TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY
Julkaisu 778 • Publication 778

Claudio Brunelli

Design of Hardware Accelerators for Embedded Multimedia Applications

Tampere 2008

Tampereen teknillinen yliopisto. Julkaisu 778
Tampere University of Technology. Publication 778

Claudio Brunelli

Design of Hardware Accelerators for Embedded Multimedia Applications

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 1st of December 2008, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2008

ISBN 978-952-15-2076-1 (printed)
ISBN 978-952-15-2205-5 (PDF)
ISSN 1459-2045

ABSTRACT

The subject of this work is the design and the implementation of hardware components which can accelerate the computation in a microprocessor-based digital system controlled by a RISC (Reduced Instruction Set Computer) core. Indeed a RISC core alone cannot achieve the desired computational capability needed to meet the requirements of modern applications, especially demanding ones like audio/video compression, image processing and 3D graphics.

Thus some additional dedicated resources are needed to provide the required performance boost; such resources are referred to as accelerators and come in various forms. In particular, this work focuses on a co-processor based approach, which aims at maximizing the modularity and the portability by adding such accelerators as additional components to be coupled with a programmable microprocessor.

Moreover, especially in the embedded systems domain, limited area and energy budgets are key design constraints which make the design of such accelerators even more challenging. Such limitations call for innovative and effective solutions; since the last few years we have been witnessing a growth in popularity of configurable and reconfigurable architectures, due to the fact that they are flexible and reusable. As a matter of fact an important advantage they provide is that they reuse at run time the same physical portion of a chip for different functionalities (provided that they can be time-multiplexed). This way it is possible to fit additional functionalities in the same chip without increasing the area occupation. Above all, they are flexible in that they are able to meet the fast changing needs of the market.

Depending on the flexibility, we could briefly summarize the spectrum of computer architectures like that: parameterized, configurable, reconfigurable, dynamically/run-time reconfigurable, programmable. The term *reconfigurable* has been used quite loosely, sometimes identifying concepts and architectures differing significantly from each other. One possible interpretation indicates machines which can be reconfig-

ured only statically (and not at run time). Those machines are not real reconfigurable architectures, but rather configurable or customizable architectures which can be changed at design time only. Reconfigurable machines, instead, are characterized by dedicated memory cells used to store special bits used possibly for the run-time or off-line reconfiguration of the architecture (the so called configware). Strictly speaking, the configware of reconfigurable machines is changed (possibly) off-line in the field, while in *dynamically reconfigurable architectures* the configware can be changed at run-time, selecting dynamically which parts of the circuits must be used by the current calculations and which ones are unused, how the hardware resources are connected to each other, and so on.

Reconfigurable hardware can be further split in coarse-grain and fine-grain, depending typically on the bit-width of their datapath and on the complexity of the elementary resources which can be selected and interconnected in order to implement a physical circuit to implement a given logical function. The simpler the building logic blocks, the finer the grain. A typical example of fine-grain reconfigurable machines is given by the FPGAs (Field Programmable Gate Arrays): their elementary building blocks are usually relatively simple and dependent on the manufacturer and on the model. Typical elementary blocks that can be found inside FPGA devices are small dedicated memories used as lookup tables (LUTs) in order to implement simple logical functions (usually the locations inside those memories are just a few bits wide), small multiplexers, and carry-chain devices. In FPGAs which are oriented to high-end performance there are also dedicated components like entire multipliers and adders (the so called Digital Signal Processing (DSP) blocks). Such components represent an attempt to bridge the gap between coarse-grain and fine-grain reconfigurable machines (hybrid granularity). Unlike fine-grain reconfigurable architectures, there is no typical example of coarse-grain machines. In this thesis the author tries to give a comprehensive report of such architectures, highlighting the ones which turned out to be particularly meaningful and successful.

Fine-grain architectures are usually featured by the fact that they are more efficient in the usage of the available logic resources when implementing a given logic function. Coarse-grain architectures, on the other hand, show several advantages like energy-efficiency and ease of programming, meeting the data abstraction present in high-level programming languages. Another key point which distinguishes them from fine-grain machines is that due to the presence of large and complicated logic

blocks inside their architecture they are especially suitable for a very efficient implementation of number-crunching applications and computationally heavy tasks. Such applications would require a significant overhead in terms of interconnection usage and latency when mapped on fine-grain machines, which are typically more suited for other application domains.

The research presented in this thesis consists mainly of the design space exploration of two coprocessors which can be used as accelerators for RISC microprocessors.

The first accelerator introduced is a floating-point unit (FPU), while the second one is a coarse-grain reconfigurable machine. Their design space was thoroughly explored using a parametric, synthesizable VHDL (Very High Speed Integrated Circuits Hardware Description Language) model, which was implemented on ASIC (Application Specific Integrated Circuit) standard-cell technologies and on FPGA devices. The implementation on FPGA was useful at first as a fast prototyping platform, but secondly it became also a platform for running real-world applications like Mp3 and H.264 decoders.

When this work started the main specification was to create an open source VHDL description. This lead to specific choices like adopting the IEEE-754 (Institute of Electrical and Electronics Engineers) standard for floating-point arithmetic, which is nowadays widely accepted. For the same reason the code is as much plain and straightforward as possible. A thorough exploration of the possible architectural choices was made, eased also by the parametric nature of the VHDL code created. A series of different implementations lead to a comprehensive description of the trade-off between area and performance, especially related to architectural variations. A tool based on a Graphical User Interface (GUI) for the simulation and debugging of the execution within the FPU was also developed.

The second part of this work describes the architecture of a coarse-grain reconfigurable machine named Butter. This machine was initially meant to be an accelerator to enable running multimedia applications, audio/video processing on a digital system based on a RISC processor. Next, the range was broadened to other application domains like image processing, Global Positioning System (GPS) signal acquisition and tracking, and 3D graphics, enabled by the introduction of special architectural features like support to subword and floating-point operations, which represent an absolute novelty in the field of coarse-grain reconfigurable machines. A GUI-based

tool was developed for the automatic generation of the configware used to configure Butter. An entire System on Chip (SoC) featuring a 32-bit RISC core, Milk FPU and Butter takes 65173 Advanced Look-Up Tables (ALUTs) on a Stratix II EP2S180 FPGA device, and runs at 34 MegaHertz (MHz). Using Butter and Milk, some algorithms can achieve a speed-up (compared to their corresponding software implementation) from one up to two orders of magnitude.

PREFACE

This thesis had been carried out from 2004 to 2008 in the Department of Computer Systems at Tampere University of Technology, Tampere, Finland.

I would like to express my deepest gratitude to Prof. Jari Nurmi for believing in my capabilities, offering me the possibility of being a MSc student at first, then a PhD student within his research group, for his constant supervision and guidance, and for providing part of the economic support that made this work possible.

I also would like to thank the reviewers of this thesis: Prof. Jouni Isoaho from the University of Turku, Finland, as well as Prof. Koen Bertels from the Technical University of Delft, The Netherlands, for their valuable comments and feedback on the draft of this thesis.

I would like to express my deepest gratitude to my parents Primo and Loretta for their unconditioned love, for the help and the support they gave me in difficult moments, for enjoying my joys, and for the efforts they put to turn the young kid I was into the man I am now.

