on the input and internal widths of the succeeding FMA units must be considered. For the first, it is necessary to increase the width of the critical A and C inputs to accommodate a 110b mantissa, while B can remain in IEEE 754 format (52b mantissa plus implied leading 1). The latter is highly beneficial since it reduces the size and latency of the multiplier. On the other hand, the *widths* of the multiplication and addition stages grows significantly: the multiplier now has a (52+1)b wide multiplicand B_M and a 110b wide multiplier C_M , yielding a total result width of 163b. The adder stage grows since, for large exponent differences, the 110b wide addend A_M must be alignable even completely left or completely right of the product $C_M * B_M$. This yields $110b + 163b + 110b = 383b$, rounded up to 385b, the next multiple of 55b as described in Sec. 6.1.5. The entire multiply/shift/add/mux structure is shown in Figure 6.11.

A look at these choices from the circuit performance view shows that the multiplier latency should be unchanged since the height of its CSA

* The reason for the possible overflow in the CS format is discussed in Sec. 6.1.5.

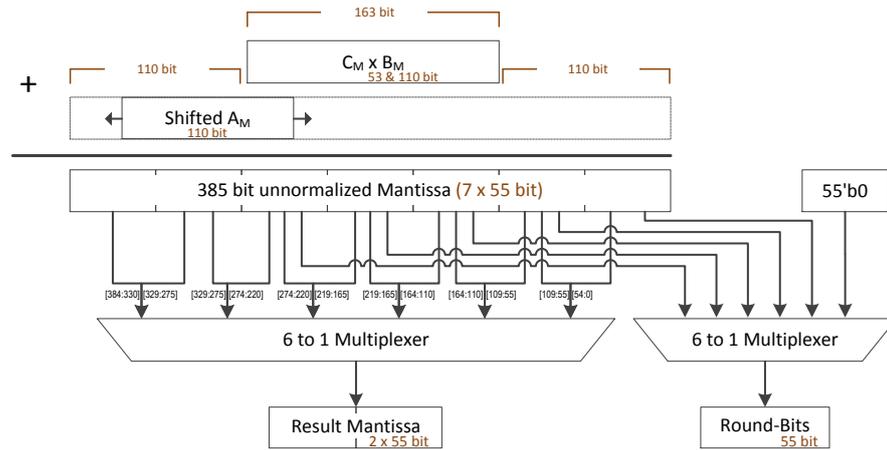


Figure 6.11: Replacing shifting by a 6-to-1 multiplexer

tree depends on the *number* of inputs which has remained constant. However, the increased *width* of the operands has a detrimental effect on adder performance: On a Xilinx Virtex-6 FPGA (speed grade -1), the register-to-register latency of even of single 385b adder is about 8.95ns. Hence, the increase in bit width due to the elimination of the variable-distance shifter can no longer be handled using plain binary format addition. Instead, excessive carry chains are broken by explicitly representing carries of smaller addition widths. This leads to a major shift away from the variations of the IEEE 754 format being used so far (mostly with different mantissa widths) towards a CS representation of mantissas in floating-point numbers.

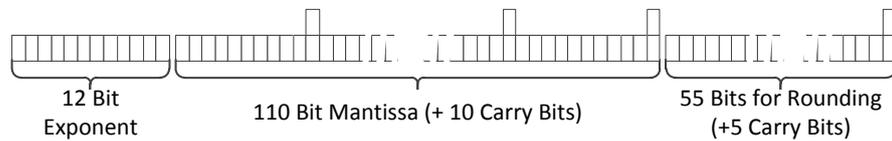


Figure 6.12: Complete floating-point format with PCS mantissa

6.1.5 Floating-point representation using a PCS mantissa

At first glance, an FCS representation using 110 carry bits in addition to the 110 binary mantissa bits is not feasible since it would again double the size of the multiplier. However, the latency of the addition can already be improved by employing just a limited number of explicit carry bits in the mantissa representation. Such a PCS approach has already been demonstrated to be efficient for FPGA implementation [34].

Two constraints need to be considered for optimal carry bit distribution: to simplify the multiplexing step, the carry bits should be equally distributed in every 55b mantissa block. To allow a regular design of the operator, the distance between all carry bits should be equal. Combined,

these two constraints allow the insertion of a carry bit only for every 5th, 11th or 55th bit of mantissa. When evaluating these alternatives, it was discovered that the delay difference between a 5b and an 11b adder is so small (1.650ns vs. 1.742ns) that the more area efficient 11b distribution can be chosen without a significant performance penalty. This reduces the internal FCS widths of a 385b wide sum and 384b of carries to the PCS format of 385b sum and 35b of carries (shown as Carry Reduction in Figure 6.13). Using the same distribution for the CS inputs and the result, the prior 110b two’s complement binary format for the mantissa (derived in Sec. 6.1.4 to match the accuracy of IEEE 754 double-precision) is extended with 10b of carries into a PCS format.

However, rounding becomes more complicated as CS does not guarantee unique representations for numbers: the plain binary representation for the value of 0.5d (decimal) is always 0.1000b (binary). However, when a CS format is used, each digit can take the values $\{0, 1, 2\}$. The decimal value 0.5d could thus be represented in CS format as 0.0200cs or 0.0120cs. Even if the most-significant fractional digit is zero, values larger than 0.5d (which would need to be rounded up) can be represented in CS (e.g., 0.75d as 0.0220cs). Thus, it no longer suffices to examine a single bit to make an exact rounding decision. Instead, all mantissa bits must be considered, even if in rounding mode “round half away from zero” and “round to +infinity”.

This would become very expensive for the current 385b addition result, which could (in the worst case) consist of five non-zero 55b blocks. Thus, this approach accepts some misrounded numbers by considering only a *narrower part of the mantissa* for rounding: only the *single* 55b block (with 5b of carries) immediately to the right of the 110b result chosen by the 6-1 multiplexer is considered for rounding. This results in a truncation before rounding. With this choice an erroneous rounding-down would only occur if the saved carries would ripple through all 55b from the lsb to the msb of the fractional part. The maximum error made when rounding IEEE 754 conform is 0.5 ULP. In this case, the maximum error is increased to 0.500000000000000028 ULP. This inaccuracy seems acceptable. If more rounding accuracy is required, a wider part of the mantissa would need to be considered.

The distribution of carry bits in the PCS format (shown in its entirety in Figure 6.12) does have an advantage with regard to sign-extension: generally, this is a difficult problem for arbitrary carry save two’s complement numbers [108]. However, the carry bits are spread out so that the top bits of the mantissa are always carry-free, allowing the use of conventional sign-extension techniques.

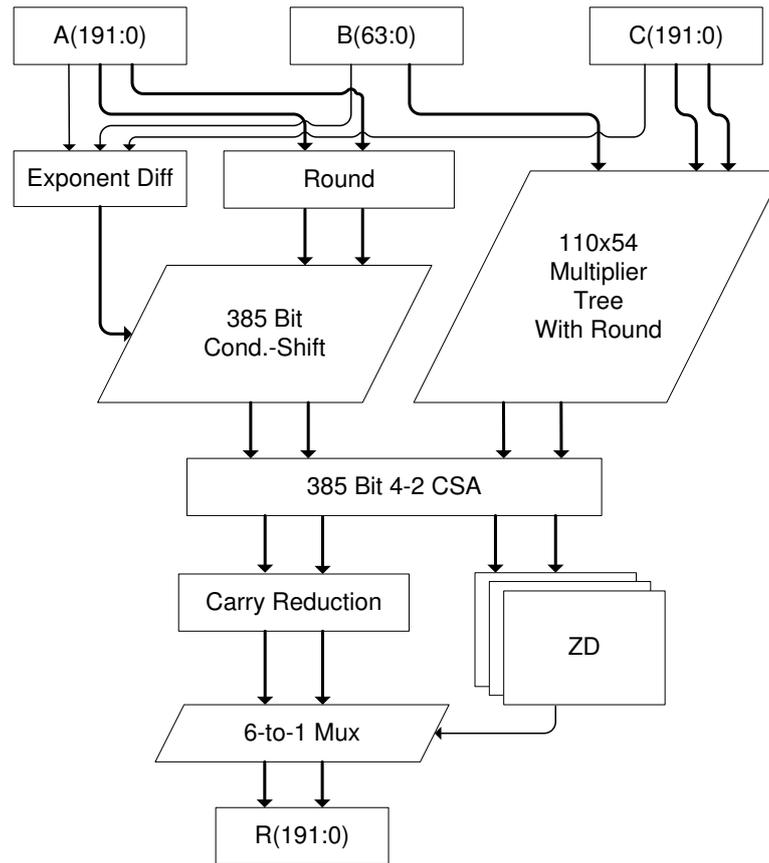


Figure 6.13: Proposed PCS-FMA architecture

6.1.6 PCS-FMA Unit

Figure 6.13 shows the final PCS-FMA unit. It accepts the non-critical input B in IEEE 754 double-precision format, the time-critical inputs A and C are represented as a mantissa in $110\text{b}+10\text{b}$ PCS format, combined with $55\text{b}+5\text{b}$ of rounding data in PCS format, combined with a 12b exponent in excess-2047 notation. The latter was explicitly chosen to surpass the range of the 11b exponent specified by IEEE 754. In total, the A and C operands, as well as the FMA result, are expressed as 192b words.

Since the variable-distance shifter used in prior art is eliminated, there is no need to identify leading zero bits at *single-bit granularity* using techniques such as LZA [99]. But it still needs to be determined how to select the input of the 6-1 multiplexer (see Figure 6.11) to choose the two most-significant non-zero 55b blocks as result. Instead of using an LZA, it suffices to detect and disregard entire 55b blocks of leading zeros using a simple Zero Detector (ZD) to identify the block holding the most significant **1**.

The ZD does need to handle some idiosyncrasies of the two's complement CS format used for the mantissa. Obviously, leading blocks with all **0**s can be skipped (see Figure 6.14.a). However, similarly, leading

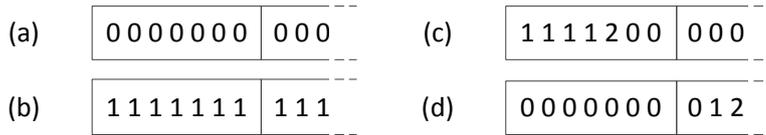


Figure 6.14: Different forms of leading zeros in two’s complement CS representation

blocks with all **1**s can *also* be skipped: while they indicate a negative number, that same number can be represented with fewer bits as long as the msb remains **1**. Thus, leading all-**1** blocks can also be skipped (see Figure 6.14.b, the leftmost all-**1** block is skipped). Furthermore, a block of **1**s followed by a single **2** followed by **0**s to the end of the block is considered a block with value zero (due to the ripple carry from the **2** upwards) and will also be skipped (see Figure 6.14.c). Finally, before actually skipping a leading all-**0** block, it must be ensured that its removal will not alter the value of the succeeding blocks. Figure 6.14.d shows an example for this: at first glance, it appears that the leftmost all-**0** block could be skipped. However, when converting the value of the succeeding block from CS into binary, 012cs becomes 100b. Since that block is now the most significant block (the first one got skipped), the **1** in the msb now indicates a *negative* number, which is incorrect (with the leading all-**0** block, the original value was *positive*). Thus, to avoid these overflows, an all-**0** block *only* is skipped if the first two CS digits of the succeeding block are also **0**, avoiding all potential overflows.

While the Carry Reduction step of Sec. 6.1.5 is carried out in parallel with ZD, the latter is now critical and determines the total FMA latency.

6.1.7 Early leading zero anticipation

The critical path can be shortened further by replacing the ZD units with early leading zero anticipation. Therefore, the idea of zero-value consideration at block granularity is combined with the prior art of LZA units [99]. For each of the FMA inputs, an LZA unit is used to compute the lower bound for the number of leading zeros in the FMA output. Since B_M is in standard format (having an implied leading **1** in the mantissa), it does not need a dedicated LZA if subnormal numbers are disregarded. As long as B is not zero, its mantissa will be greater than or equal to one. So LZA units are required only for A and C .

LZA units are often inexact and have an error of up to one bit position. A further bit of uncertainty is introduced by the product $B_M * C_M$, with $1 \leq B_M < 2$: depending on the value of B_M , the result can almost differ by a factor of two, which is a one-bit-difference regarding the number of leading zeros. Finally, the sum of the shifted (aligned for different exponents) A_M with the product can potentially require an additional bit because the sum of two n -bit values can require n or $n+1$

bits. The total uncertainty sums up to three bits. Figure 6.15 shows an example computation for this uncertainty using a block size of 8 bit.

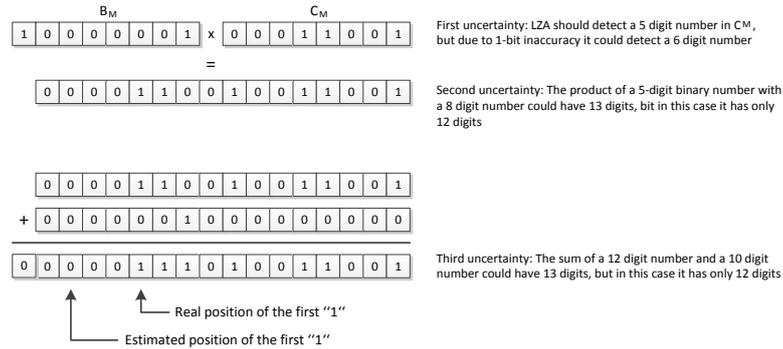


Figure 6.15: Example for the uncertainty introduced by early leading zero anticipation

To compensate for this uncertainty and still exceed double-precision accuracy in the worst case, the result mantissa block size introduced in Section 6.1.4 must be increased from 55b to 58b to make sure that in worst case, at least 53 significant mantissa bits are included in the two result blocks selected.

Special consideration must be focused on the issue of adding a product $B * C$ with an addend A that have different signs, but a similar magnitude. This will lead to many leading zero blocks in the sum. Potentially, *all* of the blocks may be zero if the two addends cancel each other out completely (total annihilation). In these cases of mantissas with very small magnitude, the anticipation error of the LZA-based approach leads to a larger relative inaccuracy compared to the precise (but slower) ZD-based approach described in Sec. 6.1.6. However, since the maximum LZA error is already taken into account by widening the mantissa, it is ensured that even in these extreme cases the FMA-unit is never more inaccurate than IEEE 754 double-precision.

The early leading zero anticipation logic must *reliably* detect all-0 input mantissas. Otherwise, the result block multiplexer could erroneously select leading all-0 blocks for the result, even though a 1 (that should actually be in the leading block) is present in the less significant bits of the sum. Two special cases must be considered:

First, all-0 input mantissas must be detected on all three inputs because the proposed leading zero anticipation technique is only valid for non-zero inputs. Otherwise, it could happen that the zero is returned as result due to a higher exponent while dropping an actual value in the lower bits of the result. For zeros detection the rounded value must be used.

Secondly, results of partial annihilation must be handled differently as they are not guaranteed to contain at least 53 significant bits. Note that the result of one FMA is usually the input of another one, so inputs with large amounts of leading zeros must be expected. While this is no problem for the C_M input because it will be multiplied with B_M and

afterwards contain at least 53 significant bits, it becomes a problem for A_M which may, in the worst case, contain only a single one at the lsb. Figure 6.16 exemplifies the problem.

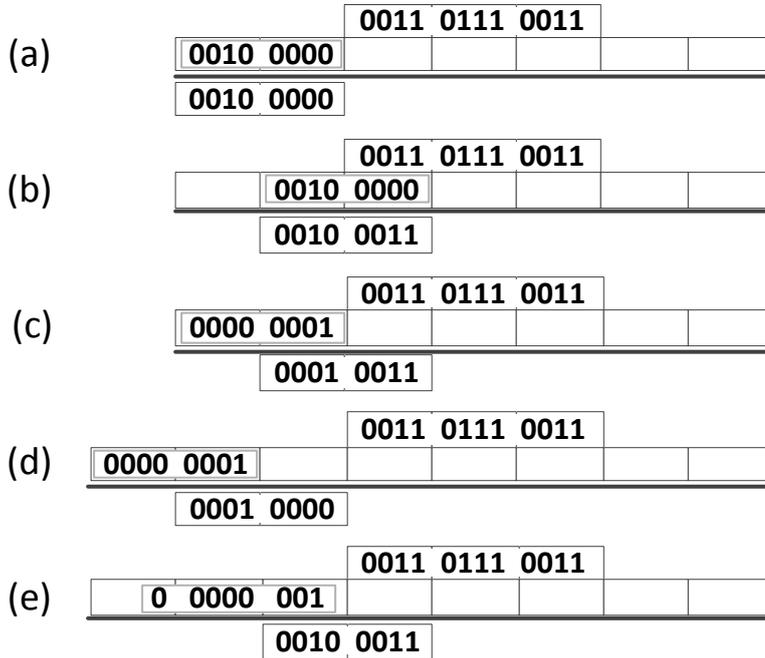


Figure 6.16: Loss of precision due to many leading 0s in maximally shifted A_M

In case (a) the exponent of A is much higher than the exponent of $B * C$. As (a) has only few leading zeros, first two blocks are chosen, which is correct.

In case (b) the exponent of A is only a little bit higher than the exponent of $B * C$. Therefore, the mantissa of A partially overlaps with the product of B_M and C_M . The second and third block are chosen which is also correct.

In case (c) the exponent of A is again much higher than the exponent of $B * C$. We assume the difference in exponent is greater than 500 so the numbers do not overlap. Again, A_M is shifted as far to the left as possible. However, this time the mantissa contains many leading zeros. The second and third block are chosen, which is not correct because the result of $B * C$ is much smaller and no bits of its product should appear in the result. Due to the high number of leading zeros in A_M , the most significant block of result will be discarded even if A_M is shifted by the maximum amount because of a much higher exponent. Therefore, the bit width of the shifter and adder must be increased by a full block width to support large amounts of leading zeros created by annihilation.

Case (d) and (e) show the behaviour with the additional block available. In (d) the second and third block are chosen again, but this time no bit of the product is used in the result. Only in the case that the

exponent of A and $B * C$ are really close (e), bits of the product are used.

6.1.8 FCS-FMA for FPGAs with DSP pre-adders

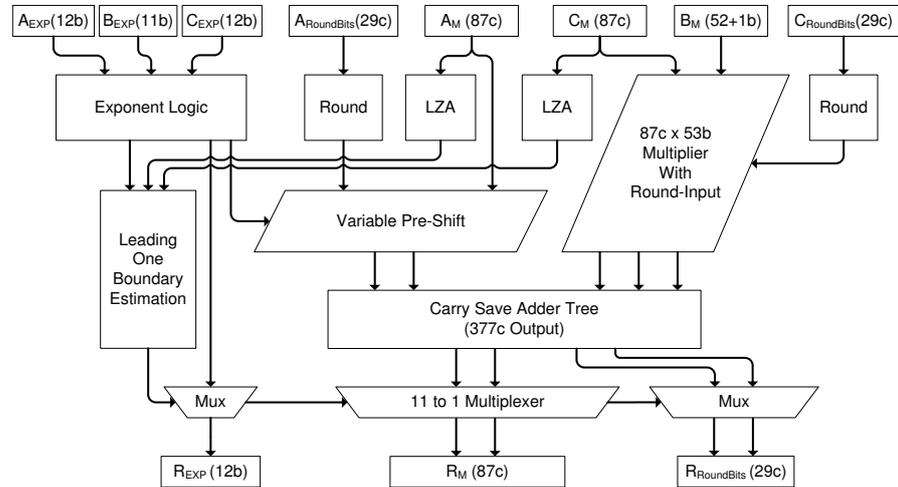


Figure 6.17: FCS-FMA unit exploiting DSP block pre-adders

The improvements described in Sec. 6.1.7 remove the ZD operation from the critical path. However, now the Carry Reduce step (Figure 6.13) becomes time critical. For FPGAs featuring fast pre-adder stages in their DSP blocks even this step can be completely eliminated, but its removal incurs a significant complexity cost.

In contrast to the Xilinx Virtex-5 family, the more recent Virtex-6 and -7 devices provide DSP48E1 blocks that implement a 25b pre-adder on one of their inputs. This pre-adder can be used for C_M to add two 23b blocks of CS partial sum and carry bits, converting them to plain binary format without the risk of a sign-changing overflow. The most significant block of C_M can actually be processed at the full pre-adder width of 25b as it is a signed number itself.

The pre-adders allow the representation of A and C in *full carry save* representation, thus eliminating the Carry Reduce step. However, such a space-intensive format begins to tax the resources even of recent FPGAs. Due to routing difficulties using ISE 14.1 on Virtex-6, it was necessary to reduce the mantissa from 116b (two 58b blocks) down to 87b (three 29b blocks). This reduces the size of most internal modules (multiplier, adder, etc.) by almost 25%, at the cost of a more complex multiplexer at the end (11-to-1 instead of 6-to-1). However, it enables 200+ MHz operation.

When the result mantissa consists of three blocks, blocks of 29 FCS digits (each digit having 1b partial sum and 1b CS carry, together expressed in the unit c from here on) are required to surpass double accuracy: in the worst case, the first result block and the first digit

of the second block can all be zero, but the following non-zero digit prevents the removal of the leading zero block (see Figure 6.14.c). In addition, when using early leading zero estimation, there is a three bit uncertainty to consider, possibly causing three further digits (4c in total) of block two to be zero. On the other hand, this means that even in the worst case, at least 25c in block two and all 29c in block three are significant FCS digits (54c in total), exceeding IEEE 754 double-precision with its mantissa of 52b+1b binary digits.

The inputs to the FCS-FMA unit (shown in Figure 6.17) consist of the three exponents (12b for A and C , 11b for B) and B_M in standard format (52b+1b leading 1). A_M and C_M are represented in FCS as 87c each, accompanied by 29c of rounding data. The output is a 87c result mantissa, 29c of rounding data and 12b exponent.

The width of the result multiplexer must be sized accordingly: the multiplication yields a five block wide result. The shifter aligning the addend A_M to match exponents has an additional three blocks on the right hand (less significant side) and five blocks on the left hand (more significant side), yielding a total of 13 blocks, each 29c wide, for a total width of 377c.

The final multiplexer for the result selects from these 13 blocks the three most significant non-zero blocks. It thus accepts 13 blocks as inputs and selects from 11 possible positions for the three block result R_M , which holds at least 53 significant mantissa digits, possibly shifted across three blocks (87c). A parallel multiplexer outputs the 29c of the mantissa immediately to the right of the actual result R_M for rounding in a subsequent FCS-FMA operator.

6.1.9 Evaluation of the proposed FMA Units

For evaluation, the partial and full carry save FMA unit have been implemented on the Xilinx Virtex-6 FPGA (xc6vsx315t-1). For comparison with the industrial and academic state-of-the-art, Xilinx CoreGen and the FloPoCo library were used to generate IEEE 754 double-precision units for the Virtex-6 family. Note that none of these units supports subnormals [32, 131] and all were constrained to achieve a minimum clock frequency of 200 MHz.

6.1.9.1 Synthesis results

FloPoCo allows the definition of target frequency, technology, and bit width by command line parameters. The FPPipeline command allows optimizations across the multiplier and adder units [35]. The resulting hardware model was synthesized with and without register balancing, using the better result as baseline for the comparison.

In contrast to FloPoCo, Xilinx CoreGen only allows the generation of separate multiply/add units and the specification of individual operator latencies. Thus, the configuration with the lowest total latency that

still managed to achieve the target clock is manually selected. The specific designs chosen were the "low latency" 5-cycle multiplier and "low latency" 4-cycle adder. The P/FCS-FMA units have been manually pipelined to >200 MHz operation.

Table 6.1 shows the synthesis results achieved using Xilinx ISE 14.7. All results are taken from post-layout timing reports. While FloPoCo achieves the smallest implementation (in terms of DSP usage), its latency of 11 cycles is also the slowest in the test. The FCS-FMA unit is the fastest unit, followed by the PCS-FMA unit. Note that the FCS-FMA unit achieves better area efficiency than the PCS variant due to its exploitation of the DSP48E1 pre-adder blocks, which would not be available on earlier FPGAs. However, both of the units presented require more area (LUTs) than their competitors.

As later chapters in this thesis target the Xilinx Zynq SoC, all units are also implemented on the Zynq (xc7z045-2ffg900). These results are included in Table 6.1. For the Zynq and Virtex-7 architecture, Xilinx offers an FMA IP core [117]. However, it is not possible to configure it for a low-latency operation, as the selection is disabled in the configuration dialogue. This may be the reason why it is much slower than a combination of the low-latency multiplier and the low-latency adder IP core.

Table 6.1: Synthesis results

Architecture	Device	f_{Max} [MHz]	Latency [cycles]	LUTs	DSPs	WCT [ns]
Xilinx CoreGen Mul+Add	Virtex-6	244.7	9	1109	13	36.8
FloPoCo FPPipeline	Virtex-6	191.6	11	1515	7	57.4
PCS-FMA	Virtex-6	239.3	5	6225	21	20.9
FCS-FMA	Virtex-6	212.9	3	4762	12	14.1
Xilinx CoreGen Mul+Add	Zynq(-2)	327.1	9	1219	13	27.5
Xilinx FMA IP	Zynq(-2)	246.6	9	1584	10	36.5
FloPoCo FPPipeline	Zynq(-2)	223.3	11	1625	7	49.3
PCS-FMA	Zynq(-2)	307.1	5	6919	21	16.3
FCS-FMA	Zynq(-2)	288.6	3	5653	12	10.4

Figure 6.18 shows the minimum computation time for a single multiply-add-operation. It is calculated by multiplying the minimum cycle time with the number of clock cycles required to complete one computation. The proposed FMA units are up to 2.6x faster than their closest competitor.

6.1.9.2 Numerical Accuracy

As discussed earlier, with the exception of limitations in rounding fidelity, the P/FCS-FMA units are guaranteed to reach or exceed IEEE 754 double-precision accuracy. To study the impact of the potential mis-rounding (see Sec. 6.1.3), valid but random data is fed into a pair of FMA units recursively computing the value $x[50]$ as described in Equa-

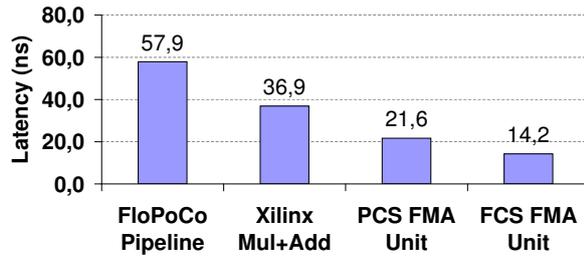


Figure 6.18: Latency (minimum clock period multiplied with the pipeline length) for FloPoCo, Xilinx and P/FCS-FMA operations on the Virtex-6

tion 6.1, where B_1 and B_2 are random numbers with $1 < |B_1| < 32$ and $1 > |B_2| > 0$.

$$x[n] = B_1 * x[n - 1] + B_2 * x[n - 2] + x[n - 3] \tag{6.1}$$

Figure 6.19 illustrates this computation pipeline, disregarding architecture specific operation latency. The same computation is also performed on data widths of 64b (IEEE 754 double), 68b, and 75b using the Xilinx CoreGen floating-point operations as reference. The 68b and 75b variants employ a wider mantissa for improved accuracy.

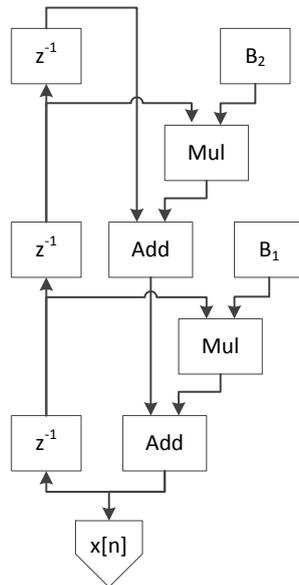


Figure 6.19: Computation pipeline used for accuracy comparison

Figure 6.20 shows the average mantissa error of the 64b, 68b, PCS- and FCS-FMA implementations. The result of the 75b CoreGen computation was used as golden reference to gauge the errors of the less

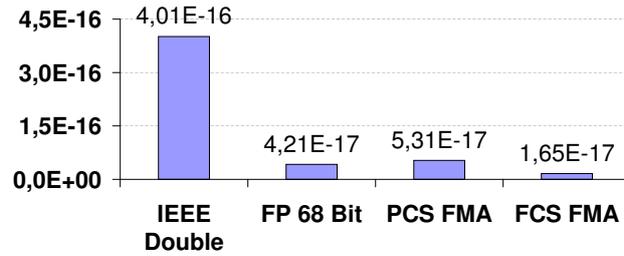


Figure 6.20: Average mantissa error in $x[50]$ (arithmetic mean over 20 computations)

accurate implementations. Both PCS and FCS-FMA units clearly outperform standard IEEE double-precision in terms of average accuracy.

6.1.9.3 Energy consumption

The energy consumption was analyzed by the Xilinx XPower tool considering the actual switching activity of the units. Post-layout delays were extracted and the activity recorded in VCD/SAIF format using the Xilinx ISim simulator on the benchmark computations described in Sec. 6.1.9.2. The pipeline is examined in steady-state (producing one $x[i]$ per clock cycle) after sufficient priming. The energy per computation shown in Table 6.2 was then calculated by dividing the power results by the number of clock cycles per second.

Table 6.2: Average energy consumption per multiply-add computation (nJ)

Xilinx (Mul+Add)	FloPoCo	PCS-FMA	FCS-FMA
0.54	0.74	2.67	2.36

The increased performance of the P/FCS-FMA units comes at a 4x to 5x increase in energy consumption. The XPower analysis details showed that most of the energy was drawn in the large CSA trees of multiplication and addition. Obviously, the P/FCS-FMA units are not suitable for ultra low power operation. However, due to the much lower general energy consumption of FPGAs compared to general purpose GPUs and SPPs [1, 54], FPGA designs using P/FCS-FMAs may still be competitive energy-wise with other implementation technologies. Furthermore, both architectures are applicable to the high-performance computing domain.

6.1.10 Automatic P/FCS-FMA unit insertion in high-level synthesis

Manually replacing critical discrete multiply-add operations by FMA operations and performing the appropriate type conversions is both tedious and error prone. A pass integrated into the C-to-hardware compiler Nymble-RS (Section 5.2) performs the required analysis and transformations automatically.

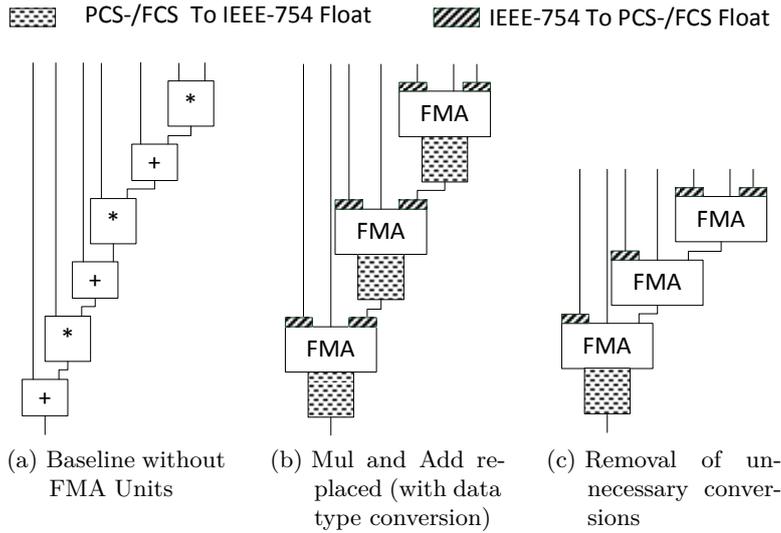


Figure 6.21: Insertion of FCS-FMA units into a CDFG during high-level synthesis steps

The datapath is initially assembled from IEEE 754 operators and scheduled (Figure 6.21a). Then, the datapath is searched for pairs of successive multiply and add operators. If they are on the critical path, the pair gets replaced by a P/FCS-FMA unit, surrounded by the required conversion logic between the CS and IEEE 754 formats. After all critical multiply/adds have been greedily replaced by FMA units (Figure 6.21b), redundant type conversions between FMA units are removed (Figure 6.21c), the entire datapath is rescheduled, and the procedure is repeated. Currently, Nymble-RS performs four iterations of this procedure.

Note that FMA units are only inserted if the FMA hardware is not already busy in the target stage. On the other hand, if the hardware is not busy in the target stage and also free in the neighbouring stages, Nymble-RS also converts single multiplications and single additions to FMA operations if they receive at least one of their input values from an FMA. The missing one of the three FMA inputs is set to "1" or "0" in this case.

6.1.11 Discussion

The FCS-FMA is superior in terms of latency, energy, size and accuracy when compared to the PCS-FMA. The use of the PCS-FMA seems only reasonable on the Virtex-5 which is not supported by the FCS-FMA.

Compared to state of the art, both FMA units offer improved latency and accuracy at cost of a higher resources and energy consumption. It depends on the applications requirements if the increased speed (up to 2.5x) is worth the higher costs. In any case, the design space is extended by the availability of these units.

The automatic insertion of the FCS-FMA during HLS is evaluated at the example of convex solvers in Section 7.7.

6.2 FAST LOOKUP-BASED DIVIDER

With the intensive use of FPGAs to accelerate classical DSP algorithms, optimization efforts have concentrated on multiplication, addition, and multiply-add units. High-speed division, which is relevant for applications such as convex optimization in model-predictive control [83], has received less attention. Since this thesis concentrates on these fields, it also tries to alleviate this deficiency.

This section will present two FPGA implementations of 1-ULP accurate double-precision division units, both based on a recent publications [47, 92] targeting the ASIC domain. The two division strategies are compared, while putting special focus on latency and area efficiency of the FPGA implementation of the mantissa division. Furthermore, FPGA-specific optimizations are applied to the superior unit to further minimize latency. The proposed dividers are then compared to state-of-the-art FPGA dividers. Most of the contributions in this section have previously been published in [7].

The implementations of the two initial divider designs were made by Daniel Schneider in his bachelor thesis [100]. However, the idea and FPGA-specific architecture as well as further optimization, error analysis and additional bug fixing originate from the author of this thesis.

6.2.1 *Goldschmidt Division for ASICs*

As stated above, the work presented here is based on two prior efforts on implementing Goldschmidt division for ASICs. In the first approach [47], which will be referred to as **TripleGS** in this section, a small lookup table *reciproc* containing the reciprocal of X is used to determine a good seed value for A_0 . Afterwards, a modified Goldschmidt algorithm is iterated two times to compute a 1-ULP accurate *single-precision* result. To reach *double-precision* accuracy, an extension using a third Goldschmidt iteration is proposed. The modified version of the Goldschmidt [77] algorithm is used to keep the required multiplications narrow (see Section 4.3.2). Figure 6.22 shows the complete computation performed for a double-precision division.

6.2.2 *Combination with Polynomial Approximation*

A second approach, which will be referred to as **PolyGS** in this work, is proposed in [92]. While the original work covers the high-speed computation of double-precision floating-point reciprocal, division, square root, and inverse square root operations, this thesis will focus on divi-

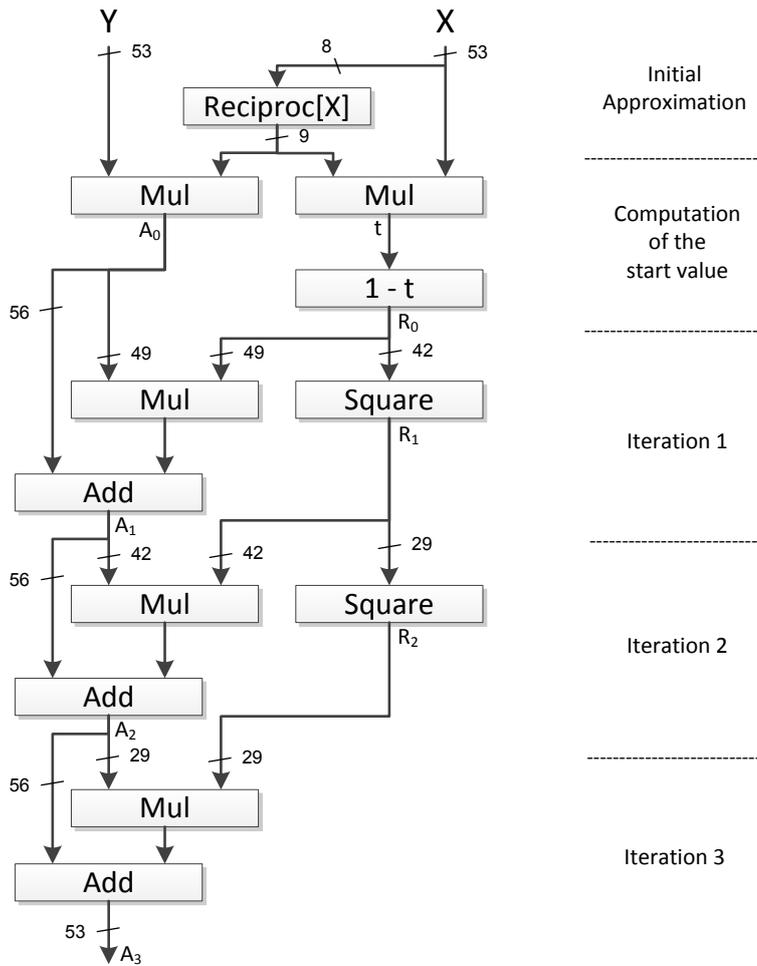


Figure 6.22: TripleGS: Compute $Q = Y/X$ using a small lookup table and three Goldschmidt iterations

sion. The structure this approach proposes for division shown in Figure 6.23. First, a piecewise second-degree polynomial approximation is performed. The first nine bits of mantissa of X are used for the table lookup of the three polynomial coefficients C_0, C_1, C_2 . The result R_d of this approximation is then used as input for a single iteration of the Goldschmidt algorithm. As R_d is already accurate to 30 bits, only a single Goldschmidt iteration is required to compute the final result. As in *TripleGS*, the modified Goldschmidt algorithm [77] is used to reduce area and latency.

As shown in Figures 6.22 and 6.23, both approaches have a chain of four multiplier/squaring units in their critical path (divisor to quotient). The next section will examine how these operations can be mapped efficiently to recent FPGA architectures.

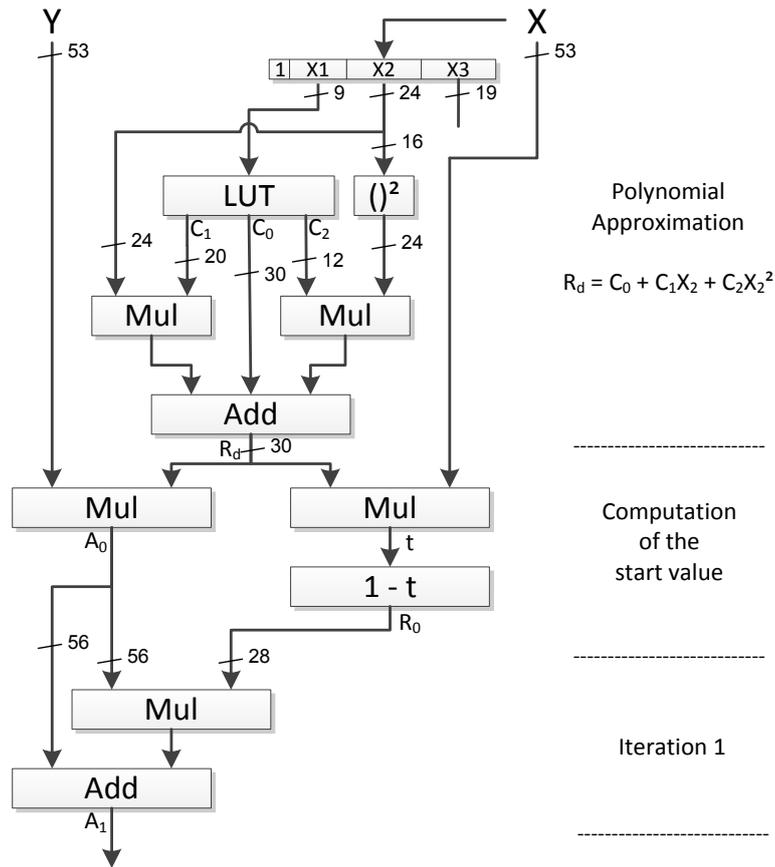


Figure 6.23: PolyGS: Compute $Q = Y/X$ using a polynomial approximation and a single Goldschmidt iteration

6.2.3 FPGA implementation of TripleGS and PolyGS

In this section, initial FPGA implementations of the TripleGS and PolyGS approaches is examined. In addition, optimizations are suggested and the accuracy of the units is evaluated. Both units target the Virtex-6/-7 architecture using BlockRAM for the lookup tables and DSP blocks for composing the multipliers and squaring units.

The small lookup table used in the **TripleGS** approach takes the eight most significant bits of X as input for addressing and returns a nine bit wide approximation of $1/X$. The total memory usage is only 2304 bits, easily fitting into a BlockRAM of type RAMB18E1.

The initial lookup is followed by three iterations of the (modified) Goldschmidt method. Although Han et al. [47] claim to aim for reduced area, and although the size of the multiplier has already been reduced by applying the modified Goldschmidt algorithm instead of the original one, some wide multiplications remain in the data path, especially in the first and second iteration (49x49, 42x42).

The traditional tiling of the multiplications into several DSP instances thus leads to a large DSP count. To some extent, this problem is due to the specifics of the target architecture, as some of the multi-

pliers are ill-matched to the 17x24 bit DSP Slices of the devices (e.g., the first two multipliers after the lookup table require input bit widths of 9x53).

Note that while Virtex-6/-7 DSP blocks offer 18x25 bit signed multiplication, only 17x24 bit are available for unsigned inputs (which are used for floating-point division). Furthermore, for FPGAs, the area savings in TripleGS due to using squaring units instead of two-operand multipliers are not as large as they would be in an ASIC design.

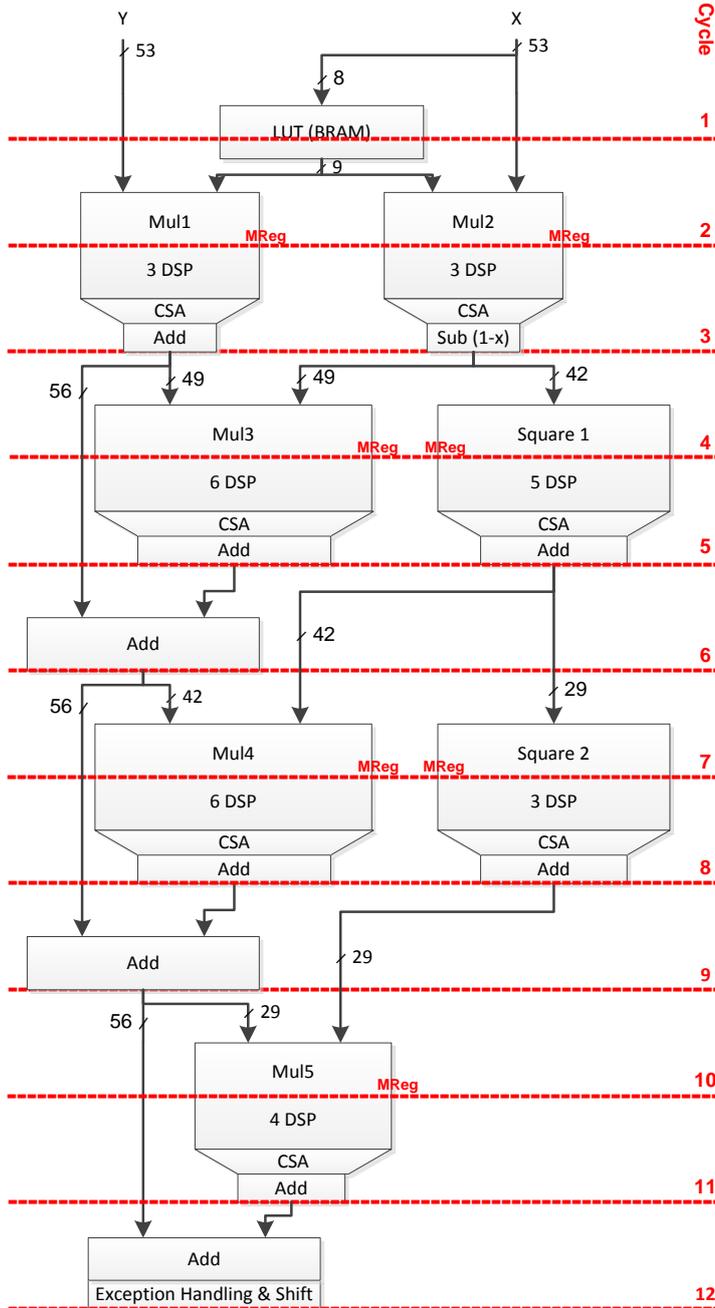


Figure 6.24: FPGA implementation of TripleGS (12 cycles, 30 DSPs)

Figure 6.24 shows the resulting division unit, pipelined for 200 MHz operation as shown by the red dashed horizontal lines. For adding more than two operands, CSAs are used to sum the DSP block outputs. In total, 30 DSP blocks have been instantiated to achieve a total latency of 12 cycles for the division.