Many thanks to my sister Laura for her love and all the wise advices she has been giving me during my life. Thanks to her husband Davide, who has been like an older brother to me since we met. Thanks to my beloved nephews Annalisa and Eleonora, for being so sweet and for making me feel the meaning of the love of a daughter.

Thanks also to my grandmother Maria for her love, for wishing me well and for spoiling me so often with her compliments. Thanks to my closest relatives, which are my enlarged family and like my second parents: my uncles Eugenio, Bruno, Mauro and their wives Claudia, Lorena, Flaviana. Thanks to my cousins Massimiliano, Valentina, Sheila for being close to me like brothers during all my life, giving me plenty of funny moments and sweet memories which I always keep inside. Thanks to Lisa and Matteo for being always so nice with me, making me feel special: ”I am

glad that you are now also my relatives, besides being my friends".

Many thanks to my dear girlfriend Henrietta for the love and the patience she always showed towards me, for the encouragements she provided me when I needed them most, giving me the confidence and the serenity necessary to face some hard stages of these years: "Thank you for being such a special person and for making my dreams come true". Thanks also to her family for making me feel like their son: Hannele, Simo, Helvi, Henri, Timi and Nea.

A very special thought goes to my grandfather Umberto, to Fabio and Maria, to Sina, and in particular to my grandparents Mario and Dina: "I will never be grateful enough to all of you for your love, for your words of wisdom which helped me facing some difficult moments, for being an example to imitate and for all the wonderful moments I had the privilege to spend together with you".

I also would like to thank my friends from Italy, who are always able to make me feel welcome home whenever I go back there: Marco Valgiusti, Agnese, Enrico ("Asso"), Francesca, Davide ("Dave"), Leonardo ("Leo"), Andrea, Antonio ("Tony"), Alberto, Cinzia, Laura, Elena ("Eli"), Alessandro ("Sandro"), Anna Maria, Angelo, Valentina ("Vale"), Andrea ("Braccio"). Thanks to Marco and Gilberta for all the nice adventures and the funny moments we had together. Thanks to my friends from the high school, with whom I still feel like a teenager when we meet: "Mazza", "Gianluc", "Ciccio", "Paolo", "Marty", "Raffa", "Berto", "Mike", "Capac", "Furia", "Biondo", "Drucci".

I would like to thank Juha Kylliäinen, Markus Moisio and Tuukka Kasanko for being my first friends after my arrival in Tampere, helping me to get quickly acquainted to my new reality. Thanks to Ari Nuuttila, Timo Rintakoski, Irmeli Lehto, Johanna Reponen, Ulla Siltaloppi and Elina Orava for their restless help and assistance in several practical matters.

I also wish to express my gratitude to several colleagues and friends at TUT who shared with me experiences from work and everyday life contributing to my well-being: Juha Pirttimäki, Tapani Ahonen, Andrea Cilio, David, Ugne, Lasse, Tullio, Heikki, Jussi, Hannu, Pekka, Perttu, Heikki, Jussi Raasakka, Jari, Jarno, Mihn, Riku, Fabio Garzia, Federico, Davide, Riccardo, Andrea, Lassi, Chiru, Carmelo, Francesco, Roberto, Alessandro, Vito. A warm acknowledgement goes also to all the trainees who have been visiting our research group during the last years, giving

their contribution to the growth of our research activity.

Many thanks to my friends and colleagues at the University of Bologna: Claudio Mucci, Andrea Lodi, Luca Ciccarelli, Andrea Zivieri, Roberto Canegallo, Massimo Bocchi, Claudia De Bartolomeis, Prof. Roberto Guerrieri and Prof. Eleonora Franchi, and all the others I met there. In particular, I would like to express my gratitude towards Fabio Campi, who was the very first person who introduced me into a real research environment when I was still a student, who made me believe in myself while guiding me as a supervisor: ”Thanks for being a friend besides a guide and a colleague, and for proposing me the MSc experience in Finland, enabling the actual beginning of my professional career and of a wonderful life experience”.

Thanks also to the numerous Finnish and foreign exchange students whom I got to know here in Tampere; in particular Marco Paolieri, Domenico (”Dom”), Federico (”Fede”), Filippo (”Filo”), Fabio (”Fibiello”), Gaetano, Elisa, Michele, Ilana, Aino, Heikki, Juho, Cesare, Francesco (”Franti”), Guido, Paolo and all the others. Thanks to Tapani, Suvi and their families, which gave me the first unforgettable insight into Finnish home atmosphere and traditions. The same applies to Maarit, Tuomas, Jenni, Antti, and all the other Finnish guys. Thanks to all the people who could not be explicitly mentioned here for lack of room, but contributed anyway to my growth and wished me well.

This work was financially supported by scholarships by GETA Graduate School and by the Department of Computer Systems at Tampere University of Technology, as well as by Nokia Foundation’s and Ulla and Yrjö Neuvo Foundation’s research grants; all of these assotiations are gratefully acknowledged.

Tampere, 2008

Claudio Brunelli

TABLE OF CONTENTS

LIST OF PUBLICATIONS

This thesis is based on the work described in a bundle of reprinted publications. The bundle containing the publications listed below is enclosed in part II. These publications are referred to in the text as [P1], [P2], ..., [P16].

[P1] M. Bocchi, C. Brunelli, C. De Bartolomeis, L. Magagni and F. Campi, "A system level IP integration methodology for fast SoC design", in *Proceedings of the International Symposium on System-on-Chip*, (SoC2003), Tampere, Finland, 19–21 November 2003, pp. 127–130.

[P2] C. Brunelli, F. Campi, J. Kylliäinen and J. Nurmi, "A Reconfigurable FPU as IP component for SoCs", in *Proceedings of the International Symposium on System-on-Chip*, (SoC2004), Tampere, Finland, 16–18 November 2004, pp. 103–106.

[P3] C. Brunelli, P. Salmela, J. Takala and J. Nurmi, "A Flexible Multiplier for Media Processing", in *Proceedings of the IEEE International Workshop on Signal Processing Systems*, (SiPS2005), Athens, Greece, 2–4 November 2005, pp. 70–74.

[P4] C. Brunelli, F. Garzia, C. Mucci, F. Campi, D. Rossi and J. Nurmi, "A FPGA Implementation of An Open-Source Floating-Point Computation System", in *Proceedings of the International Symposium on System-on-Chip*, (SoC2005), Tampere, Finland, 15–17 November 2005, pp. 29–32.

[P5] C. Brunelli, F. Cinelli, D. Rossi and J. Nurmi, "A VHDL Model and Implementation of a Coarse-Grain Reconfigurable Coprocessor for a RISC Core", in *Proceedings of the International Conference on Ph.D. Research in Microelectronics and Electronics*, (PRIME2006), Otranto, Italy, 12–15 June 2005, pp. 229–232.

[P6] C. Brunelli, F. Garzia and J. Nurmi, "A Coarse-Grain Reconfigurable Machine With Floating-Point Arithmetic Capabilities", in *Proceedings of the International workshop on Reconfigurable Communication-centric Systems-on-Chip*, (ReCoSoC2006), Montpellier, France, 03–05 July 2006, pp. 1–7.

[P7] C. Brunelli and J. Nurmi, "Design and Verification of a VHDL Model of a Floating-Point Unit for a RISC Microprocessor", in *Proceedings of the International Symposium on System-on-Chip*, (SoC2006), Tampere, Finland, 14–16 November 2006, pp. 87–90.

[P8] F. Garzia, C. Brunelli, L. Nieminen, R. Mastria and J. Nurmi, "Implementation of a Tracking Channel of a GPS Receiver on a Reconfigurable Machine", in *Proceedings of the International Conference on Computer as a Tool*, (EUROCON2007), Warsaw, Poland, 9–12 September 2007, pp.875–881.

[P9] C. Brunelli and J. Nurmi, "Co-processor Approach to Accelerating Multimedia Applications", in Jari Nurmi Ed. *Processor Design – System-on-Chip Computing for ASIC and FPGAs*, Springer, 2007, pp.209–228.

[P10] C. Mucci, F. Campi, C. Brunelli and J. Nurmi, "Programming Tools for Reconfigurable Processors", in Jari Nurmi Ed. *Processor Design – System-on-Chip Computing for ASIC and FPGAs*, Springer, 2007, pp.209–228.

[P11] F. Garzia, C. Brunelli, A. Ferro and J. Nurmi, "Implementation of a 2D Low-Pass Image Filtering Algorithm on a Reconfigurable Device", in *Proceedings of the International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, (ReCoSoC2007), Montpellier, France, 18–20 July 2007, pp.166–170.

[P12] F. Garzia, C. Brunelli, D. Rossi and J. Nurmi, "Implementation of a floating-point matrix-vector multiplication on a reconfigurable architecture", in *Proceedings of the 15th Reconfigurable Architecture Workshop* (RAW 2008), Miami, Florida, USA, April 14-15 2008, pp.1–6.

[P13] C. Brunelli, F. Garzia, C. Giliberto and J. Nurmi, "A Dedicated DMA Logic Addressing a Time Multiplexed Memory to Reduce the Effects of the System Bus Bottleneck", in *Proceedings of the International Conference on Field Programmable Logic and Applications*, (FPL 2008), Heidelberg, Germany, 8–10 September 2008, pp. 487–490.

[P14] C. Brunelli, F. Garzia, J. Nurmi, F. Campi, and D. Picard, “Reconfigurable Hardware: the Holy Grail of Matching Performance with Programming Productivity”, in *Proceedings of the International Conference on Field Programmable Logic and Applications*, (FPL 2008), Heidelberg, Germany, 8–10 September 2008, pp. 409–414.

[P15] C. Brunelli, F. Garzia and J. Nurmi, “A Coarse-Grain Reconfigurable Architecture for Multimedia Applications Featuring Subword Computation Capabilities”, in *Journal of Real-Time Image Processing*, Springer-Verlag, 2008, 3 (1-2): 21-32. doi:10.1007/s11554-008-0071-3.

[P16] C. Brunelli, F. Campi, C. Mucci, D. Rossi, T. Ahonen, J. Kylliäinen, F. Garzia and J. Nurmi, “Design space exploration of an open-source, IP-reusable, scalable floating-point engine for embedded applications”, in *Journal of Systems Architecture*, Elsevier, 2008, doi:10.1016/j.sysarc.2008.05.005.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
ALUT	Advanced Look-Up Table
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Integrated Processor
CGRA	Coarse-Grain Reconfigurable Architecture
CISC	Complex Instruction-Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chip MultiProcessor
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DDD	Data Display Debugger
DFG	Data Flow Graph
DMA	Direct Memory Access
DSP	Digital Signal Processing
EDA	Electronic Development Automation
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array

FPU	Floating-Point Unit
FU	Functional Unit
GCC	GNU C Compiler
GHz	GigaHertz
GNU	Gnu's Not Unix
GPS	Global Positioning System
GUI	Graphical User Interface
HDL	Hardware Description Language
HW	HardWare
IC	Integrated Circuit
IDCT	Inverse Discrete Cosine Transform
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
ILP	Instruction Level Parallelism
I/O	Input/Output
IP	Intellectual Property
ISA	Instruction-Set Architecture
LLP	Loop Level Parallelism
LUT	Look-Up Table
MHz	MegaHertz
MIMD	Multiple Instruction Multiple Data
MMX	MultiMedia eXtension
MPSoC	MultiProcessor System-on-Chip

NoC	Network-on-Chip
NOP	No-OPeration
NREC	Non-Recurring Engineering Costs
OCP	Open Core Protocol
OS	Operating System
RAM	Random Access Memory
RISC	Reduced Instruction-Set Computer
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SMT	Simultaneous MultiThreading
SoC	System-on-Chip
SSE	Streaming SIMD Extensions
SW	SoftWare
TLP	Task Level Parallelism
TTA	Transport Triggered Architecture
TUT	Tampere University of Technology
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Scale Integration Circuit
VLIW	Very Long Instruction Word

Part I

ARGUMENTATION

1. INTRODUCTION

In everyday life we all deal with a number of integrated digital systems which can be found in most of the available consumer electronics items. The heart of such systems is usually a programmable microprocessor core, which is mainly used to control all the components surrounding it (peripherals). Programmable microprocessors are in general characterized by high flexibility, which is usually exploited to perform control operations. On the other hand, they are usually not performing well enough to satisfy hard real-time requirements typical of embedded applications. In embedded systems the need for real time performance in complex applications is becoming more and more acute, thus calling for solutions aimed at delivering performance, yet guaranteeing modularity and flexibility. For this reason programmable microprocessors are usually augmented with special Functional Units (FUs), turning them into Application Specific Integrated Processors (ASIP), or with dedicated coprocessors, Digital Signal Processors (DSPs) or Application Specific Integrated Circuits (ASICs). The coprocessor approach is one of the most popular approaches which are usually followed, due to its scalability and modularity.

The key problems that this thesis tries to face are numerous. First of all, in embedded systems design there are multiple issues in managing the difficulty of designing, implementing and debugging complex systems. This takes a lot of energy and time, and involves employing large design teams. A solution which has been adopted in order to improve the situation consists in decomposing large systems into sub-parts, which can be designed and tested separately. Each of these parts is referred to as Intellectual Property (IP) component.

Strictly related to these problems, there are other issues that become more and more problematic: the ability to provide designs which are as flexible and reusable as possible. In the past, designers were designing digital circuits at transistor level of ab-

straction. This ensures usually high performance and reduced area occupation, but implies a long design process, long verification time, and high Non Recurrent Engineering Costs (NREC). Long design and verification implies long time to market; designers felt the need for new means and tools helping them bridge the gap between applications and transistor-level designs. One of the most successful solutions which has been proposed is based on Hardware Description Languages (HDLs): these languages have a syntax which resembles regular high-level programming languages like C, but they are profoundly different in that they do not describe an algorithm implemented in software, but rather a hardware circuit. HDL languages have been vastly supported by companies and research groups specialized in Electronic Development Automation (EDA) tools: these tools allow the engineers to verify quickly the HDL code they produced, and to implement them in hardware circuits in an automatic way. One of the main advantages of such design flow is that the same code can be implemented into different microelectronics technologies, enhancing reusability.
Still, until some time ago most of the hardware designs were implemented as Application Specific Integrated Circuits (ASIC) components: the functionality implemented in the circuit is fixed, thus cannot be changed without going through the entire design process once again. This fact has became unacceptable, as modern applications change more and more frequently. Thus designers have been forced to consider other alternatives which can guarantee flexibility. Typically, software-based solutions guarantee the maximum level of flexibility; on the other hand, software running on programmable microprocessor usually does not accomplish the speed requirements dictated by applications. A multitude of solutions have been then envisioned, ranging from simple DSP processors, to ASIPS, until true Multiprocessor System-on-Chip (MPSoC). A relatively new and innovative approach to the problem is represented by reconfigurable hardware. This means that the reconfigurable machines allow the user to make modifications to the datapath of their architecture by means of dedicated configuration bits, usually referred to as *configware*. Reconfigurable hardware is usually hosted as a coprocessor inside a system powered by a programmable processor which controls the computational and Input/Output (I/O) activity of the reconfigurable machine. The reconfigurable processor is tailored to perform specific, computationally intensive tasks as fast and efficiently as possible. The main goal is to guarantee high performance (almost as high as the one achievable using ASIC accelerators) together with high flexibility and reusability of the same hardware.
There are several examples of reconfigurable hardware: the term can indicate quite