The **PolyGS** approach [92] partitions the divisor X into three parts: $X_1 = X[51 : 43]$, $X_2 = X[42 : 19]$, $X_3 = X[18 : 0]$. X_1 is used to perform a lookup of the polynomial coefficients C_0 , C_1 , and C_2 of the piecewise polynomial approximation. The total bit width of C_0 , C_1 , and C_2 is $30 + 20 + 12 = 42$ bits, requiring $42 * 2^9$ bits of memory that can be held in single BlockRAM RAMB36E1. X_2 is then used as input value to the polynomial $C_2 \cdot X_2^2 + C_1 \cdot X_2 + C_0$. As a further optimization, it suffices to use just the uppermost 16 bits of X_2 for the squarer (please see [92] for details). The entire polynomial approximation requires only relatively narrow multipliers: $C_1 \cdot X_2$ is a 20b x 24b unit, $C_2 \cdot X_2^2$ uses 12b x 16b, and the squarer X_2^2 is organized as 16b x 16b. All of these require just a single DSP block (assisted by a few slices of logic for $C_1 \cdot X_2$). The multipliers after the approximation are wider and must be composed from several DSP blocks (up to six). In total, 20 DSP blocks are required after performing DSP tiling (using LUT-based sub-multipliers only if one operand is less than four bits wide).

Figure 6.25 shows the pipelined implementation of the second division unit. The narrow arithmetic for the polynomial approximation allows a tight pipelining of these multipliers. In summary, PolyGS requires only ten cycles, but the critical path length (four multipliers) remains similar to that of TripleGS.

Since TripleGS is both larger and slower than PolyGS already at the microarchitecture level (as well as on the layout level, as will be shown later in Section 6.2.6), further lower-level optimizations will be applied only to PolyGS.

6.2.4 FPGA specific optimization of latency and area

After deciding to use PolyGS as the baseline for further FPGA-specific optimization, the critical path of Figure 6.25 must be considered, specifically the two multipliers Mul3 and Mul4. On closer examination, it is obvious that for each of these, only a single one of the inputs, driven by the 30 bit adder, is actually timing critical. This specific configuration allows the exploitation of the DSP48E1 blocks' integrated pre-adder feature.

However, the direct approach of connecting the individual sum and carry outputs of the CSAs (see Figure 6.25 cycle 4) to the pre-adders led to excessive routing delays. They are avoided by replacing the CSA with a discrete conventional binary adder (computing $C_1 X_2 + C_0$, as shown in Figure 6.26), which is then connected to the pre-adders of Mul3 and Mul4.

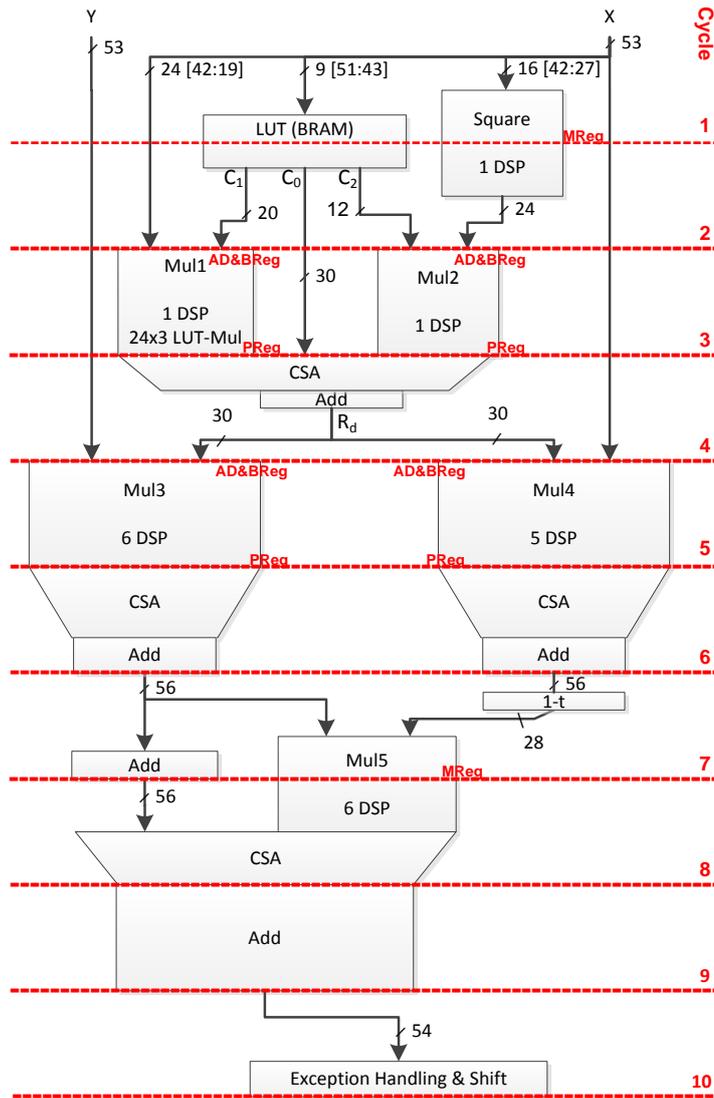


Figure 6.25: FPGA implementation of the PolyGS approach (10 cycles, 20 DSPs)

For clarity, Figure 6.26 omits the following detail: since R_d is now computed *inside* of the DSPs (using the pre-adders, as R_{d1} in Mul3 and R_{d2} in Mul4), rounding has to be performed differently than in Figure 6.25 (where it occurred before the DSPs). Rounding each of the two R_d computations individually, however, would lead to double the previous maximum rounding error: the two paths through Mul3 and Mul4 converge later and accumulate each of their individual rounding errors. As a solution, R_{d1} is rounded, R_{d2} is truncated, and a condition bit is asserted if $R_{d1} - \text{round}(R_{d1}) + R_{d2} - \text{trunc}(R_{d2}) > 0.5$, i.e., an addition of **1** is required to correctly round *up*. This bit is evaluated in the small LUT-based sub-multipliers assisting the DSPs in Mul3 and Mul4 and leads to the addition of the extra 1 if required for rounding.

Another cause for delay is the large addition required to compute the 56-bit result of Mul3 at the end of Cycle 6. In contrast to the

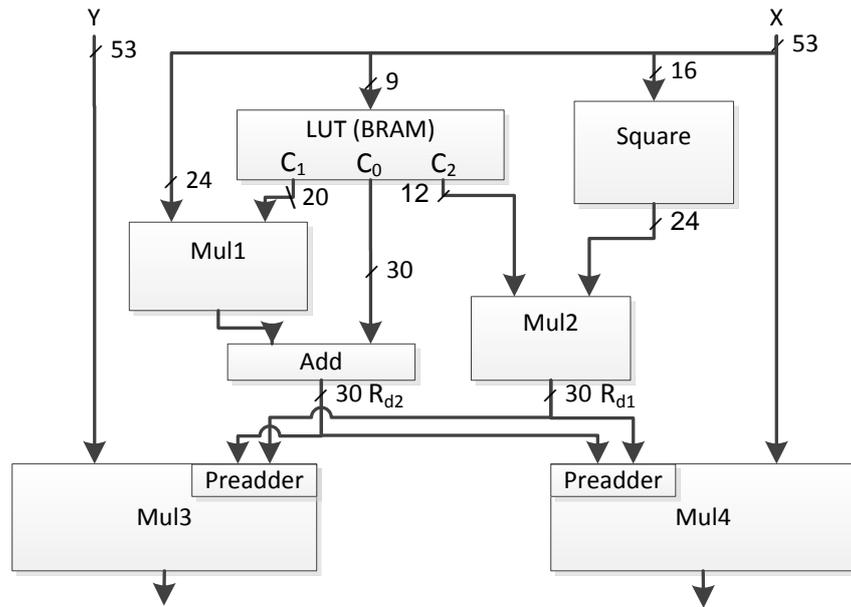


Figure 6.26: Using the DSP pre-adders in Mul3 and Mul4

final addition of the beginning of Cycle 9 (required for an IEEE 754 compatible result in binary representation), internal additions can be partitioned into narrower (and thus faster) operations using Carry Save Arithmetic. Here, the result of Mul3 is not completely computed in binary, but in a PCS representation with extra carry bits inserted at bit positions 30 and 42. This breaks the original 56b computation into three narrower additions that can run in parallel, yielding a result consisting of 56 bits plus two carry bits. The carry bits will be handled separately in the LUT-based sub-multipliers of Mul5. The widening of the result of Mul4 from 28 to 29 bits will be explained below.

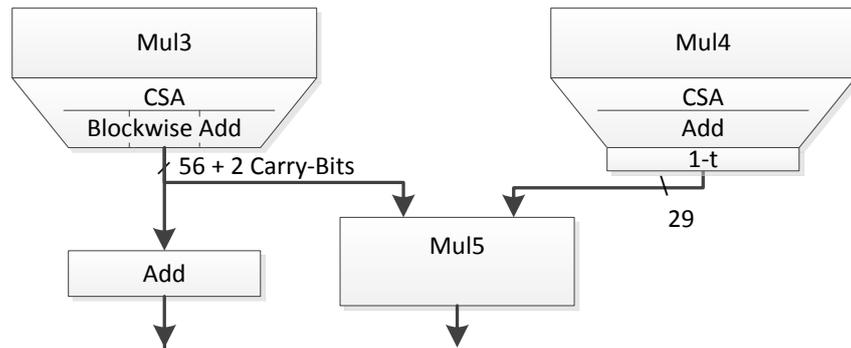


Figure 6.27: Using a partial carry save format for the result of Mul3

Based on the work of [15], the multiplier design and tiling can be re-worked to show better results in both performance and area consumption. As stated earlier, truncated multipliers reduce resources, delay, or power consumption. In the division unit under discussion, the bit width of the multiplier output is always much smaller than the sum of its in-

put bit widths (see Figure 6.25). This indicates that the full output width is not required in the next step. All multipliers are candidates for a truncated multiplication. However, Mul2 and the squaring unit are already smaller than a single DSP and therefore not considered.

Using truncated multipliers generally increases the maximum error of an arithmetic unit. However, to reach 1-ULP accuracy at the final result, it is required to carefully compensate for the accuracy loss due to tiling. This is achieved by selectively increasing the width of some operations. The error introduced by rounding the result of Mul4 to 56 bits has been shown to be the most significant error term [15]. To compensate, its bit width was increased here to 57 bits (56 fractional bits), reducing the rounding error at Mul4 by 50%. Also according to [15], the most significant 29 bits of $1-\text{Mul4}$ are known to be all zero (or all one for a negative number), so only $57 - 29 + 1 = 29$ bits are actually required for the two's complement representation of $1-\text{Mul4}$.

With these constraints, the tiling shown in Figure 6.28 can be used for the truncated multiplications Mul1, Mul3, Mul4, and Mul5. Note that empty areas in the multiplication rectangle represent the truncated parts. As another measure to compensate for the truncation error, half of the maximum result of the truncated areas is added to the result. E.g., if a 2×5 bit multiplier is truncated, $11_b \cdot 11111_b / 2 = 101110_b$ is added to the product as compensation. This addition is performed either in the adder tree of the multiplication, or using an empty C-input of a DSP block.

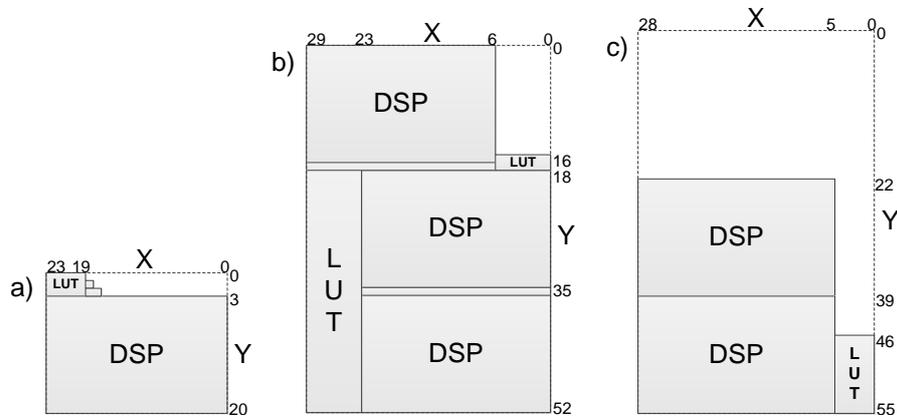


Figure 6.28: Truncated tiling for multipliers Mul1(a), Mul3+Mul4(b) and Mul5(c)

Mul1 is a 20×24 multiplier with a maximum truncation error as shown in Figure 6.29. Interpreted as an integer, the sum would have a value of $10111111111111111111001_b$. However, since the 44 bit result of Mul1 is actually a fixed-point number less than one (due to $X_2 < 2^{-9} \wedge C_1 < 1$, details in [92]), the introduced error is quite small and does not lead to a violation of the 1-ULP accuracy requirement (see Section 6.2.5 for an error analysis).

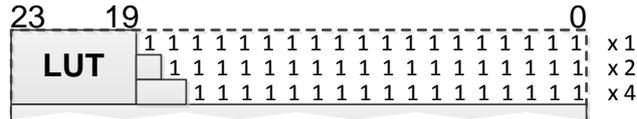


Figure 6.29: Maximum error scenario for truncated multiplication in Mul1

Mul3 (and Mul4) are 30x53 wide. In both inputs, all but one bit are fractional bits, leading to a full precision result with 83 bits, of which 81 bits are fractional bits. Furthermore, the highest bit of the result is known to be zero because R_d is known to be less or equal to one. The remaining 82 result bits are rounded to 56 (57, respectively) bits so the last 26 (25) bits are dropped after rounding. A 6x16 bit wide part of the computation is truncated with a maximum truncation error of

$$(2^6 - 1) \cdot (2^{16} - 1) < 2^{22}.$$

To compare the errors introduced by rounding and truncated multiplication, the maximum truncation error (after the compensation addition described above) is 2^4 (2^3 , respectively) times *smaller* than the error due to rounding. Also, note that for Mul4 the large LUT multiplier shown in Figure 6.28(b) is partially removed by logic optimization, since the higher 28 bits of the subsequent computation of $1 - t$ (in Cycle 7 of Figure 6.25) are not used.

Mul5 has been changed from 56x28 to 56x29 bits due to the modification described above. However, the 29 bit input was originally 57 bits wide, but had the first 28 bits removed (see discussion above). The result could therefore be regarded as a $57 + 56 = 113$ bit wide value, of which 111 bits are fractional bits. However, the final result of the mantissa result just requires the 53 fractional bits as defined by IEEE 754 double-precision (in the worst case, if $Y/X < 1$). Thus, rounding for this final computation can remove $111 - 53 = 58$ bits. Truncation is performed on Mul5 as shown in Figure 6.28(c) removing large parts of the computation. The error introduced by the truncation shown is less than 2^{52} :

$$(2^{29} - 1) \cdot (2^{22} - 1) + (2^5 - 1) \cdot (2^{24} - 1) \cdot 2^{22} < 2^{52}$$

After the compensation addition, the error due to truncated multiplication is 2^6 times smaller than the maximum error due to the rounding of the final result.

Figure 6.30 shows the complete division unit optimized using these measures, named **PolyGSopt**. Its latency is reduced to eight clock cycles at 200 MHz operation frequency. Of the 20 DSP blocks used in the first implementation, only 11 are remaining here.

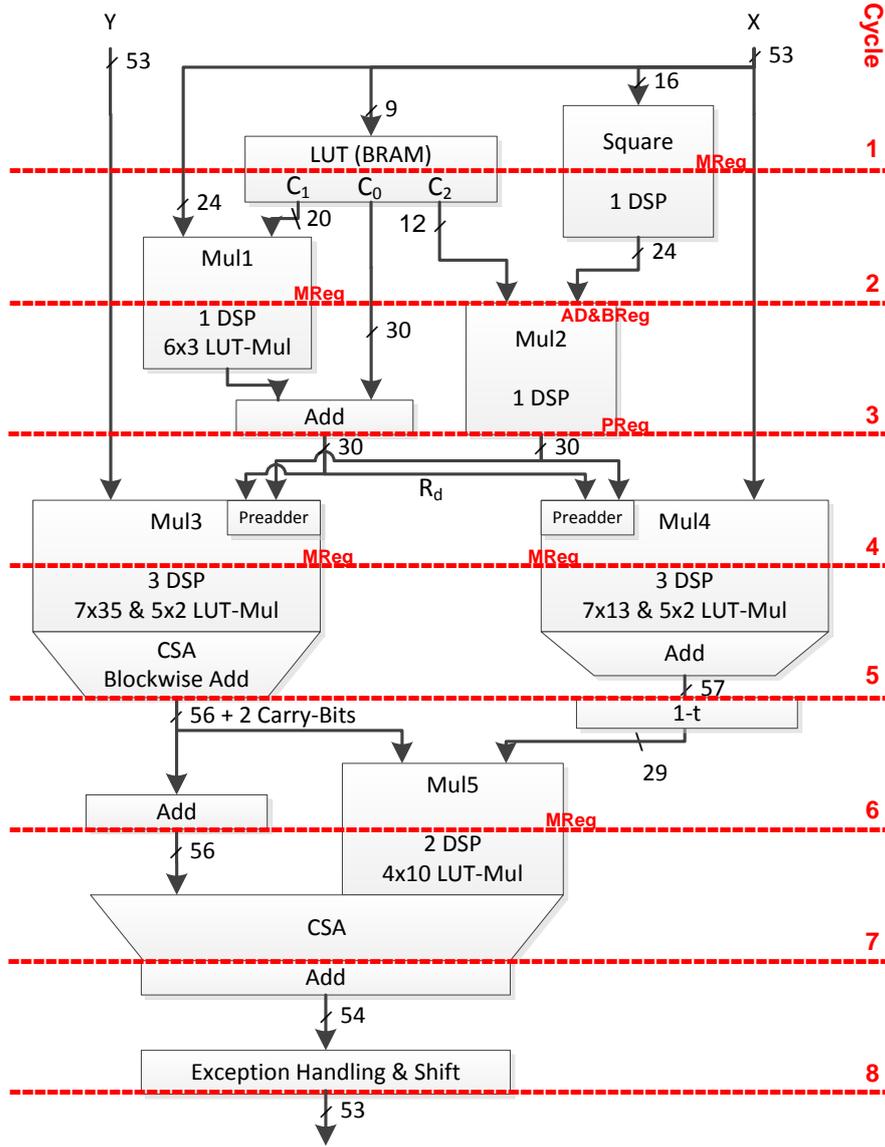


Figure 6.30: PolyGSopt implementation with reduced latency and area (8 cycles, 11 DSPs)

6.2.5 Error Analysis

As the use of truncated multipliers increases the maximum error, it is necessary to perform a new error analysis for the **PolyGSopt** implementation. A maximum error of 1-ULP in a double-precision floating-point mantissa implies an upper bound of 2^{-52} . The result of the mantissa division will be greater than 0.5 and less than 2.0, thus requiring a one bit normalization shift for results smaller than 1.0. Here, maximum error must be smaller or equal to 2^{-53} to ensure an 1-ULP accurate final result.

The overall maximum error ϵ_Z is composed of the error of the method (minimax approximation and Goldschmidt iteration), and the error introduced by using finite precision arithmetic in the computation.

$$\epsilon_Z = \epsilon_{method} + \epsilon_{comp}.$$

According to Pineiro et al. [92], the error of the method applying a single Goldschmidt iteration is

$$\epsilon_{method} = X \cdot Y \cdot \epsilon_{Rd}^2.$$

This also applies to the modified Goldschmidt method. The computation error ϵ_{comp} is calculated as

$$\begin{aligned} \epsilon_{comp} = & \epsilon_{M3} + Y \cdot Rd \cdot \epsilon_{M4} + Y \cdot \epsilon_{M4} \cdot \epsilon_{Rd} \\ & + X \cdot \epsilon_{M3} \epsilon_{Rd} \qquad \qquad \qquad + \epsilon_{M3} \cdot \epsilon_{M4} + \epsilon_{M5Z} \end{aligned}$$

with ϵ_{M5Z} denoting the error introduced by final rounding to the 54 bit result and truncated multiplication in `Mul5` and ϵ_{Mx} denoting the error introduced by rounding and truncated multiplication in multiplier x .

For the mantissa division, there are 2^{52+52} possible inputs, so a complete simulation of all possible inputs is generally not practical. However, the computation of R_d only depends on X_1 and X_2 , resulting in only 2^{9+24} possible combinations. The maximum error ϵ_{Rd} can therefore be determined experimentally. The computed value of R_d must be compared to the exact value $1/X$ with $X = \{X_1, X_2, X_3\}$. For the comparison, X_3 must be set both to all **1s** and all **0s** to maximize the error. As the maximum error decreases with growing X , the value of $X \cdot \epsilon_{Rd}^2$ was also determined experimentally. This resulted in the following upper bounds for the 8-cycle division unit shown in Figure 6.30:

$$\begin{aligned} \epsilon_{Rd} & \leq 2^{-28.969} \\ X \cdot \epsilon_{Rd} & \leq 2^{-28.331} \\ X \cdot \epsilon_{Rd}^2 & \leq 2^{-57.644} \end{aligned}$$

With $Y < 2$, an upper bound for the method error can be computed:

$$\epsilon_{method} \leq 2^{-56.644}$$

The computation error at each multiplier consists of the rounding error and the (compensated) error of the truncated multiplication. The result of `Mul3` is 56 bits wide of which 55 bits are fractional bits. Rounding to nearest therefore causes an maximum error of 2^{-56} . Furthermore,

the error introduced by truncated multiplication was shown to be 2^4 times smaller.

$$\epsilon_{M3} \leq 2^{-56} + 2^{-60}$$

The result of Mul4 is 57 bits wide of which 56 bits are fractional bits. However, the leading 28 bits are not used because they are known to be all zero after subtraction. Rounding to nearest causes a maximum error of 2^{-57} , while the error introduced by truncated multiplication is the same as at Mul3.

$$\epsilon_{M4} \leq 2^{-57} + 2^{-60}$$

The result of Mul5 is passed to the final addition without rounding. But after the addition, it is rounded to 54 bits in worst case (if the result is smaller than 1). 53 of these 54 bits are fractional bits, so the error of rounding to nearest is 2^{-54} . Furthermore, the truncated multiplication error is known to be 2^6 times smaller:

$$\epsilon_{M5Z} \leq 2^{-54} + 2^{-60}$$

Substituting these inequalities into the inequality for ϵ_{comp} results in

$$\begin{aligned} \epsilon_{comp} \leq & 2^{-56} + 2^{-60} + 2 \cdot (2^{-57} + 2^{-60}) \\ & + 2 \cdot (2^{-57} + 2^{-60}) \cdot 2^{-28.969} \\ & + 2 \cdot (2^{-56} + 2^{-60}) \cdot 2^{-28.969} \\ & + (2^{-56} + 2^{-60}) \cdot (2^{-57} + 2^{-60}) \\ & + 2^{-54} + 2^{-60} \end{aligned}$$

$$\epsilon_{comp} \leq 2^{-54} + 2^{-55} + 2^{-58} + 2^{-83.269}$$

Further substitution into the inequality for ϵ_Z proves that the divider is actually accurate within 1-ULP:

$$\epsilon_Z \leq 2^{-54} + 2^{-55} + 2^{-56.644} + 2^{-58} + 2^{-83.269} < 2^{-53}$$

6.2.6 *Synthesis Results and Comparison to State of the Art*

The division units presented in this paper have been successfully tested using randomized mantissas. The results were compared to a 75 bit Xilinx IP Core divider and did not violate the 1-ULP error constraint.

Table 6.3 shows the synthesis results for the division units presented for a Virtex-6 device, specifically a XC6VLX240T-1. The speed grade used in [38] is unclear, as it is not stated in paper and an E-Mail regarding this issue was not answered. The optimized implementation of the polynomial approximation followed by a single Goldschmidt iteration clearly outperforms all other division units. The Wall Clock

Time (WCT) of a single division is reduced by 62% compared to the Xilinx IP Core and by 58% compared to the VFLOAT division unit.

For comparison, the divider presented in [90] is also included in Table 6.3, although it was not implemented for Virtex 6 but for Altera Stratix V. It also uses polynomial approximation followed by a single Newton-Raphson iteration. Its similar structure thus makes it an interesting competitor. The divider reaches much higher clock frequencies than the prior work on Virtex-6 dividers, even though the Virtex-6 and Stratix V generally perform comparably in practice for optimized designs. The proposed divider implementation manages to outperform even that very fast unit, reaching a 40% shorter wall clock time.

For completeness, a mapping of the proposed divider implementation to Zynq XC7Z045-2 (speed grade -2) and Virtex-5 XC5VFX200T-1 devices is included, as these two FPGAs are used in the supported target platforms of Nymbler-S. The Virtex-5 lacks the DSP pre-adder capability and can thus only support the 10-cycle PolyGS, but not the faster 8-cycle PolyGSopt implementation.

Table 6.3: Performance of placed & routed division units

Method	Rounding Accuracy	Device	Latency [cycles]	f_{max} [MHz]	WCT [ns]	Resources
Xilinx IP Core [38]	CR	Virtex-6	20	192	104	3216 LUTs, 2035 FF, 0 BlockRAM, 0 DSP
VFLOAT [32] (results from [38])	FR	Virtex-6	14	148	95	6957 LUTs, 934 FF, 0 BlockRAM, 0 DSP
FloPoCo 2.5.0 FPDIV	CR	Virtex-6	17	136	125	4419 LUTs, 2509 FF, 0 BlockRAM, 0 DSP
TripleGS	FR	Virtex-6	12	201	60	1474 LUTs, 1294 FF, 1 BlockRAM, 30 DSP
PolyGS	FR	Virtex-6	10	230	44	1297 LUTs, 1244 FF, 1 BlockRAM, 20 DSP
PolyGSopt	FR	Virtex-6	8	202	40	1525 LUTs, 1094 FF, 1 BlockRAM, 11 DSP
Polyn. Appr. + Newton-Raphson [90]	FR	Stratix V	18	268	67	887 ALUTs, 823 FF, 2 M20K-RAM, 9 DSP
FloPoCo Radix-4 [90]	CR	Stratix V	36	219	164	5209 ALUTs, 5473 FF, 0 M20K-RAM, 0 DSP
PolyGS	FR	Virtex-5	10	210	48	1251 LUTs, 1244 FF, 1 BlockRAM, 20 DSP
PolyGS	FR	Zynq (-2)	10	297	34	1101 LUTs, 1265 FF, 1 BlockRAM, 20 DSP
PolyGSopt	FR	Zynq (-2)	8	268	29	1493 LUTs, 1064 FF, 1 BlockRAM, 20 DSP
Xilinx IP Core	CR	Zynq (-2)	29	222	131	3225 LUTs, 3075 FF, 0 BlockRAM, 0 DSP
Xilinx IP Core	CR	Zynq (-2)	43	224	192	3262 LUTs, 3096 FF, 0 BlockRAM, 0 DSP
Xilinx IP Core	CR	Zynq (-2)	57	463	123	3264 LUTs, 6025 FF, 0 BlockRAM, 0 DSP

6.3 HARDWARE UNITS FOR MATH LIBRARY FUNCTIONS

Some floating-point applications contain calls to the standard library of the C programming language, especially to the math library. At the start of this work, the original Nymble compiler [4] could not translate most of these calls. In this work support for the functions *log*, *exp*, *pow*, *floor*, and *ceil* calls was added. The function calls are translated into the corresponding floating-point operation which is then integrated into the CDFG. The Nymble-RS back-end then instantiates an appropriate hardware unit for the operation(s).

The *log* function is realized by an instance of the Xilinx CoreGen *log* core. The core is configured to use 34 cycles, which ensures operation above 200 MHz.

For the *exp* function, the FloPoCo *exp* core is used [33]. However, since FloPoCo is using a non-IEEE 754 conformant number format, input and output must be converted from and to IEEE 754 format. 200 MHz operation thus requires a pipeline depth of 26 with an additional two cycles required for the format conversion.

Finally, $pow(x,y)$, is realized as $exp(log(x) * y)$. While this introduces a larger error (discussed in Section 8.6), and is thus not fully IEEE754-compliant, it allows resource-sharing with other *exp*, *log* and *dmul* operations and serves to fulfil the aim of generating area-efficient hardware.

Since no realization for the *floor/ceil* functions was available to us, a custom implementation was developed.

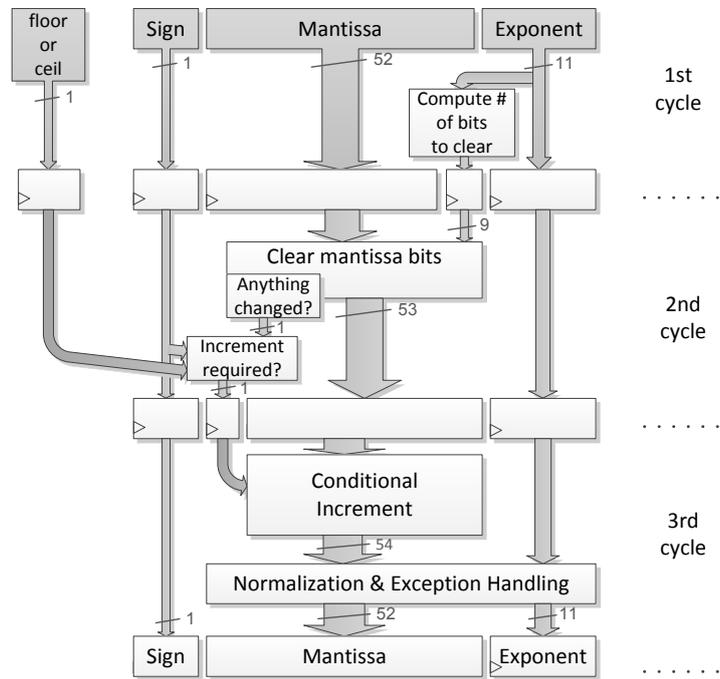


Figure 6.31: Newly developed *floor* operator

As shown in Figure 6.31, it operates as follows: the number of mantissa bits to the right of the binary point is computed in the first cycle.

The second cycle clears the required number of mantissa bits, but tracks if any of these were '1' before. This latter information is used in the third cycle to conditionally increment the mantissa. Each of these operations has only a small delay and allows operation faster than 200 MHz. The first stage has an even shorter delay and can be chained with a preceding multiplexer in resource-shared architectures.

CASE STUDY: CONVEX OPTIMIZATION SOLVERS

For many types of mathematical optimizations problems software solvers exist or can be auto-generated today. The automatic generation of software implementations for problem-specific convex solvers has been the subject of prior research [45, 76]. Some of these problem-specific solvers, however, may require long computation time for practically relevant problem instances, despite executing on fast desktop-class processors (far exceeding the power budget of many embedded scenarios).

In order to overcome the issues complicating the use of convex solvers in embedded systems, the use of custom-compiled problem-specific hardware accelerators is examined. In this chapter, Nymble-RS is shown to accelerate five different CVXGEN generated solver programs [78]. We compare our approach with other academic and industrial “C”-based HLS systems as well as with the software performance achieved on high-performance embedded hard-core processors (e.g., 800 MHz ARM Cortex-A9 including a hardwired double-precision FPU). Furthermore, the five solvers are also used for the evaluation of different techniques proposed in this thesis.

Large parts of this chapter have been previously published in [8]. Parts of system-level evaluation of the division and FMA units have been published in [6] and [7], but the results presented here are updated to match the new target platform (Zynq-7000).

7.1 TOOL FLOW AND DOMAIN-SPECIFIC OPTIMIZATIONS

The tool flow used is sketched in Figure 7.1. The convex optimization problem is described in a purely mathematical fashion using a domain-specific language. The description is then read using CVXGEN which generates a C implementation of a problem-specific solver (see Section 2.3 for details). This C code is subject to domain-specific code transformation and then used as input for Nymble-RS.

Two domain-specific optimizations are performed: first, to enable the use of non-field sensitive alias analysis (such that comes with LLVM), the global structure containing the arrays the algorithm works with is decomposed into a global variable for each member, with all accesses being transformed correspondingly (see Listing 7.1). At the same time, all arrays are marked for implementation in local BlockRAM memory.

Secondly, as the original code targets embedded CPUs, it tries hard to avoid divisions (see Listing 7.2.a). Here the condition test uses multiplication (cheaper) before performing the costly division. However, this code makes a parallel execution (by unrolling) or pipelining impossible.

Listing 7.1: Code transformation required for non-field sensitive alias analysis

```

// (a) original code

typedef struct Workspace_t {
    double h[20];
    double s_inv[20];
    double s_inv_z[20];
    double b[3];
    [...]
} Workspace;
Workspace work;

[...]

void ldl_factor(void) {
    work.d[0] = work.KKT[0];
    if (work.d[0] < 0)
        work.d[0] = settings.kkt_reg;
    else
        work.d[0] += settings.kkt_reg;
    work.d_inv[0] = 1/work.d[0];
    work.L[0] = work.KKT[1]*work.d_inv[0];
    [...]
}

// (b) decomposed structure

__attribute__((localmem)) double work_h[20];
__attribute__((localmem)) double work_s_inv[20];
__attribute__((localmem)) double work_s_inv_z[20];
__attribute__((localmem)) double work_b[3];

[...]

void ldl_factor(void) {
    work_d[0] = work_KKT[0];
    if (work_d[0] < 0)
        work_d[0] = settings.kkt_reg;
    else
        work_d[0] += settings.kkt_reg;
    work_d_inv[0] = 1/work_d[0];
    work_L[0] = work_KKT[1]*work_d_inv[0];
    [...]
}

```

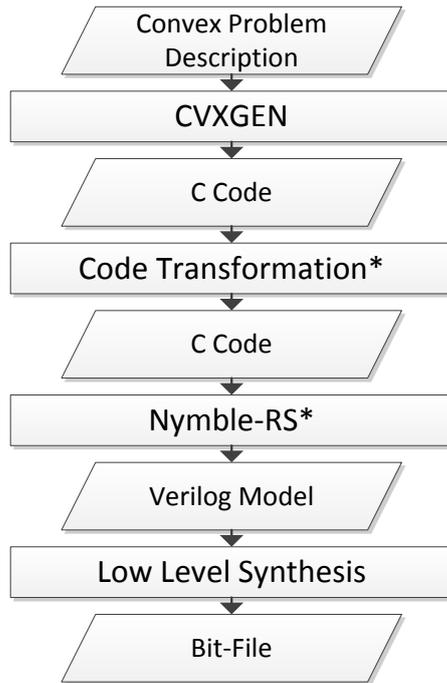


Figure 7.1: The tool chain used in this work (* = contribution of this thesis)

Listing 7.2: Transformation to canonical form of **min** operation

```

// (a) original, avoiding divisions
for (i=0; i<N; i++)
  if ( minval*a[i] < b[i] )
    minval = b[i] / a[i]

// (b) transformed into canonical min operation
for (i=0; i<N; i++)
  if ( minval < b[i] / a[i] )
    minval = b[i] / a[i]
  
```

With the domain-specific knowledge that $a[i] > 0$, this code can be transformed to the form in Listing 7.2.b. Note that for fair comparison, the other HLS tools in this study are also evaluated with this optimized code.

7.2 TEST SETUP

Four example problems provided on the CVXGEN website (SQP, SVM, Lasso, Portfolio) are translated with different problem dimensions and settings. In addition, **Stan5p**, a more complex example containing a complete model-predictive control problem for collision avoidance trajectory planning in autonomous ground vehicles [37], is used as a fifth benchmark. For the smaller examples the following dimensions were entered into CVXGEN to create the solvers with static counts of 13-57K operations (mostly FP): SQP ($m = 3, n = 10$), Lasso ($m = 100, n = 10$), SVM ($N = 20, n = 4$), Portfolio ($n = 25, m = 5$). Table 7.1 shows

Listing 7.3: Simplified structure of the key function of the solver

```

void solve(void) {
    init();
    computeStartValue();
    while ( !solution_found ) {
        doOneIteration();
    }
}

```

the number of static floating-point operations used in the generated C code of each solver.

Table 7.1: Number of operations in source code

Op	SQP	Lasso	SVM	Portfolio	Stan5P
fmul	2735	13240	3473	5916	10731
fadd	1841	12537	2640	4367	9176
fdiv	66	74	308	173	222
iadd	947	978	1653	2690	3459
cmp	137	153	189	248	320
load	6494	27934	9142	15268	28076
store	1722	2841	2773	4436	6212

Listing 7.3 shows the pseudo-code of solver core. After initialization and computing a preliminary solution, the optimum is sought in an iterative manner. With the exception of *Lasso*, the *while*-loop requires more than 90% of the CPU computation time in all of the test cases. Even in *Lasso* more than half of the execution time is spent there ($\approx 55\%$). The hardware-software co-synthesis capabilities of Nymble-RS allow to synthesize only the compute-intensive *while*-loop to hardware (saving chip area), while leaving the remainder of the algorithm in software. Nymble-RS also generates the HW/SW interfaces automatically.

This evaluation targets the Zynq platform presented in Section 5.2.5, and uses the “bare metal” software design flow. Nymble-RS currently uses LLVM 3.3 as front-end and for machine-independent optimization. Xilinx Vivado 2014.1 is used for logic synthesis, placement, and routing because newer tool version, e.g., 2017.2 and 2016.2, created faulty hardware in some (rare) cases. The initial evaluation uses a fixed microarchitecture of two FP adders, two FP multipliers, and one FP divider (Section 7.5 will explore other configurations).

For division, the faithful rounding PolyGSOpt divider is used (see Section 6.2). For multiplication, a three cycle faithful rounding multiplier is used which was built according to [15]. The use of faithful rounding does not adversely affect solver solution quality. Instead, when the precision is reduced, the solvers may require more iterations to find a solution. However such a behaviour was not observed with the solvers

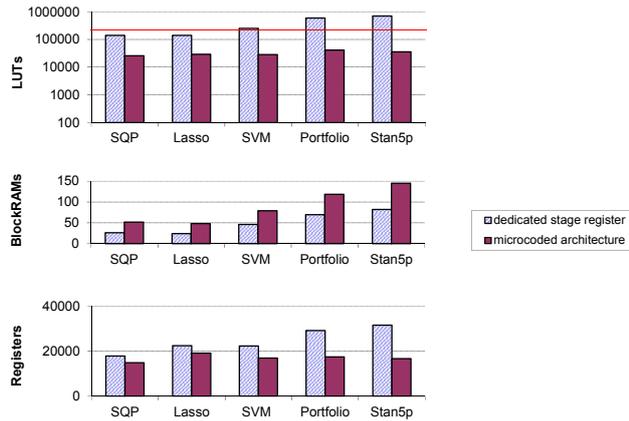


Figure 7.2: Dedicated state registers vs. microcode: Resources required

used in this test. In experiments with the variable precision floating-point library `mpfr::real` [25] this effect occurred only when the mantissa width was reduced to values below 48 bit.

7.3 MICROCODE-BASED CONTROLLERS

Figure 7.2 compares the resources required for the accelerators using conventional (dedicated state flip-flops) and the proposed microcoded controllers. The results are taken from the Vivado 2014.1 *post-synthesis* report as the conventional controllers would not even fit on the Zynq device for the more complex solver examples. Interestingly, the older Vivado 2014.1 does much better on the conventional controllers than newer Vivado versions, e.g. Vivado 2016.2 requires almost 3x the LUTs. However, even comparing against the superior LUT numbers of the older Vivado version, the area required is still about an order of magnitude higher than that for the proposed microcoded approach and exceeds the target chip resources in most cases (red line in the figure). On the other hand, the small increase of BlockRAM usage stays far below the resource limit of the target device (545). Thus, microcode-based control is a key enabler for creating accelerators for complex irregular “C” code.

7.4 HANDLING OF INTERMEDIATE VALUES

Table 7.2 examines the impact of the different intermediate result handling mechanisms discussed in Section 5.4. As baseline, a microcoded accelerator is used which employs normal registers for intermediate storage. Then, in a first step, beneficial **recomputation** of intermediates instead of storing them is allowed. Next, the **shift+recomp** column *additionally* enables the use of shift-registers together with recomputation for longer lifetimes. Finally, the **spill+recomp** column *instead* combines selective recomputation with spilling values to scratch-pad

Table 7.2: Post-synthesis area requirements for different intermediate value handling mechanisms

		base	recomp	shift+recomp	spill+recomp
Registers	SQP	41573	36615	22054	14866
	Lasso	39345	33087	27155	19128
LUTs	SQP	113190	107872	52197	26025
	Lasso	102904	95262	53033	29127
BlockRAMs	SQP	91.5	91.5	48.0	51.5
	Lasso	94.5	96	42.0	48.0

memories. Shift-registers and spilling cannot be enabled both at the same time because values can be either spilled to RAM or stored in shift registers.

The impacts are presented only for the two smaller examples SQP and Lasso, as even Nymble-RS will run out of memory on a 128 GB server when trying to process the larger solvers restricted just to **base** mode (due to an excessive number of NOP operators). Again, the results given are *post-synthesis* areas. Note that the results presented here differ from those previously published due to improvements made in Nymble-RS in mean time in scheduling (see Section 5.2.2) and in linear-scan register allocation (bug fixes).

The results show a significant improvement of spilling over the use of shift registers. Note that spilling also reduces the BlockRAM usage because fewer multiplexers require less microcode ROM which more than compensates for the BlockRAMs used as scratch pads. The logic synthesis tool automatically implements small scratch pads in LUT-RAM instead of BlockRAM.

7.5 DESIGN SPACE EXPLORATION

Nymble-RS can flexibly scale the number of hardware operators used in the solver hardware. However, more operators do not always yield a correspondingly faster accelerator, as increasing the number of operators also leads to higher interconnect demands in the FPGA, which at some point will slow down f_{\max} . Thus, Nymble-RS can be used for design space exploration to determine the best parallelism ./ frequency trade-off for each solver.

The impact of design space exploration for the test cases SQP, SVM and Stan5p is shown in Table 7.3. The reported WCT includes hardware and software parts, as well as data transfers between hardware and software. When increasing the number of FP adders and multipliers, the schedules become shorter (due to exploiting more instruction level parallelism), but f_{\max} slows down slightly (due to interconnect delays).

Table 7.3: Design space exploration. II=Initiation interval, WCT=Wall Clock Time

Test Case	# Mul/Add	# LUTs	# FFs	# BlockRAMs	# DSPs	# Schedule Length	II	f_{\max} (MHz)	WCT (ms)
SQP	1/1	20795 (10%)	15991 (4%)	44.5 (8%)	16 (2%)	3357	3350	213.6	0.140
	2/2	27651 (13%)	17170 (4%)	52.0 (10%)	21 (2%)	1945	1938	213.4	0.094
	3/3	35978 (16%)	18332 (4%)	60.5 (11%)	26 (3%)	1758	1751	201.0	0.091
	4/4	40021 (18%)	19272 (4%)	66.5 (12%)	31 (3%)	1747	1740	193.9	0.093
	5/5	45868 (21%)	20245 (5%)	67.5 (12%)	36 (4%)	1743	1736	177.1	0.098
SVM	3/3	41489 (19%)	20935 (5%)	99.0 (18%)	26 (3%)	2494	2488	184.8	0.129
	4/4	49066 (22%)	22617 (5%)	109.0 (20%)	31 (3%)	2315	2309	193.6	0.115
	5/5	56963 (26%)	24638 (6%)	102.0 (19%)	36 (4%)	2190	2184	185.9	0.114
	6/6	62416 (29%)	25728 (6%)	105.5 (19%)	41 (4%)	2171	2165	164.5	0.121
Stan5p	2/2	36089 (17%)	19048 (4%)	146.0 (27%)	21 (2%)	7258	7251	203.7	0.386
	3/3	44696 (20%)	20210 (5%)	189.5 (35%)	26 (3%)	5664	5657	174.0	0.368
	4/4	55036 (25%)	22018 (5%)	225.5 (41%)	31 (3%)	5255	5248	166.2	0.363
	5/5	63724 (29%)	23219 (5%)	235.0 (43%)	36 (4%)	5245	5238	168.3	0.357

The schedule length does not shrink in inverse proportion to the number of FP-units due to data dependencies that limit the instruction level parallelism. Especially for the small solver, using four (or more) units has only small impact on the schedule length.

Note that the results shown here differ slightly from those published in [8]. The maximum operation frequency was increased by adding an additional pipeline register to the microcode ROMs. Furthermore, loads from scratch-pad memory are no longer executed conditionally. Instead, they are executed always - no matter if the result is required or not. This way the evaluation of the condition can be saved or done later, which reduces the schedule length and the II.

7.6 EVALUATION OF 1ULP-DIVISION UNITS

The impact of the proposed division units on the solver performance is evaluated by synthesizing all test cases with 3 different units. Up to this point, the PolyGSOpt design presented in Section 6.2 was used for divisions in this section (single instance). Now it gets replaced by the PolyGS divider and a Xilinx divider IP core configured to 57 cycles. Table 7.4 shows division units used for the comparison. Note that for this comparison, the use of three adders and three multipliers was allowed.

Table 7.4: Division Units Compared

Divider	Latency
PolyGSOpt	8 cycles
PolyGS	10 cycles
Xilinx IP	57 cycles

Figure 7.3 shows the II achieved by Nymble-RS for the three different division units. While the use of the Xilinx IP causes a 9 to 52% higher schedule length compared to the division units presented in this work, the differences between PolyGS and PolyGSOpt are marginal. In two cases, PolyGS even reaches the lower II which is caused by the scheduling being a heuristic method that delivers only near-optimal results.

Figure 7.4 shows the maximum operation frequency achieved by the placed-and-routed designs. Although all units allow frequencies above 260 MHz when synthesized individually (see Section 6.2.6), PolyGSOpt and the Xilinx IP cause a significant drop in the maximum operation frequency in some test cases. This makes PolyGS the best choice for all test cases except SQP.