a broad range of products and solutions, ranging from automatic generation of configurable Systems-on-Chip (SoC), to Field Programmable Gate Arrays (FPGA). The first case is for instance addressed by companies like Tensilica, which pursued the philosophy of fast SoC development starting from application source code. This represents a very powerful and convenient solution to overcome the problems related to tight time to market; on the other hand, strictly speaking, such an approach does not represent a real reconfigurable solution, but a configurable solution.
On the other hand the second case represents one of the most popular example of run time reconfigurable hardware. FPGAs are devices which are made up of elementary blocks named ”logic blocks” or ”logic ”elements” organized in a large matrix. Such blocks usually contain special Look-Up Table (LUT) Memories which allow them to be ”programmed” for implementing elementary logic operations. Modern FPGAs feature also special, coarse logic elements named ”DSP blocks”: they are meant to implement in an efficient way multi-bit arithmetic operations that would otherwise require a large number of logic elements. The logic elements are interconnected using a network of wires featuring programmable switches, allowing the reconfiguration of the interconnection topology.
In this thesis the author concentrates in particular on coarse-grain, run-time reconfigurable hardware architectures: the circuits are obtained by instantiating ”chunks” of dedicated arithmetic logic (i.e. 32-bit multipliers, adders, etc.) interconnected using a reprogrammable network. The network and the functionalities of the circuit are specified by dedicated bits, which are usually named as ”configware”. Such an approach tries to guarantee high performance (especially in multimedia and DSP applications) and at the same time a certain degree of flexibility and reusability. On the other hand, sometimes it introduces area overhead, when dealing with data using only a few bits. For this reason some research groups make usage of fine-grain reconfigurable hardware; still, in that case new problems arise, like need for large amount of configware and reduced efficiency in performing arithmetic operations.
The field of coarse-grain reconfigurable hardware is still at its beginning and under development, and one of the issues there is mapping applications to the hardware in an optimized and automatic way. Several different approaches have been proposed, but none of them is yet completely mature and totally automated.
Modern applications (like for instance 3D graphics, automotive, Mp3 audio codecs) require floating-point arithmetic in order to guarantee a required degree of quality. On the other hand, floating-point calculations are usually demanding operations which

imply long execution times, conflicting with the real-time constraints of those applications. Thus more and more frequently such applications rely on the support of hardware accelerators in order to satisfy real time constrains on execution time. For this reason, a series of dedicated circuits named Floating-Point Units (FPUs) have been proposed. The first ones were introduced as coprocessors already at the time of the Intel 8086 microprocessor: its FPU, named 8087, was a separate chip placed on the mother board of the host computer, and was able to enhance a few times the performance of the CPU. Advances in microelectronics technology made it possible to integrate the FPU directly inside the CPU chip (already in the Intel Pentium processor), with consequent advantages in terms of speed and power consumption. This process lead to more and more complex and powerful architectures integrated on the same chip, enabling superscalar and Single Instruction Multiple Data (SIMD) instructions, giving birth to MMX instructions and SSE instructions (in Pentium III) and SSE2 and SSE3 instructions (in Pentium IV). However, in embedded systems FPUs appeared relatively late, due to the fact that in the embedded domain area and power consumption constraints are particularly strict, and did not allow for the insertion of relatively large and complex hardware. Another reason is that until a few years ago the applications usually running on embedded system did not require floating-point calculations; nowadays things changed a lot, forcing the developers to introduce at first software emulation of floating-point instructions, then real hardware accelerators, which are particularly optimized for the embedded domain.

Still, a characteristic of most of the existing FPUs is that they are proprietary designs. Indeed, most of the FPUs and Coarse-Grain Reconfigurable Architectures (CGRAs) currently available come from the industry, available to the end users only under payment of royalties. Even so, the designs are usually delivered as ”black boxes”, preventing the customer from understanding the inner details, and from customizing them. Such a situation is not always convenient, especially in the academic world, where users usually have limited budgets; what's more, students and researchers want to have access to the details of the architectures they deal with, for educational or research purposes. Also corporate customers like the possibility of having full access to the inner details of the components they purchase, so that they can customize and optimize them for their needs.

Some other FPUs come from the academia, but they are not freely available outside the research group where they were developed. Only a couple of them are open-source designs; still, they are either only simple functional units (and thus not real

coprocessors) or they implement only a subset of the IEEE-754 standard of Floating-Point arithmetic.
One of the goals of this thesis was to provide the first complete, IP reusable, scalable open source FPU available today.

1.1 Objective and Scope of Research

The research described in this thesis was carried out in a group focusing on digital signal processing (DSP) and digital communication systems-on-chip (SoCs); most of the related work is described in the publications enclosed as part II of this thesis.

This thesis describes the design exploration of two hardware accelerators, which are used as coprocessors for RISC (Reduced Instruction Set Computer) microprocessors: a Floating-Point Unit (FPU) and a coarse-grain reconfigurable architecture (CGRA) for multimedia and DSP applications. This choice has been done in the attempt of covering a large part of the ”hot” applications which are most widely implemented nowadays in embedded systems, and which demand high performance hardware.
The author decided to follow the IP-based approach in designing the proposed architectures, due to the problematic mentioned above. Thus, the first target of this work was to provide stand-alone IP components which could be easily plugged into an embedded system powered by a programmable microprocessor, in order to augment the overall performance of the system itself in executing DSP and multimedia applications.
The author planned a design space exploration to guarantee an efficient implementation of these two different accelerators. The author decided to design both of them using the Very High Scale Integration Circuit Hardware Description Language (VHDL), which is a popular example of HDL language. The choice of using VHDL was done in order to provide an easy way of simulating and implementing them on different target technologies, as well as enabling portability and general validity of the models developed. In particular, the prototyping of the proposed designs on Field Programmable Gate Array (FPGA) boards becomes relatively easy and immediate.
The author aimed at making the proposed FPU such that it is supported by the GNU C Compiler (GCC), so that the execution of applications written in C programming language could be obtained in an automatic way. Virtually any applications written

in C could be compiled and run either on a simulator, or directly on FPGA boards, in order to achieve an efficient testing and benchmarking.
The author aimed also at providing a comprehensive and self-contained open-source solution, thus to provide a complete and free package which can be used either as it is, or easily customized by any user in order to achieve his design targets.
In general, most of the available FPUs lack some features like full compliance with the IEEE-754 Standard on Floating-Point Arithmetic; this is because a complete implementation requires a relatively complex hardware, which is not always required in embedded platforms. Especially in dedicated, commercial FPUs extreme optimizations are applied, trading generality for a smaller area on chip. The author felt instead that in an open-source design such a feature is mandatory; then the implementation of the proposed FPU aimed at achieving total compliance with the IEEE standard, trying to highlight the related costs. This solution enables the ease of use and compatibility of the proposed FPU.
Moreover, one of the drawbacks which affects the usage of floating-point hardware is its relatively large area occupation, especially when double precision is implemented. Still, double precision is usually needed only in scientific calculations, while the applications mentioned above need only single-precision. Moreover, all the challenges and problematic emerge already when implementing single precision arithmetic; the double precision just introduces larger area occupation and slower operations. For these reasons, the author concentrated only on studying single-precision implementations.
Another important issue which this thesis tries to address is the need for generality; many solutions are implemented as ASIC circuits, or as embedded solutions anyway. This usually optimizes performance, but hampers the generality and the reusability of designs. The author proposed a synthesizable VHDL model of the accelerators which can be implemented on any target technology with no modifications and still can be tuned by the user in order to get an optimized cost/performance ratio.
Another aim of this work is to provide software tools which help the user in developing applications. For this reason the author aimed at providing a Debugger which allows the programmer to set breakpoints, to have an insight of the internal registers of Milk, its status, etc.

A similar effort was put in the development of the reconfigurable machine: the author

planned to obtain a tool which could allow the user to represent graphically the algorithms to be mapped on the accelerator, generating automatically the corresponding configware. This would be a first step toward the bridging of the application world towards the hardware implementation. Further goals were to reduce the area overhead typical of the CGRAs, trying to find a way to better exploit the reuse of its internal resources. In this sense the author aimed at proposing a CGRA which features SIMD subword operations: multiple 8-bit and 16-bit operations can be carried on in parallel by the very same hardware which implements single 32-bit integer operations. This choice represents (together with ADRES, as explained further in this thesis) the coarsest grain available today in reconfigurable machines, allowing a thorough study of the potential of this choice and the related trade-offs. The author wanted to push the research further, proposing the first CGRA ever able to elaborate floating-point numbers. This choice was never proposed before due to the relatively large area occupation implied by floating-point hardware. The author wanted to demonstrate that it is possible to have such a feature without increasing significantly the area occupation by properly reusing existing hardware resources. At the same time, this choice opened up the horizon for an unprecedented usage of CGRAs in several application domains.

Summarizing, the main objectives of this thesis could be briefly summarized as follows.

- Provide coprocessors to accelerate a programmable microprocessor in executing a series of algorithms. The range of such algorithms should be as large as possible, while the amount of hardware resources necessary to achieve this goal should remain within reasonable limits.

- Design them in such a way that makes them easily reusable in other systems. For this reason, the author designed the coprocessor to be stand-alone IP components.

- Make them as flexible as possible, enabling an easy customization by the user. For this purpose, the author chose to use the VHDL language to design the proposed accelerators, so that the customization of the design could be eased by using built-in functionalities of the VHDL language like the so called *generics*. Furthermore, the usage of synthesizable VHDL guarantees immediate porta-

bility of the design over different microelectronics technologies, either ASIC standard cells or FPGAs.

- Combine more traditional accelerators (like FPUs) with cutting-edge research ones (like reconfigurable machines).
- Make the FPU so that it supports well known compilers like GCC, and well known standards like the IEEE-754.
- Provide the CGRA with a dedicated software tool which could ease its usage.
- Explore the potentiality of the CGRA providing it with special features aimed at enhancing its performance while reusing existing resources.

1.2 Thesis Outline

This thesis is based on a bundle of international publications reprinted in part II. Part I has an introductory nature containing background information to better understand the motivation for and the content of the published work. The published work is self-explanatory and it is only referred to from part I where necessary.

The rest of this part (part I) is organized as follows. Chapter 2 discusses the need for accelerators in embedded systems and the approaches commonly taken to design and use them. Chapter 3 introduces the design of the first accelerator: a floating-point unit (FPU). Chapter 4 introduces the design of the second accelerator: a coarse-grain reconfigurable machine. The enclosed publications are summarized and the author's contribution to the published work is highlighted in chapter 5. Conclusions are drawn in chapter 6.

2. ACCELERATING RISC MICROPROCESSORS

The purpose of the following discussion is to illustrate the most important issues affecting the design of accelerators for RISC microprocessors in embedded systems. The basics of accelerators are briefly introduced in section 2.1.

In this chapter and the rest of this thesis the term application-specific integrated circuit (ASIC) is used to denote libraries of standard cells, that is ready-made elementary digital circuits that can be handled automatically by synthesis Electronic Design Automation (EDA) tools, and cell library-based designs (unless differently specified).

2.1 Accelerators

Nowadays users usually want their portable devices (PDAs and mobile phones) to support complex multimedia and 3D graphics applications, besides other ones commonly referred to as general purpose computations [1]. Multimedia applications need to squeeze as much performance as possible out of the computation system, and standard programmable microprocessors (like RISC cores) cannot provide the necessary computational capabilities. For this reason, computer architectures shifted from the usage of conventional microprocessors to entire Systems on a Chip (SoC). Inside SoCs, a main core (which can be either a simple RISC microprocessor or a powerful DSP microprocessor) is usually hosted together with a set of other components ranging from I/O peripherals and Direct Memory Access (DMA) controllers to memory blocks and dedicated computation engines (coprocessors and dedicated processors tailored to accelerate a precise set of algorithms) [2]. At the beginning such dedicated accelerators were usually custom ASIC blocks; more recently the fast pace at which applications and standards are changing has pushed SoC designers to go for components which are programmable and flexible, while providing at the same time a significant speed-up. The topic of microprocessor acceleration is so wide (ranging

from the diverse metrics which have been proposed to measure performance in a fair and coherent way, going through the different types of algorithmic exploitations, to the analysis of different architectures of accelerators) that even an entire book could hardly cover it in a proper way. This chapter thus will not provide an exhaustive dissertation about all the existing typologies of accelerators, but will rather try to give a brief overview of the main issues related to the topic of microprocessor acceleration, taking then a deeper insight into one particular type of accelerators which has been used for many years and is still used in several designs thanks to its modularity and ease of use: the coprocessors.

2.1.1 Accelerators and different types of parallelism

Several different architectural solutions aiming at high performance computation have been presented in the history of microprocessors. In principle, each of those solutions tries to achieve computational efficiency by exploiting properties of the applications running on them. One of such key properties is the *parallelism*, which can be interpreted at several levels:

- Instruction Level Parallelism (ILP)
- Loop Level Parallelism (LLP)
- Task Level Parallelism (TLP)
- Program Level Parallelism
- Data parallelism

As summarized in [7], to each type of parallelism there is a corresponding architecture that is aimed at exploiting it.
ILP expresses the possibility of executing more instructions at the same time. There are several architectural techniques which exploit ILP; the most common ones are: pipelining, superscalar processors and Very Long Instruction Word (VLIW) processors. Pipelining is based on overlapping in time the execution of multiple instructions by splitting it into multiple stages; each stage has a physical support in the processor datapath, and is separated from the others using registers. Pipelining improves the throughput of the processor, ideally providing a theoretical Cycles Per Instruction

(CPI) equal to one. In practice, such performance is never reached due to (the so called) *hazards*. Hazards are originated by data (and control) dependencies between instructions and by limited physical resources present in the processor. Hazards are tackled using a number of techniques which are not reported here for the sake of brevity. A basic pipelined processor which issues sequentially one instruction per cycle is also called scalar processor [8].