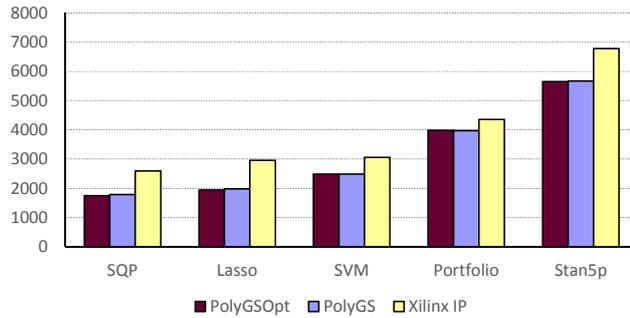


Figure 7.3: Initiation interval achieved by Nymble-RS using PolyGS, PolyGSOpt and the Xilinx divider IP

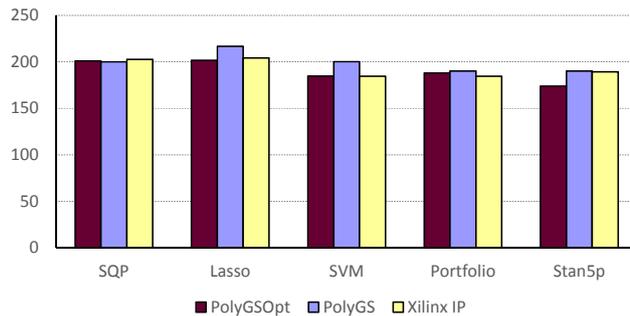


Figure 7.4: Maximum operation frequency (MHz) of the Nymble-RS results using PolyGS, PolyGSOpt and the Xilinx divider IP

7.7 EVALUATION OF THE FCS-FMA UNIT

In this section the FCS-FMA unit proposed in Section 6.1 is automatically inserted into the generated hardware by Nymble-RS and the performance of the hardware kernels with and without the FMA is compared. Note that the PCS-FMA is not included at this point as the FCS-FMA was already shown to be superior in latency and size in Section 6.1.9.

In the first step, the maximum speed-up reachable on parts of the convex solver code shall be investigated. The solver loop contains two calls to a function named `ldl_solve()` in each iteration. The function basically consists of floating-point additions, subtractions, and multiplications.

To visualize the maximum impact that is *theoretically* reachable, Nymble-RS is executed *without resource limitations*. For this comparison, a 4-cycles addition unit and a 5-cycle (IEEE 754 conform rounding) multiplication unit are used. Each solver is synthesized with and without the automatic insertion of the FMA. Figure 7.5 shows the resulting schedule length. A reduction of schedule lengths by up to 62% is reached in this case.

The `ldl_solve()` function is only one of multiple functions called within the solver loop. However, only translating parts of the solver loop to hardware requires hardware-software communication in each iteration.

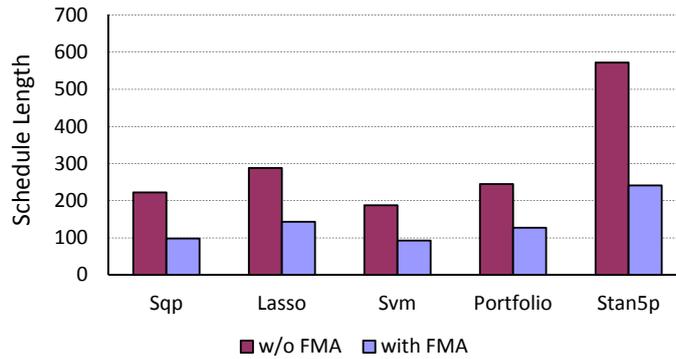


Figure 7.5: `ldl_solve()` schedule length in absence of resource limitations with and without FMA insertion

To determine what is feasible in practice, the FMA is integrated into the test setup used in the previous sections of this chapter. That is, the whole solver loop is translated to hardware and the 4+1 cycle addition as well as the faithful rounding three cycle FR multiplication unit are used (if addition or multiplication not replaced by the FMA).

In early experiments a significant drop in f_{max} was observed. In almost all cases, the critical path was located at the B input of the FMA. Large multiplexers were identified as the cause of this delay (up to 40-1). They exist because the B input is defined in IEEE 754 format and can potentially be connected to all operations with IEEE 754 conforming results, e.g., adders, multipliers, main memory load and scratch pad load operations. However, as discussed in Section 6.1, the B input is not expected to be in the critical path of the CDFG. Therefore, Nymble-RS was modified to add an additional register at the B-input. The additional register caused a slight increase of the II (0-2%) but the maximum operation frequency was increased by more than 5% in all test cases.

Table 7.5 compares the fastest result of the design space exploration to configurations with one or two FMA Units. In these configurations, the hardware kernels that use FMA units can achieve only a moderate II reduction. However, the drop in maximum operation frequency is also moderate so the configurations using FMA units can still reduce the WCT by 1-7%. Furthermore, in two test cases the FMA-enabled configurations can reach higher or equal WCT at a reduced LUT usage. This can be beneficial when targeting smaller FPGAs, because the generated designs use up to 29% of the available LUTs, while DSP usage is below 5%.

Note that this comparison is not completely fair as the faithful rounding multiplier is less accurate than the IEEE 754 standard demands while the FCS-FMA was shown to exceed the *average* accuracy of IEEE 754 double-precision by more than an order of magnitude.

Table 7.5: Impact of FMA insertion on the hardware kernel performance.
II=Initiation interval, WCT=Wall Clock Time

Test Case	# Mul/Add	# FMA	# LUTs	# DSPs	II	f_{\max} (MHz)	WCT (ms)
SQP	3/3	0	35978	26	1751	201.0	0.091
	3/3	1	49069	38	1613	188.0	0.089
	2/2	2	56199	45	1591	187.8	0.088
SVM	5/5	0	56963	36	2184	185.9	0.114
	3/3	1	52566	38	2203	185.8	0.114
	2/2	2	57856	45	2162	186.8	0.113
Stan5p	5/5	0	63724	36	5238	168.3	0.357
	3/3	1	55657	38	5184	174.3	0.346
	2/2	2	60233	45	5011	177.8	0.331

7.8 COMPARISON TO STATE-OF-THE-ART HIGH-LEVEL SYNTHESIS TOOLS

For comparison with Nymble-RS, the *smallest* test case SQP is synthesized using a leading industrial HLS system for Xilinx devices* and using the open-source LegUp [23] compiler. Both tools support FP operations and pointers.

LegUp 5.1 has become a commercial product and may no longer be used for public benchmark comparisons. Earlier tests with LegUp 3.0 had failed due to its lack of a configurable upper limit for the number of floating-point units. While LegUp 4.0, which does allow setting such a limit to force resource-sharing, still crashes in HW/SW co-compilation mode (called hybrid mode), it is able to progress further when using its pure hardware flow. Since LegUp 4.0 does not fully support Xilinx devices, it was chosen to target the Intel Cyclone V family of chips for comparison. As LegUp interprets the constraints on FP units on a per-function basis, the limits are set to “one each of Mul/Add/Div”, which (over the entire benchmark) still results in many more units being instantiated than with Nymble-RS (which applies the limits globally). To avoid this area explosion inlining was tested, using both the `static inline` keywords, as well as increasing the inlining threshold in LegUp’s LLVM component. Even after this significant manual optimization effort, no result smaller than 122,737 LUTs and 186,929 registers was achieved for the smallest test case SQP. This is about 6x and 11x *larger* than the hardware generated by Nymble-RS given the same FP unit constraints. The LegUp-generated design did not fit even on the largest Cyclone V, so no maximum clock frequency can be given

* Anonymized due to licensing terms.

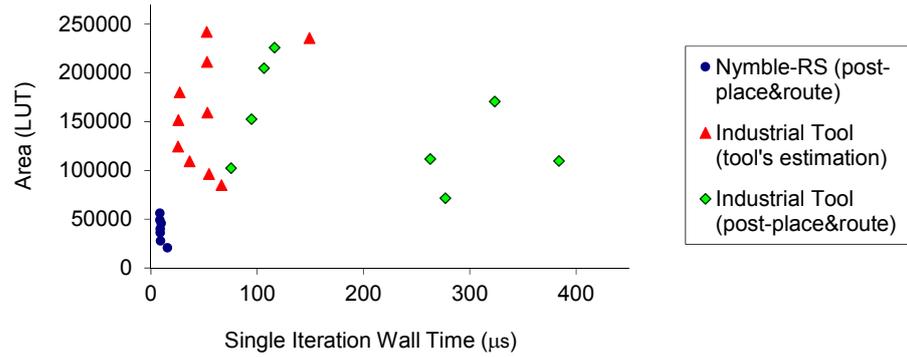


Figure 7.6: Quality comparison of results of Nymble-RS and the industrial HLS tool

here. From simulation, it was determined that it would require 232K clock cycles which is more than 7x of the slowest (=smallest, < 10% of Z7045) SQP by Nymble-RS (31K cycles).

The version of the industrial tool released in 07/2016 was able to translate the benchmark “C” code into hardware (earlier versions simply failed). As with LegUp, significant manual effort was performed to tune the code for the industrial tool: this included runs with and without directives/pragmas for pipelining, loop unrolling, inlining as well as constraining the microarchitecture to 1...2 FP units each and allowing numerically unsafe mathematical optimizations.

Figure 7.6 compares the post-HLS estimates as well as post-place & route (post-P&R) results of the industrial tool to the actual post-P&R results for Nymble-RS. The runtime of a single iteration is computed as the II multiplied by the clock period (estimated and actual). Note the large discrepancy between the post-HLS estimates and the actual post-P&R delays for the industrial tool.

Despite f_{\max} estimates of 140...164 MHz, no hardware created by the industrial tool could exceed 65 MHz on the target Z7045 SoC after P&R, with many results being unroutable or not even fitting on the Zynq device. For completeness, the industrial tool was also allowed to target a large Virtex 7 device, which resulted in post-P&R clock frequencies of 16...84 MHz. Here results requiring fewer clock cycles often had difficulties reaching a high f_{\max} .

Remember that these experiments were performed on the *smallest* of the solver codes. Attempts to run the larger solvers through hardware synthesis either had failures in the flow or resulted in *even larger* area growth and slowdown for LegUp and the industrial tool compared to Nymble-RS.

The comparison was repeated with the version released in 12/2017 (see Figure 7.7). While the estimates now promise a lower resource utilization, such an improvement cannot be observed when looking at the placed & routed results. In some cases, better maximum operation frequencies are reached when using the newer tool version (up to 129

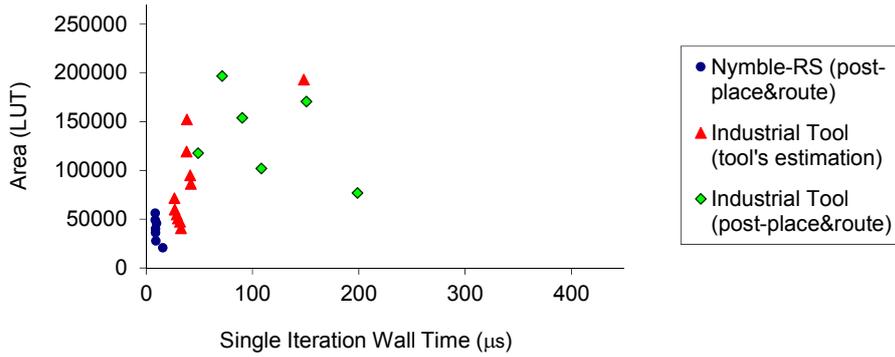


Figure 7.7: Comparison of results of Nymble-RS and the newest version of the industrial HLS tool

MHz on the Zynq). However, only designs with large II could reach high operation frequencies while designs with shorter II only reached a low f_{\max} or were completely unroutable. Therefore, optimizing the HLS settings to achieve a lower II or schedule length did not result in a faster hardware in most cases.

Although the newer tool version could significantly improve the quality of the generated hardware, the smallest hardware generated by the industrial tool is still 3.7x larger than the smallest hardware generated by Nymble-RS. Furthermore, the fastest result of the industrial tool needs 5.8x more time per iteration than the fastest Nymble-RS result.

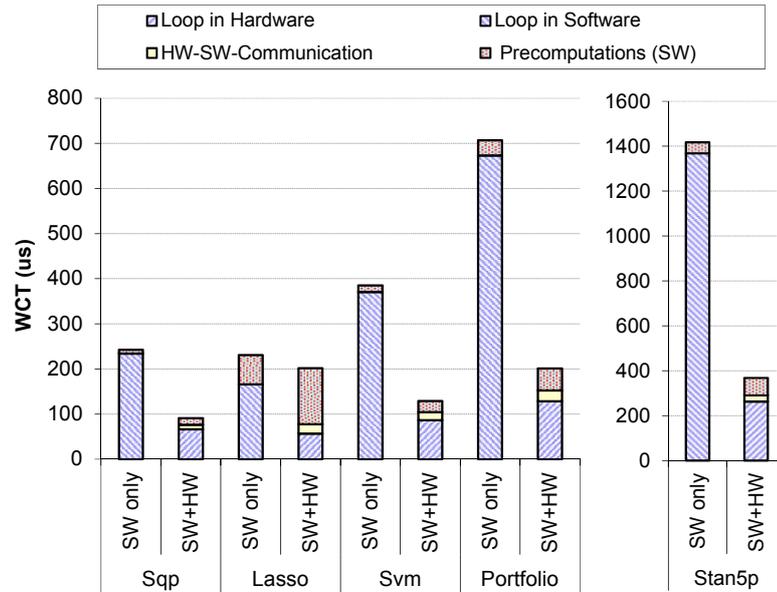
7.9 SPEED-UP VS. SOFTWARE ON ZYNQ SOC

Figure 7.8 finally compares the WCT of executing a single solve operation with and without hardware acceleration on the ZC706 platform. The performance of the hardware accelerated solvers is compared to pure software versions running on the 800 MHz ARM Cortex-A9 core. The software version of the solvers was compiled with `-O3` and uses the hardwired NEON FPU in the processor. Following the recommendation of the CVXGEN authors, the solvers are also alternatively compiled with `gcc -Os`. The faster of the two builds is used for the comparison.

In most cases, the hardware-accelerated solvers perform significantly better than their software-only counterparts. Note that the low system-level speed-up of Lasso is a benchmark-specific anomaly as the *software* code generated by HW/SW-co-synthesis is not optimal for the ARM processor.

7.10 DISCUSSION

In this section, a tool chain was presented to directly generate hardware and software from the domain-specific convex solver description. Furthermore, the performance of the accelerators created by Nymble-RS can be flexibly scaled, achieving significant speed-ups even when



Loop Speed-up:	3.5 x	2.9 x	4.3 x	5.3 x	5.2 x
System Speed-up:	2.7 x	1.1 x	3.0 x	3.5 x	3.8 x

Figure 7.8: Wall clock time on ZC706 platform with and without hardware acceleration

dedicating just a quarter of a mid-size FPGA to the accelerator circuit. The speed-ups reported here are even more significant, as they are achieved not over slow simple soft-core processors (often used as reference in related work), but against 800 MHz dual-issue out-of-order cores with hardwired FPU.

In the tool chain, Nymble-RS significantly exceeds the performance of state-of-the-art academic as well as industrial compilers for translating the floating-point-heavy irregular “C” code generated by CVXGEN, even when expending significant manual effort of optimizing the code for the specific compiler or allowing the competing tools to target larger devices. It is therefore a key-component for the direct synthesis of hardware accelerators for convex optimization problem solvers.

CASE STUDY: SYNTHESIS OF CELLML SIMULATIONS

Modern medical research and drug discovery have profited highly from high-performance computing. An often used application in these fields is the cell-level simulation of partial or complete organs. A key component of these simulations is the numerical integration of an Ordinary Differential Equations (ODE) system, which requires frequent evaluations of the equations in order to achieve satisfactory accuracy for the iterative solvers. This problem is extremely compute intensive.

For the simulation of biological systems at the cell level, CellML [31] has proven to be a useful domain-specific representation, from which simulation models for a number of execution platforms can be created. Since experiments commonly require a multitude of simulation runs with different input data, achieving energy efficiency has become an objective in addition to raw simulation performance.

In this section a case study is presented using CellML-generated code as input for Nymbler-S. The generated hardware is shown to exceed the performance of current generation desktop CPUs in most cases and has energy savings of up to 96% for a single accelerator, which requires just 25...30% area on a mid-sized FPGA. Compared to a Tesla K80 GPU, it offers a lower latency and lower energy consumption.

Parts of the work presented in this section were accepted for publication in [9].

8.1 CELLML-BASED SIMULATION

A CellML model is an XML-based description of a cell comprised of interconnected components and expressed as a system of ODE. OpenCMISS [18] is a simulation workbench where instances of cell models are used as the data points in larger grids (or other spatial arrangements), e.g., to simulate a piece of ventricular tissue.

Roughly, the simulation approach is as follows: the interactions between the neighbouring grid cells are computed at discrete *macro* time steps. In order to progress the state of each cell from the current to the next macro time step, numerical integration of the ODE system is used. The simulation accuracy depends on the granularity of the integration steps: the more *micro* time steps are used to discretise the time between two macro time steps the better the approximation. As the integration phase is done for each cell independently, the resulting potential for acceleration on parallel architectures is huge [18, 133]. Note that this observation has a direct impact on the aim at smaller accel-

Listing 8.1: Excerpt from CCGS output for [39], reformatted for readability

```

void computeRates(double VOI, double* CONSTANTS, double* RATES, double* STATES, double
* ALGEBRAIC) {
    RATES[0] = CONSTANTS[2] * STATES[2] * STATES[3] * (1.0 - STATES[0])
              - CONSTANTS[3] * STATES[0];
    ALGEBRAIC[0] = 1.0 / (1.0 + exp(CONSTANTS[5] * (STATES[1] - CONSTANTS[6])));
    RATES[2] = (ALGEBRAIC[0] - STATES[2]) / CONSTANTS[687];
    ALGEBRAIC[9] = CONSTANTS[116] /
                  (1.0 + pow(CONSTANTS[119] / STATES[10], CONSTANTS[117]));
    ...
}

```

ators: Even though this work focuses on *single-accelerator performance*, the entire FPGA could be tiled with independent processing elements, as the accelerators are not bottlenecked by memory bandwidth. This use of MIMD computation structures also allows FPGA-based systems to scale beyond the SIMD/SIMT-approach used by GPUs.

While it would be possible to build a front-end to parse and interpret CellML directly, the “C Code Generation Service” (CCGS) [82] was used in this approach which infers a sequential execution order for the underlying initial-value problem. The generated highly idiomatic C code is then used as a domain-specific IR for the rest of the compile flow. Listing 8.1 shows an excerpt from the translation of a model by [39]. One execution of the function `computeRates` corresponds to one micro time step in the numerical integration. `VOI` is the “variable of integration” which is usually the time. `CONSTANTS` is a read-only array that contains model parameters. The current state of each component in the model is passed as the read-only array `STATES`. Intermediate values are stored in the array `ALGEBRAIC`. Its elements are always written before read. Finally, the values in the output-only array `RATES` represent the rates of change of the components for the next micro time step. All entities use the C type `double`, i.e. the 64-bit IEEE 754 FP format and all arrays have statically known sizes and are accessed by literal indices. Each statement in the equation-evaluation code is an assignment to one of the aforementioned arrays. The right-hand side of the assignment may be conditional (using the ternary operator `?:`), but other than that, no control structures occur.

8.2 CELLML-SPECIFIC HLS WITH ODOST

The vast potential for parallel processing motivated ODoST [133, 134], a high-level synthesis system custom-made to construct accelerators for the numerical integration phase. ODoST reads the C code derived from a CellML model, emits a *fully-spatial* data path for the `computeRates` function ready for calling by the OpenCMISS framework, and handles hardware synthesis for the accelerator. Internally, FP operators generated by FloPoCo [36] are used.

The ODoST architecture is deeply pipelined and favours throughput instead of latency: while the computation of subsequent micro-time

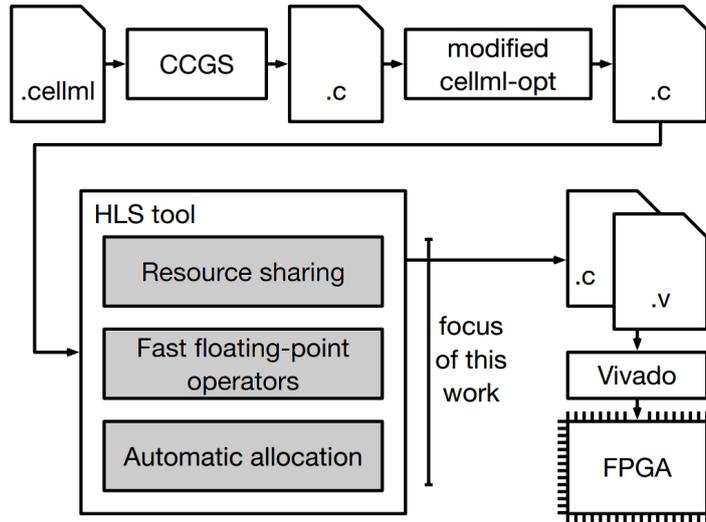


Figure 8.1: CellML-to-accelerator compilation flow

steps for a *single* cell cannot be pipelined, a typical simulation setting contains a sufficient number of cells so that the accelerator can begin to compute a micro-time step for a *different* cell in every cycle on the same accelerator. While the fully-spatial design does achieve the optimum throughput of one result per clock cycle, it may have excessive hardware requirements for larger models, even after applying additional standard and domain-specific compiler optimizations [87].

In contrast, the approach proposed in this thesis constructs latency-optimized, non-pipelined micro-time step accelerators for even the largest CellML models. It leverages the intelligent resource-sharing and fast FP operators proposed by [8], and allows to flexibly trade-off parallelism (number of function units) with area limits. This enables a micro-architectural design-space exploration also considering f_{\max} . Throughput could then be increased by tiling the accelerators as required.

8.3 PROPOSED COMPILATION FLOW

Figure 8.1 shows the compilation flow from an XML-based CellML model to the FPGA hardware design. CCGS [82] is used to translate the actual CellML descriptions from the CellML repository [75] to idiomatic C code.

This code, representing the equation systems, is optimized by CellML-opt [87] a domain-specific high-level synthesis flow with an LLVM-based optimisation pipeline. It originally aims to reduce the hardware area requirements of CellML-based simulation accelerators by selectively applying general-purpose as well as domain-specific transformations, tailored for hardware synthesis. In this work, the “z” flow is applied, which leads to the optimizations of LLVM’s aggressive size optimization preset “-Oz” being applied before the actual hardware synthesis. At this stage,

no unsafe FP transformations are performed. In addition, CellML-opt was modified to rewrite the CCGS output to use actual local variables for intermediate and reused values instead of storing them in arrays. This removal of memory operations allows the actual HLS to more flexibly select between multiple storage mechanisms for intermediate values and also enables, e.g., schedule-sensitive tree height optimization.

The output of the HLS tool consists of Verilog RTL description which is then fed for actual logic synthesis into the vendor tools (e.g. Xilinx Vivado). As the Nymble-RS compiler is a true hardware-software-co-compilation system, it also performs interface synthesis to access the newly generated accelerator from the software. For evaluation purposes, a short software program is used here which performs only the numerical integration step using the accelerators instead of the full-scale OpenCMISS [18] simulation framework.

8.4 TEST SETUP

For ease of integration, the Xilinx Zynq XC7Z045 FPGA on the ZC706 evaluation board [124] was chosen as target, running Xilinx’ “bare metal” software environment [119].

As Nymble-RS is a hardware-software-compiler, it generates hardware and software including the glue-logic in between. However, the CellML examples used in this case study are completely converted into hardware, therefore the focus in this section is on the hardware-part.

As stated before, earlier approaches target the translation of small CellML problems. As this case study targets the translation of large CellML problems, five of the largest models from the CellML repository [75] (at the time of writing) are used as test cases in this chapter. Table 8.1 lists the models with the number of their FP operations.

Table 8.1: Large examples from CellML repository [75]

Model	# operations									Source
	+ -	*	/	pow	exp	log	[]	other	total	
A	615	687	290	160	36	20	0	11	1819	[39]
B	310	386	133	42	57	6	1	29	964	[44]
C	452	411	0	0	0	0	0	30	893	[52]
D	342	456	106	11	45	3	1	12	976	[62]
E	330	448	98	8	45	3	1	11	944	[61]

In all tests and for all compilers benchmarked, inexact mathematical optimizations will be allowed. For Nymble-RS this includes the use of faithful rounding FP units. If not stated otherwise, simulations are run on ten cells with 1 million micro-step iterations each.

Faithful rounding operations for multiplication and division are used to reduce the latency and thereby the II. For division, the 10-cycle

division unit PolyGS proposed in this thesis in Section 6.2 is deployed. For multiplication, a three cycle faithful rounding multiplier is used as described in [15]. Furthermore, a 5-cycle Xilinx floating-point IP-core adder [116] is used for addition and subtraction. The log, exp, pow, and floor calls are handled by the floating-point units described in Section 6.3.

8.5 DESIGN SPACE EVALUATION

Vivado 2014.1 was used for synthesis as it gave better results than more recent versions (2016.2, 2017.2), which created faulty hardware in some cases. All timing and area results shown are post-place-and-route.

Table 8.2 shows the results when generating accelerators with an increasing number of FP operator units (and thus growing hardware area). As can be seen, increasing the number of FP units generally carries an f_{\max} penalty, often due to slower wiring. Thus, the fastest accelerator will not necessarily be the largest one.

In general, using ca. 25...30% of the Zynq Z7045 device (a mid-sized chip) per accelerator achieves the best execution time. The total simulation performance will ideally scale linearly by employing multiple accelerators so the remaining FPGA area can be put to good use.

8.6 COMPUTATION ACCURACY

Table 8.3 compares the average and maximum relative error introduced by single precision arithmetic (as used in ODoST) to the error introduced by the proposed approach using Nymble-RS. The errors shown here are computed relative to a IEEE745-compliant reference software execution using double-precision and compiled without `-ffast-math`.

The error introduced by single precision becomes rather large when one million (or more) iterations are used for integration. Depending on the required simulation accuracy, single precision may not suffice, whereas the presented approach double-precision based approach carries a much smaller average error.

8.7 COMPARISON TO STATE-OF-THE-ART HLS TOOLS

To evaluate the performance of the hardware generated by the Nymble-RS HLS engine, it is compared against other state-of-the-art HLS systems, both academic and industrial.

LegUp: With LegUp being the most prominent academic C-based HLS tool in recent years, it is a natural reference. However, due to its recent commercialization, the license terms for LegUp 5.1 prohibit comparative benchmarking, thus limiting the evaluation to the latest open-source version 4.0, which does not carry that restriction. Since

For the 4-cycle adder IP core, an additional register is required at the input. Otherwise the delay of the first cycle in combination with a large multiplexer would slow down the total maximum operation frequency

Table 8.2: Design space exploration. II=Initiation interval, WCT=Wall Clock Time

Test Case	# FP-Units	# FF	# LUT	# BlockRAM	# DSP	Schedule Length	II	f_{max} (MHz)	WCT (s)
A	12	27647 (6%)	46335 (21%)	36 (3%)	92 (10%)	319	305	193	15.803
	16	33973 (8%)	56610 (26%)	50 (5%)	134 (15%)	238	229	184	12.446
	20	36620 (8%)	65826 (30%)	56 (5%)	144 (16%)	231	204	164	12.439
	24	38834 (9%)	68752 (31%)	60 (6%)	169 (19%)	226	215	169	12.722
B	12	27713 (6%)	42383 (19%)	27 (2%)	77 (9%)	224	202	202	10.000
	16	30310 (7%)	47743 (22%)	31 (3%)	102 (11%)	212	192	199	9.648
	20	33615 (8%)	52579 (24%)	38 (3%)	121 (13%)	191	179	188	9.521
	24	33532 (8%)	57144 (26%)	37 (3%)	131 (15%)	191	179	182	9.835
C	12	21895 (5%)	37186 (17%)	21 (2%)	30 (3%)	202	186	203	9.163
	16	24101 (6%)	43513 (20%)	24 (2%)	40 (4%)	177	161	204	7.892
	20	27207 (6%)	52324 (24%)	25 (2%)	45 (5%)	158	142	191	7.435
	24	28671 (7%)	55580 (25%)	32 (3%)	55 (6%)	148	132	191	6.911
D	12	27763 (6%)	42814 (20%)	28 (3%)	77 (9%)	179	167	200	8.350
	16	29360 (7%)	46705 (21%)	30 (3%)	87 (10%)	169	158	202	7.822
	20	31269 (7%)	51069 (23%)	29 (3%)	97 (11%)	161	150	190	7.895
	24	33798 (8%)	56588 (26%)	32 (3%)	127 (14%)	158	149	173	8.613
E	12	27650 (6%)	42393 (19%)	34 (3%)	77 (9%)	179	163	201	8.109
	16	29264 (7%)	47269 (22%)	29 (3%)	87 (10%)	166	155	196	7.908
	20	31188 (7%)	50596 (23%)	32 (3%)	97 (11%)	157	147	188	7.819
	24	33586 (8%)	54845 (25%)	32 (3%)	127 (14%)	156	148	188	7.872

Table 8.3: Average relative error

	Average Relative Error		Maximum Relative Error	
	single-precision	this approach	single-precision	this approach
A	5.67E-02	2.22E-14	1.37E-00	6.74e-13
B	2.62E-04	1.16E-14	4.78E-03	1.46E-13
C	3.09E-03	1.78E-14	1.75E-02	6.96E-14
D	6.06E-03	3.04E-14	4.09E-02	1.82E-13
E	7.12E-02	9.43E-15	2.93E-01	4.79E-14

LegUp 4.0 does not fully support Xilinx devices, the Altera Cyclone V family of chips is chosen as target for comparison.

Unfortunately, a number of issues caused this attempt to fail: the use of the *floor* function often led to crashes during LegUp HLS. To get around this, the *floor*-calls can be temporarily removed. This allows to proceed to logic synthesis using the Altera Quartus II vendor tools. However Quartus prohibits the use of `exp` as a module name which was present in the Verilog RTL description generated by LegUp. Manually renaming all of the occurrences of `exp` allows a bit more progress during the synthesis, but then leads to multiple “module signcopy not found” error messages. As this module does not seem to have been distributed with the virtual machine image the LegUp authors kindly have made available for experimentation, synthesis had to be given up at this point.

Bambu: In contrast to LegUp and Nymbble/Nymbble-RS, Bambu does not offer a hardware-software co-design flow. However, it can be used to compare the size and speed of the generated hardware. The XC7Z045 FPGA is not natively supported as target device, but a target description file can be found in one of the examples provided by the authors. Using this target file, it accepts a target frequency of 200 MHz*. For fair comparison, the use of FloPoCo FP-Unit library is enabled, as a comparison to unoptimized floating-point units generated from C code would be unfair to Bambu. However, a few seconds after starting, the tool refused to create working hardware as it could not find hardware units for the `pow` and `log` functions. This is surprising because FloPoCo does contain hardware for these functions. In any case, the synthesis could not proceed at that point so an industrial tool had to be used for comparison.

Industrial tool: As described above, it is not allowed to publish the name of the industrial HLS tool used as a reference here due to license restrictions. For these experiments a recent version of a leading C-based HLS tool targeting Xilinx devices was used. Logic synthesis was performed using Xilinx Vivado 2017.2. All numbers reported are post-place-and-route.

* When targeting a Zynq FPGA with speed grade -1 with 200MHz, Bambu shows an error message saying that the BlockRAM would not be fast enough on this device.

Table 8.4: Comparison to accelerators created by industrial HLS tool (*=area constraints for pow violated by the industrial tool, f_{\max} in MHz)

		FFs	LUTs	DSPs	BRAM	II	f_{\max}	WCT (s)
A	Nymble-RS	37K	66K	144	56	204	164	12.4
	Industrial Tool	85K	134K	494	22	413	place&route failed	
		233%	204%	343%	39%	202%		
B*	Nymble-RS	34K	53K	121	38	179	188	9.5
	Industrial Tool	80K	84K	578	68	1202	84	143.1
		237%	160%	478%	179%	672%	45%	1506%
C	Nymble-RS	27K	52K	45	25	142	191	7.4
	Industrial Tool	55K	76K	135	0	169	153	11.0
		201%	144%	300%	0%	119%	80%	148%
D*	Nymble-RS	31K	51K	97	29	150	190	7.9
	Industrial Tool	69K	80K	404	23	1933	85	228.5
		222%	156%	416%	79%	1289%	45%	2894%
E*	Nymble-RS	31K	51K	97	32	148	188	7.8
	Industrial Tool	70K	79K	404	23	1230	83	148.0
		224%	156%	416%	72%	837%	44%	1893%

As before, multiple combinations of inlining, unrolling, and pipelining are manually explored for the input C code, all expressed using the tool’s directives. The best (=fastest) results are reported here. They are achieved with inline and unrolling enabled (pipelining had no effect on the result). For comparability, the upper bound on the number of FP units for each operation type is defined identically to that for Nymble-RS, computed as discussed in Section 5.2.1 with an upper bound of a total of 20 FP units. In addition, the industrial tool is permitted the use of one dedicated pow operator which Nymble-RS emulates using log and exp. For unclear reasons, the industrial tool exceeded this restriction for pow operator in three test cases. Table 8.4 shows the results of the commercial tool expressed in absolute values and as percentage of the Nymble-RS result.

In contrast to LegUp, the industrial tool was actually able to compile the code for all of the models. However, for the largest model (case A, Faville), place-and-route failed due to congestion. Attempts to alleviate this by using specific anti-congestion settings in the Vivado tools failed. When the mapping process did succeed, the generated accelerators were all larger and slower (sometimes by an order of magnitude!) than those generated by Nymble-RS.

8.8 PERFORMANCE / ENERGY RELATIVE TO CPU

The performance and energy efficiency of the Nymble-RS approach relative to a fast CPU is evaluated as follows: on the CPU side, a fast desktop-class Intel Core i7 6700K CPU is used, running at 4.2

Table 8.5: Execution Wall-Clock-Time (10 Cells, 1M iterations each), Power and Energy Consumption (FPGA vs CPU)

Test Case	A	B	C	D	E
WCT i7 6700K (s)	60.17	10.50	1.45	9.67	8.10
WCT XC7Z045 (s)	12.44	9.52	6.91	7.82	7.82
Speed-Up	4.84x	1.10x	0.21x	1.24x	1.04x
Power i7 6700K (W)	19.1	20.4	22.4	20.0	20.2
Power XC7Z045 (W)	3.6	3.0	3.0	3.1	3.2
Energy i7 6700K (J)	1149.8	214.2	32.5	193.6	163.7
Energy XC7Z045 (J)	44.9	28.9	22.1	24.4	24.7
Energy Reduction	96%	87%	32%	87%	85%

GHz. The code is compiled with gcc 5.3.1 with `-O3 -ffast-math`. On the FPGA side, a Xilinx ZC706 Zynq 7045-based prototyping board is used. Again, this comparison concentrates on single-core performance since both the CPU and the FPGA could easily scale to run multiple threads / accelerators in parallel.

The execution time of simulating 10 cells is measured with 1 million iterations per cell. For the FPGA, the 20 FP unit variant of each accelerator is used, running at its specific maximum f_{\max} (see Table 8.2). As before, wall-clock time is reported for the complete execution, including overhead for HW-SW interfaces.

Power measurements on the Intel CPU are performed using the Running Average Power Limit (RAPL) performance counters [114]. On the FPGA, voltage and current measurements are obtained by directly querying Channel 1 of the on-board Voltage Regulator Module, which covers the Zynq’s programmable logic and the ARM cores (both ARM cores sleep here). In both cases I/O power consumption is not included. Afterwards, the total amount of energy used for the computation is calculated by multiplying the run time with average power consumption. The results are shown in Table 8.5.

The generated FPGA-based accelerators are significantly more energy-efficient than the CPU, in the largest model A saving 96% of energy. On the performance side, the FPGA-based accelerators are generally faster than the CPU, for the large model A by almost 5x. The outlier here is model C, which is significantly faster on the CPU than on the FPGA.

8.9 PERFORMANCE / ENERGY RELATIVE TO GPU

This section compares the FPGA result to a GPU implementation. For the GPU implementation, the C code was used to create a CUDA Kernel. Note that the C code generated by `cellml-opt` is used as these optimization improves the CUDA performance as well. 100k cells are sim-

ulated to determine the average rate of cells-computed-per-second on a single NVidia Tesla K80 GK210 GPU. The GK210 is one of NVidias computing-optimized GPUs and has much higher double-precision performance than consumer GPUs. The power is measured using `nvidia-smi`. These results are shown in Table 8.6. While the Nymble-RS approach has better latency, the Tesla-GPU produces more results per second (as expected for a throughput-architecture such as a GPU). Furthermore, the FPGA is more energy efficient (in terms of Joules per cell).

Table 8.6: Single FPGA kernel vs GK210 GPU

Test Case	Latency [s] for one cell	Throughput [Cells per Second]	Power [W]	Energy per Cell [J]
A on FPGA	1.2	0.81	3.6	4.48
A on GPU	322.3	12.1	138.4	11.46
B on FPGA	1.0	1.05	3.0	2.89
B on GPU	91.3	38.69	131.6	3.41
C on FPGA	0.7	1.35	3.0	2.21
C on GPU	23.9	34.60	132.5	3.83
D on FPGA	7.9	1.27	3.1	2.44
D on GPU	72.0	39.64	145.7	3.67
E on FPGA	7.8	1.28	3.2	2.47
E on GPU	75.3	42.98	147.0	3.42

8.10 IMPACT OF SPILLING AND MICROCODE

To visualize the impact of the techniques proposed in Section 5, each test case was synthesized again, one time with spilling disabled and one time with microcode generation disabled. The results are then compared to the result presented in Table 8.2 (spilling and microcode enabled). In all cases, the number of FP-Units was set to 16. The post place and route results are shown in Figure 8.2.

The impact of spilling and microcode on CellML test cases is lower than the impact measured during synthesis of convex problem solvers (see Section 7.3 and Section 7.4). However, while the convex solvers' loop bodies consist of tens of thousand floating-point operations, the CellML computations contain "only" around 893 to 1819. This results in a reduced schedule length with a corresponding reduced number of intermediate values to handle. However, the effect is still visible: especially in test case A, which is the largest test case, spilling reduces the amount of LUTs significantly. This reduction comes at cost of only six BlockRAMs.

The impact of microcode is higher. It reduces LUT usage by about 50% test case A. In the other test cases, 30-35% of the LUTs are saved.

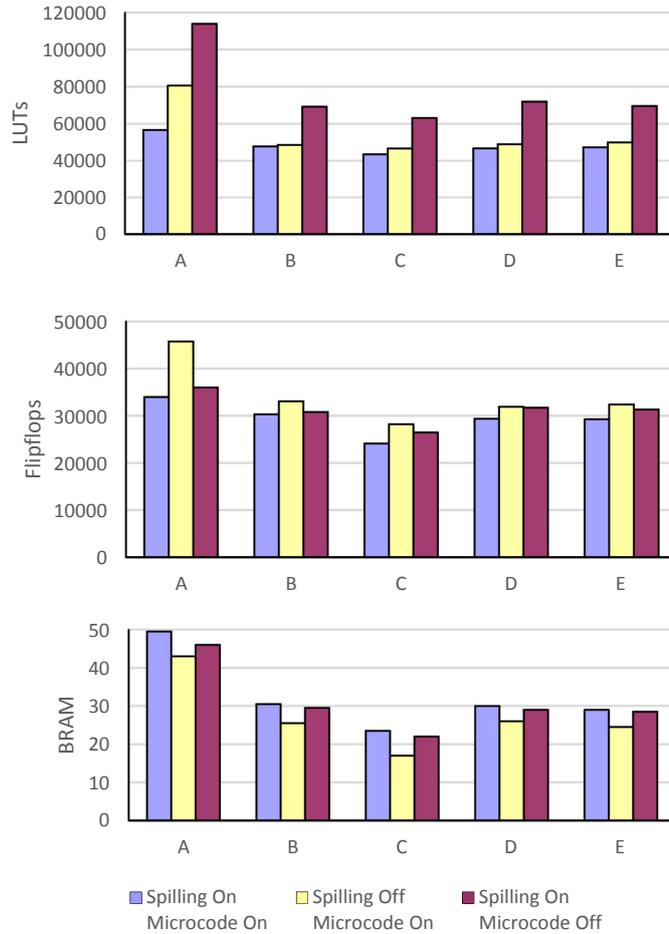


Figure 8.2: Impact of spilling and Microcode on the design size

Note that LUTs are the most critical resource because their percentage usage is the highest of all resource types (see Table 8.2).

8.11 IMPACT OF THE PROPOSED LOW-LATENCY DIVISION UNITS

In this section, the impact of different division units on the CellML accelerator performance is evaluated. Up to this point the PolyGS divider proposed in Section 6.2 was used in all test cases. Now it is replaced by three other division units: the optimized version PolyGSOpt and two configurations of the Xilinx divider IP core. The Xilinx IP cores are configured to a latency of 29 and 57 clock cycles. In the figures they are referred to as Xilinx29 and Xilinx57. All division units are pipelined for a similar operation frequency. More details of the four division units, e.g., standalone synthesis results, can be found in Section 6.2.6.

In Figure 8.3 the II of all test cases with all four division units is shown. As expected, division units with higher latency cause a higher II, therefore and potentially slower accelerator. Note that test case C does not change at all because it does not contain any division operation.

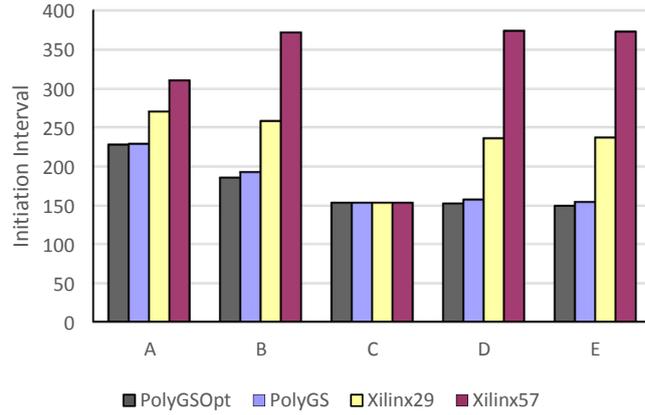


Figure 8.3: Impact of the proposed division units on the initiation interval

Figure 8.4 compares the maximum operation frequency reachable with the four division units. The designs using the 29 cycle version of the Xilinx division units suffer from significant frequency drops. Meanwhile, the use of the PolyGS and PolyGSOpt units cause an average frequency decrease of only 2% and 4% respectively.

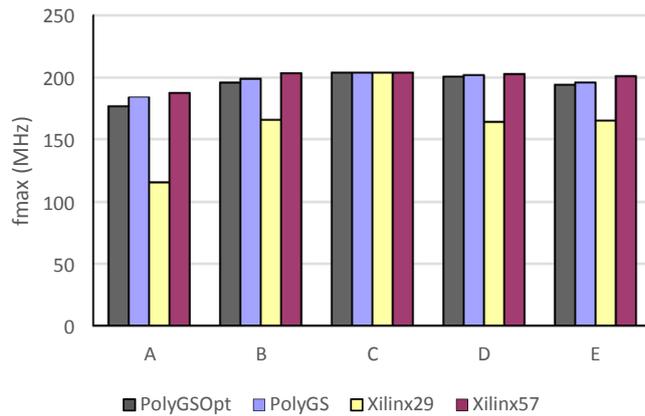


Figure 8.4: Impact of the proposed divider on the maximum operation frequency

The total accelerator performance depends on the II and on the maximum operation frequency. The duration of a single iteration can be computed by multiplying the number of cycles required per iteration (the II) with the duration of a single clock cycle at maximum operation frequency. The result is shown in Figure 8.5.

In all cases except the divider free test case C, the proposed division units provide a significant performance increase. In most cases, the PolyGSOpt unit reaches the fastest result. Only in test case A the PolyGS division unit is faster because this test case reaches a higher maximum operation frequency with this unit.

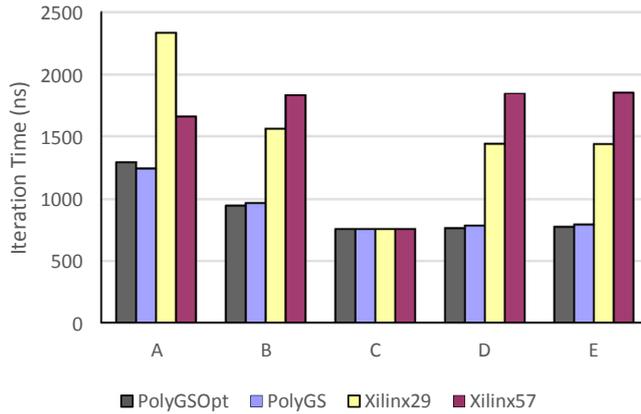


Figure 8.5: Impact of the proposed divider on the single iteration wall time

8.12 ACCELERATOR TILING ON LATEST GENERATION FPGAS

To extrapolate the power of the presented approach beyond the Virtex 7-class devices to latest generation FPGAs, compiling and mapping to a modern XCVU13P-3 UltraScale+ FPGA can be performed. The UltraScale+ is not yet directly supported by the Nymble-RS compiler. Especially, there is no support for the new features of this FPGA generation (like URAM and larger DSP blocks) and no support for any soft core processor that could be used on this FPGA. However, the UltraScale+ architecture is backwards compatible with the Virtex-7 and Virtex-5 architecture. Therefore, it is possible to implement one or multiple hardware kernels on such an FPGA.

Test case A was chosen for hardware implementation on the UltraScale+. The 16 FP unit configuration was used as it is a good trade-off between area usage and performance (see Table 8.2).

On the Zynq platform, the master and the slave port of the kernel are both connected to the ARM processing system (see Section 5.2.5 for details). The XCVU13P FPGA does not contain ARM cores so a different design is chosen. Figure 8.6 shows the system architecture used for the tiling experiments, at the example of a design using two accelerator kernels.