Superscalar processors instead extend each of the pipeline stages so that multiple instructions can be executed in each stage, which require then multiple functional units in parallel. For this reason superscalar processors belong (as well as VLIW processors) to the category of the so called *multiple-issue processors*. Superscalar processors fetch multiple instructions at a time and selectively issue a variable number of instruction words on the same instruction cycle [9]. The instructions are then temporarily stored in (the so called) instruction queue, from where they are later issued in packets to the functional units for elaboration. The selection for execution of the instructions stored in the queue can happen in two ways: in *static scheduling* the instructions are selected from the beginning of the queue, while in *dynamic scheduling* they can be issued even in an out-of-order fashion (in-order processors can also be dynamically scheduled though). In superscalar processors it is the hardware itself which selects which instructions are to be issued and when; this leads to an increased area occupation, even though it on the other hand leverages the compiler from most of the optimization burden.

VLIW processors are multiple-issue processors like superscalar ones, but are characterized by significantly wider instruction words. A wide instruction word is made up of several RISC-like instructions which are packed together at compile time. Each slot in the wide instruction corresponds to a functional unit in the datapath of the processor; due to data and control dependencies it is not always possible to create a wide instruction having meaningful operations for each functional units, leading to the necessity of having some slots filled with a no-operation (NOP). This leads to the main drawback of VLIW processors: they imply wasting of instruction memory space, even though several solutions to that have been proposed; they are manly based on dedicated compression/decompression techniques and compiler optimization. Moreover, the problem of large program size can be alleviated by using instruction compression techniques: a normal VLIW word is compressed into a variable-length word, which is then de-compressed to the original format only when the instruction is actually to be executed [10]. Another drawback is that they rely heavily on the capabilities

of the related compiler; this feature, on the other hand, is also the reason why VLIW processors are relatively simple, small and cheap, which is their main advantage.

Loop Level Parallelism is present when consecutive loop iterations can be executed in parallel. A possible way of exploiting LLP consists in converting it into instruction-level parallelism by using techniques like *loop unrolling* and *software pipelining*.

Task level and program level parallelism can be usually exploited by the operating system (OS). Task Level Parallelism finds a suitable application in very complex systems, like multi-processor SoCs, even though it is still applicable also to single-processor systems. In the latter case the processor runs multiple threads, switching the execution between active and idle processes; thus only one program thread is executed at a time. To rise the TLP introduced there are mainly two possibilities:

- simultaneous multithreading (SMT) [11]. Used in wide-issue superscalar processors: issue slots which are unused by a single thread are filled with instructions from the other program threads.
- chip multiprocessors (CMP). The idea consists of executing multiple threads in parallel on multiple processor cores.

SMT and CMP belong to the family of MIMD (Multiple-Instruction Multiple Data) computers [16]. SMT is very flexible but requires complicated hardware support. CMP, on the other hand, has the great advantage of allowing scalability, together with intensive reuse of existing processor cores: this leads to significant savings in terms of design and verification time [13].

To take advantage of data parallelism instead, SIMD, systolic, and vector architectures have been introduced [3] [4]. The basic idea behind SIMD computation consists of performing in parallel the same arithmetic instruction on a set of operands. SIMD computation is very popular in the domains of multimedia processing, where data are either 8 or 16 bits wide: this allows for the usage of subword precision in existing 32-bit (or wider) arithmetic functional units. This way, a 32-bit ALU can be used to perform a given arithmetic operation on two 16-bit operands or four 8-bit

operands, boosting performance without instantiating additional arithmetic units. Of course, a N-bit functional unit supporting subword SIMD computations is slightly larger than the corresponding "classical" implementation, due to the presence of special resources used for internal routing and support for saturating arithmetic. SIMD instructions improve significantly the performance in the execution of popular kernels like motion estimation and IDCT [17]. In order to enhance the performance in executing 3D graphics applications SIMD-like instructions were introduced to allow two single-precision floating-point instructions to be executed concurrently (*paired instructions*). This idea, together with subword SIMD computation, stands at the base of *Publication* [P3], where both capabilities are present inside a double precision floating-point multiplier: the same hardware supports either the execution of a double precision multiplication, two (paired) single precision multiplication, plus 32-bit (or subword) integer multiplication, addition and sum of products.

2.1.2 Processor architectures and different approaches to acceleration

A special family of microprocessors is made up of Digital Signal Processing (DSP) processors: they try to guarantee high performance by executing a very specific category of algorithms while keeping some flexibility relying on the fact that they can be programmed using high-level languages. In the 90s also RISC microprocessors became very popular, to the point that they replaced some CISC architectures (which were very widely adopted because of the fact that they needed a relatively small amount of program memory to run applications, due to the high code density provided by their complex instructions). The success of ASIC and SoCs eased the advent of post-RISC processors, which are usually generic RISC architectures, augmented with additional components. Several different approaches have been proposed to implement the acceleration of a microprocessor, but in general the main idea consists of starting from a general-purpose RISC core and adding extra components like dedicated hardware accelerators (for example, this approach is followed by Tensilica with their Xtensa architecture [5]). This is because it is good to keep a certain degree of software programmability to keep up with applications and protocols changing fast, so having a programmable core in the system is recommendable to guarantee general validity and flexibility to the platform. Another possible way of accelerating a programmable core consists of exploiting instruction and/or data parallelism of applications by providing a processor with VLIW or SIMD extensions; yet another way

consists of adding special functional units (for example MAC circuits, barrel shifters, or other special components designed to speed up the execution of DSP algorithms) in the data-path of the programmable core (usually a RISC core or a DSP) [5]. This way the instruction set of the core is extended with specialized instructions aiming at speeding up operations which are both heavy and frequent. This approach is anyway not always possible or convenient, especially if the component to plug into the pipeline is very large. Moreover, large ASIC blocks are nowadays not so convenient in the sense that they usually cost a lot and lack flexibility, so that they become useless whenever the application or the standard they implement changes. Many general purpose microprocessors thus come with a special interface meant to ease the attachment of external accelerators; there are basically two possibilities:

- using general-purpose accelerators to be used for a broad range of applications
- using very large, powerful, run-time reconfigurable accelerators

The design and verification issues related to coprocessors can be faced independently from the ones related to the main processor: this way it is possible to parallelize the design activities, saving then time. In case the core already exists before the coprocessors are designed the coprocessors can be just plugged into the system as black boxes, with no need to modify the architecture of the processor.

2.1.3 Common applications and related constraints for hardware systems

These applications are then to be carefully analyzed to determine an optimal way to map them to the hardware available. Since normally applications are made up of a control part and a computation part, the first stage usually consists of locating the computational kernels of the algorithms. These kernels are usually mapped on dedicated parts of the system (namely dedicated processing engines), optimized to exploit the regularity of the operations processing large amounts of data, while the remaining part of the code (the control part) is implemented by software running on a regular microprocessor. Sometimes special versions of known algorithms are set up in order to meet the demand for an optimal implementation on hardware circuits. Different application domains call for different kinds of accelerators, e.g. applications like robotics, automation, Dolby digital audio, and 3D graphics require

floating-point computation [1] [5] [6] making thus the usage of floating-point unit (FPU) very useful and sometimes unavoidable. To cover the broad range of modern and computationally demanding applications like imaging, video compression, multimedia, we also need some other kind of accelerator: those applications usually benefit from large vector architectures able to exploit the regularity of data patterns while satisfying the high bandwidth requirements. A possibility consists of producing so called multimedia SoCs, which usually are a particular version of multiprocessor systems on chip (MPSoCs) containing different types of processors, which satisfy far better the requirements than homogeneous MPSoCs. Such machines are usually quite large, so a very effective way (which is widely used nowadays) of solving this problem is to make those architectures run-time reconfigurable. This means that the hardware is done so that the data-path of the architecture can be changed by modifying the value of special bits, named configuration bits or configware. One first example of reconfigurable machines which became very popular is given by FPGA devices, which can be used to implement virtually any circuit by sending a specific pattern of configuration bits to the device [14] [15]. The idea of reconfigurability was developed further, leading to custom devices used to implement powerful computation engines; this way it is possible to implement several different functionalities on the same component, saving area and at the same time tailoring the hardware at run time to implement an optimal circuit for a given application. Reconfigurability is an excellent mean of combining the performance of hardware circuits with the flexibility of programmable architectures. Accelerators come in different forms and can differ a lot from each other: differences can relate to the purpose for which they are designed (accelerators can be specifically designed to implement a single algorithm, or can instead support a broad series of different applications), their implementation technology (ASIC custom design, ASIC standard-cells, FPGAs), the way which they interface to the rest of the system, and their architecture.

3. FLOATING-POINT UNITS

The purpose of the following discussion is to illustrate the most important issues concerning the design of FPUs, and to describe in detail the one developed by the author.

3.1 Introduction

As stated in chapter 2, more and more often modern applications require the usage of floating-point arithmetic. Historically, the main drawback related to floating-point arithmetic consists of its significantly higher complexity when compared to integer arithmetic. Such complexity translates into very long execution time for software implementations of algorithms based on floating-point arithmetic. For this reason special devices were introduced, able to implement the basic floating-point operations in hardware in much less time; such devices are commonly known as ”floating-point units” (FPUs). Again, the complexity of floating-point arithmetic leads to a relatively large area of FPUs; this point usually discouraged designers from including them into embedded systems, at least until some years ago, because of the strict area constraints which usually apply to that domain. In the past, also personal computers had the FPU as an expensive extra; the FPU was a separate chip since it was too costly to integrate it with the main Central Processing Unit (CPU). Nowadays, thanks to the advances in microelectronics technology, that problem is becoming less acute, so FPUs appeared also in embedded and area-constrained devices. There exist different types of floating-point units, ranging from proprietary to open-source ones, either supporting the IEEE-754 standard or not, capable of performing single-precision or double precision computation, for usage with CISC or RISC machines; they can come either as external coprocessors or internal functional units.

3.2 MILK coprocessor: aims

The author designed a floating-point unit (FPU) named Milk. Milk handles all the floating-point instructions while the main microprocessor executes only general-purpose and control-flow code. Milk is an IP component which can be plugged as it is inside a host system. The approach of using IP components to build a complex System-on-Chip (SoC) is very convenient: each IP component can be designed either as a full-custom ASIC circuit or as a block described at high abstraction level [23] using hardware description languages like VHDL. Indeed, designing a system-on-chip (SoC) is a complex task [24]. Because of this complexity, the philosophy of reusing pre-made and pre-verified IP blocks [25] [26] has been adopted [27]. A good point in describing the IP blocks using an HDL language (like VHDL) is that the VHDL code can be handled by dedicated logic synthesis tools, which select the best physical resources to efficiently implement the described hardware [28]. The author created an open-source, fully parametric VHDL model for our FPU, which is characterized by flexibility and the ability to easily and immediately customize it at will of the user. Other similar components are available on the market, but they usually are not easily adaptable to different implementations, while our design is highly parametric: the author used extensively VHDL generics in the code, so that a series of key features of our design can be by simply modified by them.
One of our primary targets was to provide a device as reusable and scalable as possible, due to its open-source nature. For this reason the first design choice was to be fully compliant with the IEEE-754 standard for floating-point arithmetic. This is not always true for other FPUs, especially for the proprietary ones, which trade general validity for performance: for example, some FPUs elaborate floating-point operands which are expressed using a 16-bit or 24-bit format rather than 32-bit or 64-bit ones. This is a way of simplifying the hardware (which becomes smaller) but forces the adoption of a non-standard representation of floating-point numbers, which then might lead to compatibility issues with other processors or with standard software and programming languages. Furthermore it can lead to loss of precision in the calculations.
The author wanted to have an FPU having high performance execution, thus the author introduced architectural solutions like

- concurrent functional units (FUs) operating in parallel (almost in a superscalar

fashion, since each clock cycle any of the functional units can start a new computation, and many of them can run in parallel and write their results back to the register file at the same time, in a single-issue, multi-commit way)

- a multi-port register file (which allows the concurrent writeback)
- a special register locking mechanism and a related stall logic, which stalls the host microprocessor only when necessary

A high degree of parallelism introduces some synchronization issues due to different latencies in the execution of each floating-point instruction. These issues could be solved by the compiler, but it would become more complicated. For this reason the author preferred to solve those problems using the hardware register locking mechanism, which proved to be effective and occupied a small area.
Our design is meant to be independent from the chosen target technology and platform, and can thus map equally well on FPGA or on standard-cell technologies. Some other designs instead are instead optimized either for ASIC standard-cells or FPGA technologies [29] [30], which exhibit good implementation figures (especially a small area occupation) but can work only in a specific target FPGA and cannot be used anywhere else.
Another key feature (that the author decided to provide) to broaden the portability and the ease of use of Milk is the support of the GNU GCC compiler: C code can be compiled into assembly language which is compatible with Milk. Due to the compiler support there is no need to write assembly code manually: the users can take any C algorithm they have and get it running on our FPU. To demonstrate this, the author ran some DSP and 3D graphics algorithms written in C on two FPGA-mapped systems which include our FPU. Compiler support is rarely provided by other open-source FPUs.
The following features distinguish Milk from the FPUs on the market:

- It includes a hardware logic which handles so called denormalized operands automatically, in order to avoid a performance degradation typically referred to as denormals bug (due to trap calls related to software interrupt service routines). Our FPU has a highly parallel architecture, since each functional unit operates independently from the others. In case that two or more functional units end their elaboration simultaneously, the multi-port register file allows

a concurrent writeback of their results as long as the destination registers are different from each other. A very interesting feature which is not present in other designs, is that the author created the FUs so that the user can extract them from the design and use them as stand-alone computation engines. This gives the great benefit of providing ready-made (and already tested) components which can be used as such inside any other system (controlled by any microprocessor).