The slave port of each accelerator kernel is connected to a central AXI interface through an AXI crossbar. Using this interface, a controller device such as an SPP can set input values or start the accelerators.

On the other side, each accelerator's AXI master port is connected to a dedicated memory. Here, a URAM-based IP block and a BlockRAM controller were chosen from the Xilinx IP library. This memory replaces the main memory of the original design which is used to store input and output data of each kernel. The second port of each memory block is connected to a second central AXI Slave interface which makes this memory accessible from the outside.

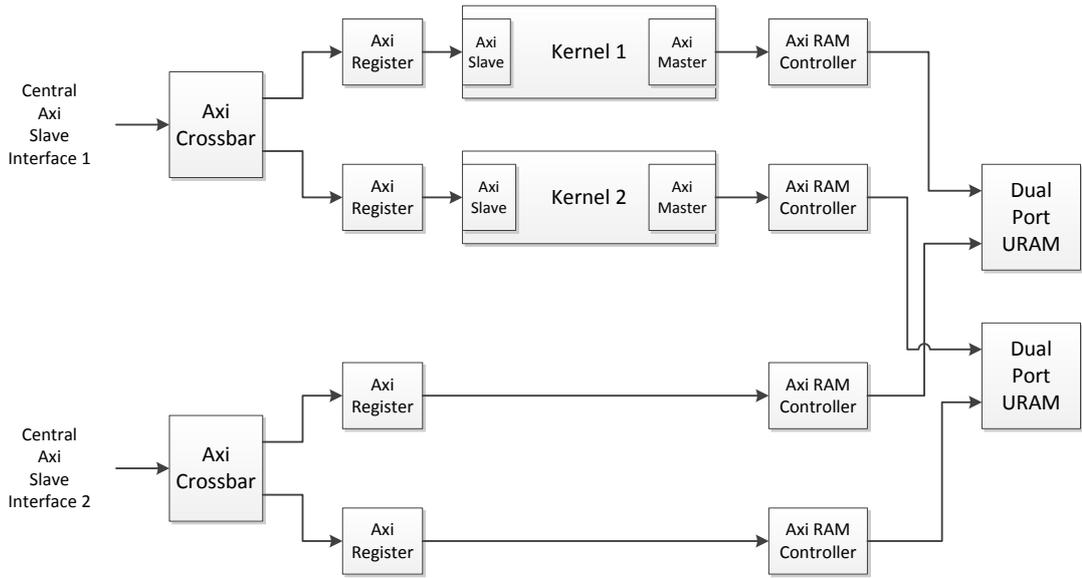


Figure 8.6: System architecture used for multi-kernel synthesis

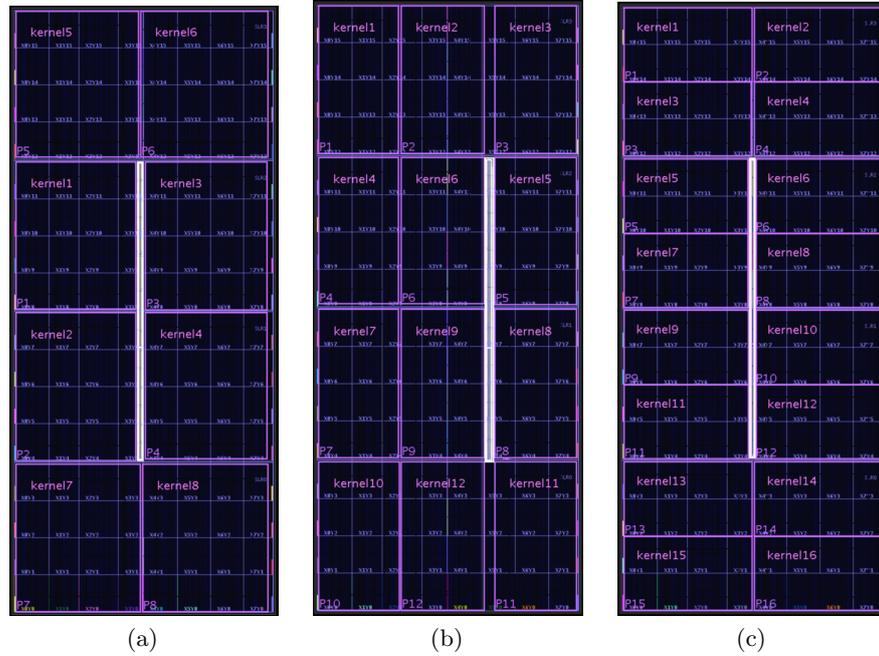


Figure 8.7: Floor plans used for 8, 12 and 16 kernels

Vivado 2017.2 was used for low-level synthesis. However, the tool failed to route all nets if more than four kernels were instantiated, even if the predefined congestion settings were used. Therefore, the floor plans shown in Figure 8.7 have been defined to force a separate placement of each kernel. As the XCVU13P is a multi-die FPGA and timing problems may arise at the crossings without intervention, the floor plans also ensure that each kernel is placed on a single die. The white area in the middle of the floor plan is reserved for the AXI cross-

bars that connect all kernels and the kernel memory to the two central AXI Slave interfaces.

Table 8.7: Synthesis results for multiple kernel instances of test case A on the UltraScale+. II=Initiation interval, WCT=Wall Clock Time

# of Kernels	# FF	# LUT	# BRAM	# DSP	f_{\max} (MHz)
1	28044 (1%)	49507 (3%)	48 (2%)	202 (2%)	316
8	217334 (6%)	384692 (22%)	388 (14%)	1616 (13%)	282
12	325735 (9%)	578110 (33%)	582 (22%)	2424 (20%)	257
16	429840 (12%)	765334 (44%)	776 (29%)	3232 (26%)	failed

Table 8.7 shows the results for a varying number of instances of test case A. A single instance achieves a maximum operation frequency of up to 316 MHz. This is an increase of 71% compared to the Zynq FPGA. The maximum clock frequency decreases when the number of kernels increases. However, with 12 kernels active, the design still reaches 257 MHz, which is about 81% of the single kernels clock frequency.

The 16 kernel design failed in the routing step due to congestion even if a floor plan was provided and the predefined congestion settings were used. Although only 44% of the device resources are used, no solution could be found to achieve a successful routing.

Figure 8.8 shows the speed-ups achieved by the UltraScale+ FPGA when compared to a single thread on the i7 6700K. A single kernel already reaches a speed-up of 8.3x due to the higher clock frequency on the UltraScale+ FPGA. By using 12 kernels an acceleration of more than 80x can be achieved.

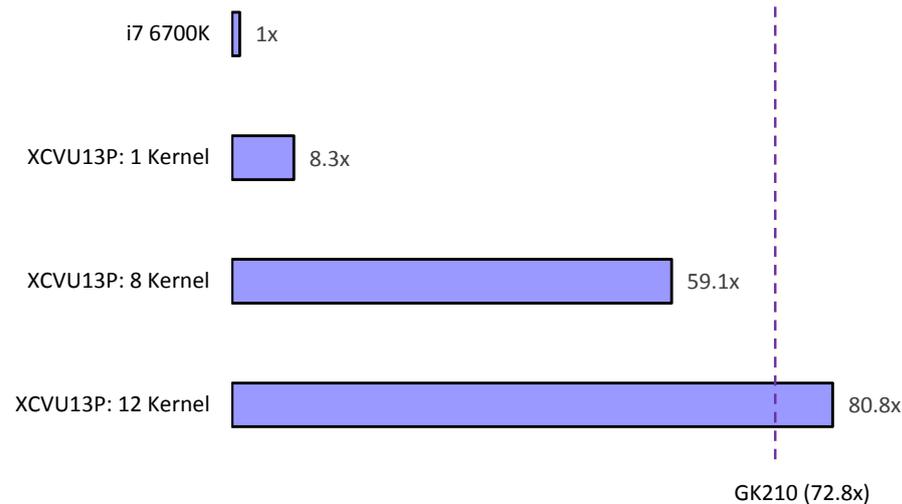


Figure 8.8: Speed-up of multiple kernels on the UltraScale+ compared to the i7 6700K (single thread)

Note that the GK210 is manufactured in a 28nm process, while the much newer UltraScale+ FPGA uses 16nm FinFET+ technology. Therefore a direct comparison would not be fair. However, a newer GPU was not available for testing, so the GK210 result is included, but only for informational purposes.

8.13 DISCUSSION

In this chapter, many of the previously presented methods and technologies have been used to create a tool chain that enables fully automated implementation of a CellML model in hardware. While the ODoST approach translates small to mid-size models using single-precision floating-point arithmetic, the presented method can handle even the largest models in double-precision.

Of course, a method that uses shared resources cannot achieve the high bandwidth of a fully-spatial approach (one result per clock). Therefore ODoST is preferable if single-precision accuracy is sufficient and the model is not too large for target FPGA. For all other cases where ODoST cannot be used, the presented tool flow offers a solution.

SUMMARY AND FUTURE WORK

9.1 SUMMARY

In this work multiple techniques for the translation of high-level floating-point applications to FPGAs have been proposed and evaluated.

For the implementation of floating-point arithmetic, multiple highly-optimized low-latency floating-point units have been developed. While the proposed FCS-FMA unit outperforms its closest competitors by a factor of 2.6x in direct comparison (Section 6.1.9), it could not reveal its full potential in the application-level test due to a drop of the maximum operation frequency (Section 7.7). Further research must show if it is possible to completely avoid the drop of the maximum operation frequency so that the performance increase of the FMA can be fully utilized. In contrast, the proposed division units were shown to deliver a significant performance increase compared to state-of-the-art dividers, not only in stand-alone synthesis but also in the real applications.

The experimental compiler framework Nymble has been successfully extended to Nymble-RS (Section 5), a HLS tool capable of advanced floating-point computations and resource sharing. For the first time in HLS targeting FPGAs, spilling of intermediate values and micro-code controlled multiplexers could be identified as key-technologies for synthesis of large blocks of floating-point code.

In the two case studies that use Nymble-RS as HLS engine in domain-specific compilation flows, the compiler optimizations and the improved floating-point units could be employed in practice. In the case of convex optimization solvers, the accelerators created by Nymble-RS achieve significant speed-ups even when dedicating just a quarter of a mid-size FPGA to the accelerator circuit. The speed-ups are even more significant, as they are achieved not over slow simple soft-core processors (often used as reference in related work), but against a 800 MHz dual-issue out-of-order ARM core with hardwired FPU. In the case of CellML simulation acceleration, the accelerators can beat a 4.2 GHz i7 6700K CPU (in four of five cases) and a Tesla GPU in latency and energy per computation.

In both cases, Nymble-RS significantly exceeds the performance of state-of-the-art academic as well as industrial compilers for compiling floating-point-heavy irregular complex “C” code, even when expending significant manual effort of optimizing the code for the specific compiler, or allowing the competing tools to target larger devices. Furthermore, the compiler allows to flexibly trade-off between accelerator size and speed by defining limits for the number of floating-point unit instances.

9.2 FUTURE WORK

Faithful rounding floating-point operations have been shown to be very beneficial in terms of latency and area while causing only a small loss in precision. In this work FPGA-optimized divider designs for use with double-precision accuracy were presented. In order to fully use the potential of FPGAs, it would be necessary for the compiler to reduce (or increase) the precision of floating-point computations to exactly meet the demands of the target application. To support faithful rounding in every word-length, a set of parametrized floating-point operations could be developed. For each operation (div, log, ...) multiple scalable designs are required because a single approach is only optimal for a very narrow band of word lengths. For example, for a 12-13 bit mantissa division, a simple lookup will be sufficient, while above 14 bits, linear (and later quadratic) approximation could be the optimum. In this work, the PolyGS approach was shown to be superior to the TrippleGS approach at a mantissa width of 52+1 bit. It is reasonable to assume that PolyGS is also superior for, e.g., 51 or 54 bit wide mantissas. However, for other word lengths it is unknown which approach delivers the best performance. Furthermore, other approaches, e.g., polynomial approximation of a higher degree must be considered, too.

Finding the optimal floating-point word length is an other important topic that should be addressed in future work. This task could be supported by the compiler itself. By instantiating variable precision floating-point units, the compiler could generate a design that can be used for evaluation of different floating-point precisions. The variable precision floating-point could be controlled by micro-code so that each operation gets an individual precision assigned. A software program would control the evaluation results and execute a heuristic to define and configure the precision for the next run. CPUs as well as GPUs do not deliver good performance for custom floating-point formats which they do not natively support. Therefore, the use of FPGAs could accelerate this optimization significantly.

Nymble-RS can translate loop bodies of almost any size, only being limited by the amount of BlockRAM required for the micro code. With the new Ultra Scale and Ultra Scale+ FPGAs, Xilinx introduced new RAM technologies, namely UltraRAM and High Bandwidth Memory. Although both new RAM types cannot be programmed with predefined values, the microcode could be uploaded before the accelerator starts. This should enable the synthesis of even larger loop bodies, e.g., the larger convex solvers.

For the intermediate storage and as scratch-pad memory, the compiler currently uses simple dual-port RAM implemented in BlockRAM or in LUT-Ram. A read access takes two cycles and if multiple read ports are required, the RAM is replicated. This architecture delivered good performance in most cases examined in this work. However, it

may not be the optimal solution. Especially the CellML test case C suffered from the lack of array partitioning in Nymble-RS. This is a feature that should be implemented in the compiler in the future. But also a change in the underlying memory is worth considering. Having the BlockRAM running at twice the clock speed could allow four memory accesses per cycle. Further approaches for implementing multi-port memories on FPGAs can be found in [67]. Even if the increase in memory ports comes at the cost of an increased read access latency, it could be an acceptable trade-off for many applications. The compiler could automatically evaluate the schedule length for multiple memory architectures and choose the best one.

Finally, the proposed allocation heuristic could be taken as starting point for a more advanced approach. By using iterative scheduling and simulated annealing [110], a (near-)optimal solution could be found that minimizes the schedule length (or Π).

PUBLICATIONS

- [1] Andreas Engel, Björn Liebig, and Andreas Koch. “Feasibility Analysis of Reconfigurable Computing in Low-Power Wireless Sensor Applications.” In: *Reconfigurable Computing: Architectures, Tools and Applications (ARC)*. 2011, pp. 261–268. URL: http://dx.doi.org/10.1007/978-3-642-19475-7_27.
- [2] Andreas Engel, Björn Liebig, and Andreas Koch. “Energy-efficient heterogeneous reconfigurable sensor node for distributed structural health monitoring.” In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2012, pp. 1–8.
- [3] Andreas Engel, Björn Liebig, and Andreas Koch. “HaLOEWEn: A heterogeneous reconfigurable sensor node for distributed structural health monitoring.” In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2012.
- [4] Jens Huthmann, Björn Liebig, Julian Oppermann, and Andreas Koch. “Hardware/software co-compilation with the Nymbler system.” In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, July 2013, pp. 1–8. ISBN: 978-1-4673-6180-4. DOI: 10.1109/ReCoSoC.2013.6581538.
- [5] Oliver Janda, Björn Liebig, Holger Lange, Ulrich Konigorski, and Andreas Koch. “Design and Hardware Implementation of a Controller for Active Damping of a Smart Structure.” In: *In 14th International Adaptronic Congress, Darmstadt*. 2011.
- [6] Björn Liebig, Jens Huthmann, and Andreas Koch. “Architecture exploration of high-performance floating-point fused multiply-add units and their automatic use in high-level synthesis.” In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 134–143.
- [7] Björn Liebig and Andreas Koch. “Low-latency double-precision floating-point division for FPGAs.” In: *Field-Programmable Technology (FPT), 2014 International Conference on*. IEEE. 2014, pp. 107–114.
- [8] Björn Liebig and Andreas Koch. “High-level synthesis of resource-shared microarchitectures from irregular complex C-code.” In: *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE. 2016, pp. 133–140.

- [9] Björn Liebig, Julian Oppermann, and Andreas Koch. “Improved High-Level Synthesis for Complex CellML Models.” In: *Reconfigurable Computing: Architectures, Tools and Applications (ARC)*. 2018.

BIBLIOGRAPHY

- [10] ARM. *Cortex-A9 Technical Reference Manual Documentation*. 2010. URL: https://static.docs.arm.com/ddi0388/f/DDI0388F_cortex_a9_r2p2_trm.pdf (visited on 01/11/2018).
- [11] ARM. *AMBA AXI and ACE Protocol Specification*. 2011. URL: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf (visited on 01/11/2018).
- [12] Thomas L Adam, K. Mani Chandy, and JR Dickson. “A comparison of list schedules for parallel processing systems.” In: *Communications of the ACM* 17.12 (1974), pp. 685–690.
- [13] S.P. Amarasinghe, J.M. Anderson, C.S. Wilson, Shih-Wei Liao, B.R. Murphy, R.S. French, M.S. Lam, and Mary W. Hall. “Multiprocessors from a software perspective.” In: *Micro, IEEE* 16.3 (1996), pp. 52–61. ISSN: 0272-1732. DOI: 10.1109/40.502406.
- [14] Jason D Bakos. “High-performance heterogeneous computing with the Convey HC-1.” In: *Computing in Science & Engineering* 12.6 (2010), pp. 80–87.
- [15] Sebastian Banescu, Florent de Dinechin, Bogdan Pasca, and Radu Tudoran. “Multipliers for floating-point double precision and beyond on FPGAs.” In: *ACM SIGARCH Computer Architecture News* 38.4 (Jan. 2010), pp. 73–79. ISSN: 01635964. DOI: 10.1145/1926367.1926380. URL: <http://dl.acm.org/citation.cfm?id=1926380><http://portal.acm.org/citation.cfm?doid=1926367.1926380>.
- [16] M. Benmohammed, S. Merniz, and M. Bourahla. “Asip micro-code generation from high-level specifications.” In: *Proceedings. 2004 International Conference on Information and Communication Technologies: From Theory to Applications, 2004*. IEEE, 2004, pp. 587–588. ISBN: 0-7803-8482-2. DOI: 10.1109/ICTTA.2004.1307899. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1307899>.
- [17] Lieven Boyd, Stephen and Vandenberghe. *Convex Optimization*. December. Cambridge university press, 2004. ISBN: 978-0-521-83378-3.
- [18] Chris Bradley et al. “OpenCMISS: A multi-physics & multi-scale computational infrastructure for the VPH/Physiome project.” In: *Progress in Biophysics and Molecular Biology* 107.1 (2011). Experimental and Computational Model Interactions in Bio-Research: State of the Art, pp. 32–47. ISSN: 0079-6107. DOI:

- <https://doi.org/10.1016/j.pbiomolbio.2011.06.015>. URL: <http://www.sciencedirect.com/science/article/pii/S0079610711000629>.
- [19] D. Brèlaz. “New methods to color the vertices of a graph.” In: *Communications of the Assoc. of Comput. Machinery*. 1979, pp. 251–256.
- [20] JS Brooks and CH Olson. *Processor Pipeline which Implements Fused and Unfused Multiply-Add Instructions*. 2012. URL: <http://www.freepatentsonline.com/y2012/0221614.html>.
- [21] J. R. Brown. “Chromatic scheduling and the chromatic number problem.” In: *Management Science* 19. 1972, pp. 456–463.
- [22] T.J. Callahan. “Automatic compilation of C for hybrid reconfigurable architectures.” PhD thesis. UC Berkeley, 2002.
- [23] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. “LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems.” In: *Proc. Intl. Symp. on Field Programmable Gate Arrays (FPGA)*. Monterey, CA, USA, 2011, pp. 33–36. ISBN: 978-1-4503-0554-9.
- [24] B. Catanzaro and B. Nelson. “Higher Radix Floating-Point Representations for FPGA-Based Arithmetic.” In: *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’05)* (2005), pp. 161–170. DOI: 10.1109/FCCM.2005.43. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1508536>.
- [25] Christian Schneider. *Programming mpfr::real*. URL: http://chschneider.eu/programming/mpfr_real/ (visited on 01/08/2018).
- [26] J. Cong and J. Xu. “Simultaneous FU and Register Binding Based on Network Flow Method.” In: *2008 Design, Automation and Test in Europe*. 2008, pp. 1057–1062. DOI: 10.1109/DATE.2008.4484821.
- [27] Jason Cong, Hui Huang, and Wei Jiang. “A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis.” In: (Mar. 2010), pp. 1255–1260. URL: <http://dl.acm.org/citation.cfm?id=1870926.1871228>.
- [28] Jason Cong and Wei Jiang. “Pattern-based behavior synthesis for FPGA resource reduction.” In: *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays - FPGA ’08*. New York, New York, USA: ACM Press, Feb. 2008, p. 107. ISBN: 9781595939340. DOI: 10.1145/1344671.1344688. URL: <http://dl.acm.org/citation.cfm?id=1344671.1344688>.

- [29] National Research Council et al. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press, 2011.
- [30] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. “An introduction to high-level synthesis.” In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 8–17.
- [31] Autumn A. Cuellar, Catherine M. Lloyd, Poul M. F. Nielsen, David P. Bullivant, David P. Nickerson, and Peter J. Hunter. “An Overview of CellML 1.1, a Biological Model Description Language.” In: *Simulation* 79.12 (2003), pp. 740–747. DOI: 10.1177/0037549703040939. URL: <https://doi.org/10.1177/0037549703040939>.
- [32] Jérémie Detrey and Florent Dinechin. “A Tool for Unbiased Comparison between Logarithmic and Floating-point Arithmetic.” In: *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 49.1 (May 2007), pp. 161–175. ISSN: 0922-5773. DOI: 10.1007/s11265-007-0048-7. URL: <http://www.springerlink.com/index/10.1007/s11265-007-0048-7>.
- [33] Jérémie Detrey and Florent de Dinechin. “Parameterized floating-point logarithm and exponential functions for FPGAs.” In: *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing* 31.8 (2007), pp. 537–545. DOI: 10.1016/j.micpro.2006.02.008.
- [34] Florent De Dinechin and Bogdan Pasca. “An FPGA-specific approach to floating-point accumulation and sum-of-products.” In: *... Technology, 2008. FPT ...* (2008), pp. 33–40. URL: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=4762363.
- [35] Florent de Dinechin, Cristian Klein, and Bogdan Pasca. “Generating high-performance custom floating-point pipelines.” In: *2009 Int. Conf. on Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 59–64. ISBN: 9781424438921. DOI: 10.1109/FPL.2009.5272553. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5272553>.
- [36] Florent de Dinechin and Bogdan Pasca. “Designing Custom Arithmetic Data Paths with FloPoCo.” In: *IEEE Design & Test of Computers* 28.4 (July 2011), pp. 18–27. ISSN: 0740-7475. DOI: 10.1109/MDT.2011.44. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5753874>.
- [37] Stephen M Erlien, Joseph Funke, and J Christian Gerdes. “Incorporating non-linear tire dynamics into a convex approach to shared steering control.” In: *American Control Conference (ACC), 2014*. IEEE, 2014, pp. 3468–3473.

- [38] Xin Fang and Miriam Leeser. “Vendor agnostic, high performance, double precision Floating Point division for FPGAs.” In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sept. 2013, pp. 1–5. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6670335>.
- [39] Richard A Faville, Andrew J Pullan, Kenton M Sanders, S D Koh, Catherine M Lloyd, and Nicholas P Smith. *Biophysically based mathematical modeling of interstitial cells of Cajal slow wave activity generated from a discrete unitary potential basis*. CellML file: faville_model_2008.cellml (Catherine Lloyd). 2009. URL: https://models.cellml.org/workspace/faville_pullan_sanders_koh_lloyd_smith_2009.
- [40] Mike Fingeroff and Thomas Bollaert. *High-Level Synthesis Blue Book*. Mentor Graphics Corp., 2010.
- [41] Hagen Gädke-Lütjens. “Dynamic Scheduling in High-Level Compilation for Adaptive Computers.” Dissertation. TU Braunschweig, 2011, p. 242.
- [42] Hagen Gädke and Andreas Koch. “Comrade-A Compiler for Adaptive Systems.” In: *Design, Automation and Test in Europe (DATE)*. 2007. URL: https://www.esa.informatik.tu-darmstadt.de/twiki/pub/Staff/AndreasKochPublications/comrade_date07.pdf.
- [43] Robert E Goldschmidt. “Applications of division by convergence.” PhD thesis. Massachusetts Institute of Technology, 1964.
- [44] Eleonora Grandi, Francesco S Pasqualini, and Donald M Bers. *A novel computational model of the human ventricular action potential and Ca transient*. CellML file: grandi_pasqualini_bers_2010_flat.cellml (Geoffrey Nunns). 2010. URL: https://models.physioomeproject.org/workspace/grandi_pasqualini_bers_2010.
- [45] S Grant, M and Boyd. *CVX: Matlab software for disciplined convex programming*. 2008. URL: <http://www.stanford.edu/~boyd/cvx/>.
- [46] Stefan Hadjis, Andrew Canis, Jason H. Anderson, Jongsok Choi, Kevin Nam, Stephen Brown, and Tomasz Czajkowski. “Impact of FPGA architecture on resource sharing in high-level synthesis.” In: *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA '12* (2012), p. 111. DOI: 10.1145/2145694.2145712. URL: <http://dl.acm.org/citation.cfm?doid=2145694.2145712>.
- [47] Kyung-Nam Han, Alexandre F. Tenca, and David Tran. “High-speed floating-point divider with reduced area.” In: *SPIE Optical Engineering + Applications*. Ed. by Mark S. Schmalz, Gerhard X. Ritter, Junior Barrera, Jaakko T. Astola, and Franklin T.

- Luk. International Society for Optics and Photonics, Aug. 2009, 744400–744400–8. URL: <http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=786791>.
- [48] R. Hartley and A. Casavant. “Tree-height minimization in pipelined architectures.” English. In: *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*. IEEE Comput. Soc. Press, pp. 112–115. ISBN: 0-8186-1986-4. DOI: 10.1109/ICCAD.1989.76916. URL: <http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=76916>.
- [49] K. Scott Hemmert and Keith D. Underwood. “Fast, Efficient Floating-Point Adders and Multipliers for FPGAs.” In: *ACM Transactions on Reconfigurable Technology and Systems* 3.3 (Sept. 2010), pp. 1–30. ISSN: 19367406. DOI: 10.1145/1839480.1839481. URL: <http://dl.acm.org/citation.cfm?id=1839480.1839481><http://portal.acm.org/citation.cfm?doid=1839480.1839481>.
- [50] Dietmar Hildenbrand, Holger Lange, Florian Stock, and Andreas Koch. “Efficient Inverse Kinematics Algorithm Based on Conformal Geometric Algebra - Using Reconfigurable Hardware.” In: *Intl. Conf. on Computer Graphics Theory and Applications (GRAPP), Portugal*. Jan. 2008, pp. 300–307.
- [51] Erdem Hokenek, Robert K Montoye, and Peter W Cook. “Second-generation RISC floating point with multiply-add fused.” In: *IEEE Journal of Solid-State Circuits* 25.5 (1990), pp. 1207–1213.
- [52] Jorrit J Hornberg, Bernd Binder, Frank J Bruggeman, Birgit Schoeberl, Reinhart Heinrich, and Hans V Westerhoff. *Control of MAPK signalling: from complexity to what really matters*. CellML file: hornberg_binder_brugge-man_schoeberl_heinrich_westerhoff_2005.cellml (Catherine Lloyd). 2005. URL: https://models.cellml.org/workspace/hornberg_binder_brugge-man_schoeberl_heinrich_westerhoff_2005.
- [53] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. “The effect of compiler optimizations on high-level synthesis-generated hardware.” In: *ACM Transactions on Reconfigurable Technology and Systems (TRETTS)* 8.3 (2015), p. 14.
- [54] Jens Huthmann, Peter Müller, Florian Stock, Dietmar Hildenbrand, and Andreas Koch. “Compiling Geometric Algebra Computations into Reconfigurable Hardware Accelerators.” In: *Dynamically Reconfigurable Architectures*. Ed. by Peter M Athanas et al. Dagstuhl Seminar Proceedings 10281. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. URL: <http://drops.dagstuhl.de/opus/volltexte/2010/2838>.

- [55] IEEE. *IEEE Standard for Floating-Point Arithmetic*. 2008.
- [56] Intel, Inc. *Stratix V Device Handbook Volume 1: Device Interfaces and Integration (Version 2015.12.21)*. 2014. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf (visited on 11/28/2017).
- [57] Intel, Inc. *Floating-Point IP Cores User Guide*. 2016. URL: https://www.altera.com/literature/ug/ug_altfp_mfug.pdf (visited on 11/28/2017).
- [58] Intel, Inc. *Intel Arria 10 Native FloatingPoint DSP Intel FPGA IP User Guide (Version 2017.11.06)*. 2017. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf (visited on 11/28/2017).
- [59] Intel, Inc. *Intel Arria 10 Device Overview*. 2018. URL: https://www.altera.com/en_US/pdfs/literature/hb/arria-10/a10_overview.pdf (visited on 01/11/2018).
- [60] Intel, Inc. *Hardened Floating-Point Processing in Arria 10 FPGAs and SoCs*. 2017. URL: <https://www.altera.com/products/fpga/features/dsp/arria10-dsp-block.html> (visited on 12/22/2017).
- [61] V Iyer, R Mazhari, and R L Winslow. *A computational model of the human left-ventricular epicardial myocyte*. CellML file: `iyer_mazhari_winslow_2004.cellml` (Steven Niederer). 2004. URL: https://models.cellml.org/workspace/iyer_mazhari_winslow_2004.
- [62] Vivek Iyer, Roger J Hajjar, and Antonis A Armoundas. *Mechanisms of Abnormal Calcium Homeostasis in Mutations Responsible for Catecholaminergic Polymorphic Ventricular Tachycardia*. CellML file: `iyer_2007_ss.cellml` (Penny Noble). 2007. URL: https://models.cellml.org/workspace/iyer_hajjar_armoundas_2007.
- [63] Shailendra Jain, Vasantha Erraguntla, Sriram R. Vangal, Yatin Hoskote, Nitin Borkar, Tulasi Mandepudi, and Karthik Vp. "A 90mW/GFlop 3.4GHz Reconfigurable Fused/Continuous Multiply-Accumulator for Floating-Point and Integer Operands in 65nm." In: *2010 23rd Int. Conf. on VLSI Design* (Jan. 2010), pp. 252–257. DOI: 10.1109/VLSI.Design.2010.59. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5401328>.
- [64] Andreas Koch et al. "Advances in Adaptive Computer Technology." In: *Habilitation, Integrated Circuit Design (EIS), Tech. Univ. Braunschweig, Germany* (2004).

- [65] I. Koren and O. Zinaty. “Evaluating elementary functions in a numerical coprocessor based on rational approximations.” English. In: *IEEE Transactions on Computers* 39.8 (1990), pp. 1030–1037. ISSN: 00189340. URL: <http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=57042>.
- [66] Nasser Kurd, Jonathan Douglas, Praveen Mosalikanti, and Rajesh Kumar. “Next generation Intel® micro-architecture (Nehalem) clocking architecture.” In: *VLSI Circuits, 2008 IEEE Symposium on*. IEEE. 2008, pp. 62–63.
- [67] Charles Eric LaForest and J Gregory Steffan. “Efficient multi-ported memories for FPGAs.” In: *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2010, pp. 41–50.
- [68] Monica S Lam. “An overview of the SUIF2 system.” In: *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 1999.
- [69] Holger Lange. “Reconfigurable Computing Platforms and Target System Architectures for Automatic HW/SW Compilation.” PhD thesis. Darmstadt: Technische Universität, 2011. URL: <http://tuprints.ulb.tu-darmstadt.de/2560/>.
- [70] Holger Lange, Thorsten Wink, and Andreas Koch. “MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers.” In: *DATE'11*. 2011, pp. 1352–1357.
- [71] Holger Lange, Florian Stock, Andreas Koch, and Dietmar Hildenbrand. “Acceleration and energy efficiency of a geometric algebra computation using reconfigurable computers and GPUs.” In: *17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*. IEEE. 2009, pp. 255–258.
- [72] M. Langhammer. “High performance matrix multiply using fused datapath operators.” In: *2008 42nd Asilomar Conference on Signals, Systems and Computers*. 2008, pp. 153–159. DOI: 10.1109/ACSSC.2008.5074382.
- [73] Martin Langhammer and Tom VanCourt. “FPGA Floating Point Datapath Compiler.” In: *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 2009, pp. 259–262. ISBN: 978-0-7695-3716-0. DOI: 10.1109/FCCM.2009.54. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5290908>.
- [74] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation.” In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

- [75] Catherine M. Lloyd, James R. Lawson, Peter J. Hunter, and Poul M. F. Nielsen. “The CellML Model Repository.” In: *Bioinformatics* 24.18 (2008), pp. 2122–2123. DOI: 10.1093/bioinformatics/btn390. URL: <https://doi.org/10.1093/bioinformatics/btn390>.
- [76] J. Lofberg. “YALMIP : a toolbox for modeling and optimization in MATLAB.” In: *2004 IEEE International Conference on Robotics and Automation (IEEE Cat. No.04CH37508)*. IEEE, 2004, pp. 284–289. ISBN: 0-7803-8636-1. DOI: 10.1109/CACSD.2004.1393890. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1393890>.
- [77] Peter Markstein. “IA-64 and Elementary Functions: Speed and Precision, ser.” In: *Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ* (2000).
- [78] Jacob Mattingley and Stephen Boyd. “CVXGEN: a code generator for embedded convex optimization.” In: *Optimization and Engineering* 13.1 (Nov. 2012), pp. 1–27. ISSN: 1389-4420. DOI: 10.1007/s11081-011-9176-9. URL: <http://www.springerlink.com/index/10.1007/s11081-011-9176-9>.
- [79] Larry McMurchie and Carl Ebeling. “PathFinder: a negotiation-based performance-driven router for FPGAs.” In: *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*. ACM, 1995, pp. 111–117.
- [80] Mentor Graphics Corp. *ModelSim Product Website*. 2017. URL: <https://www.mentor.com/products/fv/modelsim/> (visited on 12/03/2017).
- [81] Mike Wissolik and Darren Zacher and Anthony Torza and Brandon Da. *Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance*. 2017. URL: https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf (visited on 12/22/2017).
- [82] Andrew K. Miller, Justin Marsh, Adam Reeve, Alan Garny, Randall Britten, Matt D. B. Halstead, Jonathan Cooper, David P. Nickerson, and Poul M. F. Nielsen. “An overview of the CellML API and its implementation.” In: *BMC Bioinformatics* 11 (2010), p. 178. DOI: 10.1186/1471-2105-11-178. URL: <https://doi.org/10.1186/1471-2105-11-178>.
- [83] Manfred Morari and Jay H Lee. “Model predictive control: past, present and future.” In: *Computers & Chemical Engineering* 23.4 (1999), pp. 667–682.
- [84] NVIDIA Corporation. *NVIDIA TESLA K80*. 2017. URL: <http://www.nvidia.com/object/tesla-k80.html> (visited on 12/03/2017).

- [85] R. Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2513673.
- [86] Neal Oliver, Rahul R Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, et al. “A reconfigurable computing system based on a cache-coherent fabric.” In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE, 2011, pp. 80–85.
- [87] Julian Oppermann, Andreas Koch, Ting Yu, and Oliver Sinn. “Domain-specific optimisation for the high-level synthesis of CellML-based simulation accelerators.” In: *25th International Conference on Field Programmable Logic and Applications, FPL 2015, London, United Kingdom, September 2-4, 2015*. IEEE, 2015, pp. 1–7. ISBN: 978-0-9934-2800-5. DOI: 10.1109/FPL.2015.7294019. URL: <https://doi.org/10.1109/FPL.2015.7294019>.
- [88] Geoffrey K Ottman, Heath F Hofmann, Archin C Bhatt, and George A Lesieutre. “Adaptive piezoelectric energy harvesting circuit for wireless remote power supply.” In: *IEEE Transactions on power electronics* 17.5 (2002), pp. 669–676.
- [89] Behrooz Parhami. *Computer Arithmetic*. Second. Oxford University Press, 2010. ISBN: 978-0-19-532848-6.
- [90] Bogdan Pasca. “Correctly rounded floating-point division for DSP-enabled FPGAs.” In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2012, pp. 249–254. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6339189>.
- [91] F. Philipp, F. A. Samman, and M. Glesner. “Design of an autonomous platform for distributed sensing-actuating systems.” In: *2011 22nd IEEE International Symposium on Rapid System Prototyping*. 2011, pp. 85–90. DOI: 10.1109/RSP.2011.5929980.
- [92] J.-A. Pineiro and J.D. Bruguera. “High-speed double-precision computation of reciprocal, division, square root, and inverse square root.” In: *IEEE Transactions on Computers* 51.12 (Dec. 2002), pp. 1377–1388. ISSN: 0018-9340. URL: <http://dl.acm.org/citation.cfm?id=626534.627261>.
- [93] Massimiliano Poletto and Vivek Sarkar. “Linear scan register allocation.” In: *ACM Transactions on Programming Languages and Systems* 21.5 (Sept. 1999), pp. 895–913. ISSN: 01640925. DOI: 10.1145/330249.330250. URL: <http://dl.acm.org/citation.cfm?id=330249.330250>.

- [94] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. “A reconfigurable fabric for accelerating large-scale datacenter services.” In: *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE. 2014, pp. 13–24.
- [95] E Quinnell, EE Swartzlander Jr, and C Lemonds. *Three-path fused multiply-adder circuit*. 2011. URL: <http://www.google.de/patents/US8037118?printsec=abstract&hl=de&dq=floating+point+fused+multiply+addition#v=onepage&q&f=falsehttp://www.google.com/patents?hl=en&lr=&vid=USPAT8037118&id=SgD4AQAAEBAJ&oi=fnd&dq=Three-path+fused+multiply-adder+circuit&printsec=abstract>.
- [96] Eric Quinnell. “Floating-point fused multiply-add architectures.” Dissertation. University of Texas at Austin, 2007, p. 163.
- [97] B. Ramakrishna Rau. “Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops.” In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: ACM, 1994, pp. 63–74. ISBN: 0-89791-707-3. DOI: 10.1145/192724.192731. URL: <http://doi.acm.org/10.1145/192724.192731>.
- [98] Hani H. Saleh and Earl E. Swartzlander. “A floating-point fused dot-product unit.” In: *2008 IEEE Int. Conf. on Computer Design*. IEEE, Oct. 2008, pp. 427–431. ISBN: 978-1-4244-2657-7. DOI: 10.1109/ICCD.2008.4751896. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4751896>.
- [99] MS Schmookler and KJ Nowka. “Leading zero anticipation and detection—a comparison of methods.” In: *Computer Arithmetic, 2001*. ... (2001), pp. 7–12. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=930098.
- [100] Daniel Schneider. “Schnelle Fließkommadivision auf aktuellen FPGA-Architekturen.” Bachelor Thesis. Technische Universität Darmstadt, 2014.
- [101] M.J. Schulte and J.E. Stine. “Symmetric bipartite tables for accurate function approximation.” English. In: *Proceedings 13th IEEE Symposium on Computer Arithmetic*. IEEE Comput. Soc, 1997, pp. 175–183. URL: <http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=614893>.
- [102] M.J. Schulte and E.E. Swartzlander. “Truncated multiplication with correction constant [for DSP].” In: *Proceedings of IEEE Workshop on VLSI Signal Processing*. IEEE, 1993, pp. 388–396. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=404467>.

- [103] Richard M Stallman et al. *Using and porting the GNU compiler collection*. Vol. 86. Free Software Foundation, 1999.
- [104] Stretch Inc. *The S6000 Family of Processors*. 2009. URL: http://www.stretchinc.com/_files/s6ArchitectureOverview.pdf (visited on 01/11/2018).
- [105] EE Swartzlander and HHM Saleh. “FFT implementation with fused floating-point operations.” In: *Computers, IEEE Transactions ...* 61.2 (2012), pp. 284–288. URL: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=5669293.
- [106] Synopsys, Inc. *Synphony C Compiler User Guide*. 2011.
- [107] Synopsys, Inc. *VCS Datasheet*. 2017. URL: <https://www.synopsys.com/content/dam/synopsys/verification/datasheets/vcs-ds.pdf> (visited on 12/03/2017).
- [108] Alexandre F Tenca, Song Park, and Lo’ai A Tawalbeh. “Carry-Save Representation Is Shift-Unsafe: The Problem and Its Solution.” In: *IEEE Transactions on Computers* 55 (2006), pp. 630–635. ISSN: 0018-9340. DOI: <http://doi.ieeecomputersociety.org/10.1109/TC.2006.70>.
- [109] Triscend Inc. *Triscend a7s configurable system-on-chip platform Data Sheet 1.10*. 2002.
- [110] Peter JM Van Laarhoven and Emile HL Aarts. “Simulated annealing.” In: *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.
- [111] A K Verma and P Ienne. “Improved use of the carry-save representation for the synthesis of complex arithmetic circuits.” In: *Proceedings of the 2004 IEEE/ACM Int. conf. on Computer-aided design*. IEEE Computer Society, 2004, pp. 791–798. ISBN: 0-7803-8702-3. DOI: 10.1109/ICCAD.2004.1382683. URL: <http://dx.doi.org/10.1109/ICCAD.2004.1382683>.
- [112] Amit Verma, Ajay K. Verma, Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. “Synthesis of Floating-Point Addition Clusters on FPGAs Using Carry-Save Arithmetic.” In: *2010 Int. Conf. on Field Programmable Logic and Applications* (Aug. 2010), pp. 19–24. DOI: 10.1109/FPL.2010.15. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5694214>.
- [113] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. “Designing modular hardware accelerators in C with ROCCC 2.0.” In: *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual Int. Symp. on*. IEEE, pp. 127–134.
- [114] Vince Weaver. *Reading RAPL energy measurements from Linux*. URL: <http://web.eece.maine.edu/~vweaver/projects/rapl/index.html> (visited on 12/31/2017).

- [115] Xilinx, Inc. *AccelDSP User Guide - Xilinx*. 2008. URL: https://www.xilinx.com/support/documentation/sw_manuals/acceldsp_user.pdf (visited on 11/28/2017).
- [116] Xilinx, Inc. *LogiCORE IP Floating-Point Operator v5.0 DS335*. 2011. URL: https://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf (visited on 12/31/2017).
- [117] Xilinx, Inc. *LogiCORE IP Floating-Point Operator v7.0 PG060*. 2011. URL: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf (visited on 01/25/2018).
- [118] Xilinx, Inc. *Vivado Design Suite User Guide - High-Level Synthesis*. 2012.
- [119] Xilinx, Inc. *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors XAPP1079 (v1.0.1)*. 2014. URL: https://www.xilinx.com/support/documentation/application_notes/xapp1079-amp-bare-metal-cortex-a9.pdf (visited on 12/31/2017).
- [120] Xilinx, Inc. *Virtex-6 FPGA Data Sheet: DC and Switching Characteristics DS152 (v3.6)*. 2014. URL: https://www.xilinx.com/support/documentation/data_sheets/ds152.pdf (visited on 11/28/2017).
- [121] Xilinx, Inc. *7 Series DSP48E1 Slice User Guide UG479 (v1.9)*. 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf (visited on 11/28/2017).
- [122] Xilinx, Inc. *7 Series FPGAs Configurable Logic Block UG474 (v1.8)*. 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf (visited on 11/28/2017).
- [123] Xilinx, Inc. *7 Series FPGAs Memory Resources v1.12 UG473*. 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf (visited on 01/06/2018).
- [124] Xilinx, Inc. *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide UG954 (v1.6)*. 2016. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf (visited on 12/31/2017).
- [125] Xilinx, Inc. *UltraScale+ FPGAs Product Tables and Product Selection Guide*. 2017. URL: <https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf#VUSP> (visited on 12/22/2017).

- [126] Xilinx, Inc. *Virtex-7 FPGA Gen3 Integrated Block for PCI Express v4.3 (PG023)*. 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v4_3/pg023_v7_pcie_gen3.pdf (visited on 12/26/2017).
- [127] Xilinx, Inc. *Virtex-7 T and XT FPGAs Data Sheet: DC and Switching Characteristics DS183 (v1.27)*. 2017. URL: https://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf (visited on 11/28/2017).
- [128] Xilinx, Inc. *Xilinx Virtex-7 FPGA VC707 Evaluation Kit*. 2017. URL: <https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html> (visited on 12/25/2017).
- [129] Xilinx, Inc. *Zynq-7000 All Programmable SoC Data Sheet v1.11 (DS190)*. 2017. URL: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (visited on 12/28/2017).
- [130] Xilinx. *Xilinx LogiCORE IP Floating-Point Operator v5.0 Product Specification*. Tech. rep. 2011. URL: http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.
- [131] Xilinx. *Xilinx LogiCORE IP Floating-Point Operator v5.0 Product Specification*. Tech. rep. 2011. URL: http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.
- [132] Tjalling J Ypma. “Historical development of the Newton–Raphson method.” In: *SIAM review* 37.4 (1995), pp. 531–551.
- [133] Ting Yu, Chris Bradley, and Oliver Sinnen. “ODoST: Automatic Hardware Acceleration for Biomedical Model Integration.” In: *TRETS* 9.4 (2016), 27:1–27:24. DOI: 10.1145/2870639. URL: <http://doi.acm.org/10.1145/2870639>.
- [134] Ting Yu, Julian Oppermann, Chris Bradley, and Oliver Sinnen. “Performance optimisation strategies for automatically generated FPGA accelerators for biomedical models.” In: *Concurrency and Computation: Practice and Experience* 28.5 (2016), pp. 1480–1506. DOI: 10.1002/cpe.3699. URL: <https://doi.org/10.1002/cpe.3699>.