- It is highly scalable and configurable: Many key parameters of the architecture can be adjusted (width of the bus, width of the registers, polarity of the control signals, latency of the functional units, opcodes, etc.).
- Each internal functional unit performs an arithmetic operation, and can be extracted directly from the architecture to be used as a stand-alone computational element in any design. For instance, the author is planning to insert them as computing elements inside a matrix of elements for reconfigurable computation.
- It has a internal register locking logic which ensures the consistency of the execution in a way which is invisible to the user and the compiler. This logic generates the bits (lock vector) which specify which register is free to be used and which is instead about to be written by a functional unit (and is thus locked). Based on the lock vector this logic also stalls the host CPU in case it tries to access a locked register.
- It has hardware support for division and square-root (sqrt).
- It supports the GCC compiler.
- It is fully open source, scalable, and portable. It can be mapped to either ASIC standard-cells or FPGA technologies.

Milk is totally open source and not bound to any technology in particular; our VHDL model can be mapped either to ASIC standard-cells or FPGA technologies, even though (to preserve generality) the author did not make explicit usage of any FPGA-specific optimization. This choice leads to results for FPGA implementations which are less optimized than they could be, but keeps the VHDL model of general validity and portability.

3.3 MILK coprocessor: interface

The interface of our FPU (as well as its internal architecture) is as simple and straightforward as possible: it is made up of a 32-bit bidirectional data bus, an address bus, and some control signals. This choice was aimed at facilitating its usage (and possible modifications, if needed) by third-party users. It is possible to quickly interface our design to any existing core just by providing a simple wrapper to match the interface signals (whenever it is needed). Indeed, due to the simplicity of the external interface the author could attach our FPU to different cores with no need to modify our design. As a first proof the author successfully interconnected it with 2 different RISC processors: the Coffee RISC core [31], developed at Tampere University of Technology, and the Qrisc core [32], from the University of Bologna. Our FPU can be downloaded from the web page of the Coffee RISC core [33].

3.4 MILK coprocessor: architecture and design space exploration

The author made some architectural choices to provide high computational power while keeping the design also immediately understandable and easily adaptable to diverse implementations. To achieve this goal the author had to study the trade-offs between different design choices and the features that our design should support. For example it would be possible to achieve computational power and efficiency by adopting a parallel nature of the data path, and also using compact tri-state buffer arrays to allow fast switching of internal buses. The author used pipelined functional units to be able to issue a new instruction every clock cycle when needed. The author designed the pipeline stages so that the latencies are as low as possible, while at the same time the maximum clock frequency achievable is high enough to compete with other similar devices.

In addition the author tried to maximize the exploitation of parallelism allowing the simultaneous elaboration of more functional units in parallel, enabling this way optimizing compilers to schedule the instruction in an efficient way (for example issuing several simple operations after a heavy one has been issued and has still to be completed). This process may involve some kind of reordering of instructions, which then must deal with issues related to instruction dependencies. Some compilers can take care of solving all these problems; to guarantee the highest possible portability

of our design the author decided anyway to introduce a hardware mechanism for register locking and processor stalling, which detects the hazards and stalls the processor whenever it tries to access data from a register which is not ready, or tries to execute floating point instructions dependent on previous ones which are still in execution. This way the consistency of the execution is guaranteed, with no need to rely on special functionalities of the compiler; still it is true that an optimizing compiler which is aware of the hardware can schedule instructions so that the amount of conflicts (and thus the number of times the core is stalled) is minimized.
Allowing parallel execution of any combination of functional units implied some area redundancy, since a few hardware components had to be replicated inside all the functional units; on the other hand, the global computation efficiency is increased, as well as the modularity of the design: each functional unit can thus be freely removed or inserted in the data-path, without affecting the rest of the system.
The functional units get their operands from a multi-port register file, where they also store the corresponding results when their elaboration is finished. The register file is multi-ported because this way it is possible to allow an arbitrary number of functional units to write their results simultaneously, provided that they do not have also to write to the same destination register. The operands of arithmetic operations are sampled to a register inside each functional unit whenever its execution is triggered, limiting unnecessary switching activity that would lead to useless energy consumption.
The user can select whether or not to mask interrupt signals like illegal instruction or the exceptions mentioned by the IEEE Std754-1985 standard (underflow, overflow, division by zero, inexact result and invalid operation). The internal architecture of Milk is described by Figure 1, where it is depicted at logic block level. The first block is the *I/O logic*, which takes care of handling the communication with the host microprocessor. The control logic controls the execution of the instructions, the flow of the operands and of the results across the datapath, and generates the enable signals for the functional units. The register file stores operands and results of the arithmetic instructions. The status register holds the flag bits related to typical exceptions (division by zero, overflow and underflow). The status bits are related to the last instruction executed. The sticky status logic is a simple unit which freezes the status bits and keeps them available for later reading, until the content is reset by the user. The interrupt generator issues a hardware exception signal to the host CPU

Fig. 1. *Block scheme of the internal architecture of Milk FPU. © 2007 Springer [P9].*

when an exception occurs in the computation. The exception generator contains a register which is controllable by the user via software and is used to dynamically choose which exception should be masked, and which one instead should actually generate an interrupt signal to the host CPU.

The functional units are a set of components (each of them implements a given arithmetic operation) operating in parallel and independently from each other. This approach maximizes the amount of parallelism achievable in the computation of a given algorithm and gives the user the option to include or exclude each FU freely and easily from the datapath, saving area when they are not needed. The author will show later how this choice comes at the cost of an increased area of each functional unit: just 1.3% of the total area on ASIC std-cells technology, while10% when mapped on FPGA. Other designs do not allow this usually for two reasons:

- they are optimized for area occupation, thus are designed using custom layout techniques and ASIC technologies. This saves area and ensures high speed, but at the same time it is costly and the device created is monolithic and fixed.
- even when designs are made using HDL languages, designers may want to squeeze some area by sharing components between different functional units. This way, if some parts are to be detached and ported to other designs, a substantial modification should be done to the parts which are extracted, in order to obtain a component which can work correctly. At least all the parts which were shared with other components must be added back. This process can be very complicated when the design is highly embedded and is not planned for modularity.

All the functional units are pipelined, so that they can accept new operands and produce new outputs every clock cycle. This leads to a larger area occupation when compared to corresponding non-pipelined versions, but also their performance is considerably increased. This becomes especially true when considering the divider (or other functional units exhibiting long latency): if a non-pipelined version of the unit is used, a new operand must wait a long time before it can be processed. In this way, instead, a new operand is processed each clock cycle. The architecture of the divider is shown in Figure 2.

The latencies of functional units are chosen so that the clock frequency of Milk is

kept high enough to keep pace with the host processor. The pipeline stages were carefully designed to guarantee low latencies without deteriorating the maximum clock frequency achievable, as will be demonstrated in the results.
Input operands of arithmetic operations are temporarily latched in each functional unit until a new identical operation is issued: this way only the power dissipation due to leakage current will be present (besides the dynamic power consumption for performing the desired operation), since no new switching activity is present in the input ports until new operands are routed to that functional unit.
The different functional units communicate among each other and with the processor only by fetching operands and storing the corresponding results in an 8-slot, multiport register file (meaning a multiport register file with 8 registers and a dedicated writeback port associated to each functional unit instantiated in the datapath). Due to the multi-port structure of the register file, an arbitrary number of functional units can store their results simultaneously (since there are as many write ports as functional units instantiated), as long as they do not use the same destination register. The drawback is that additional area is required when compared to a simple structure having a single write port; on the other hand, the performance is increased considerably. Moreover the control and stall logic are considerably simplified, balancing this way the area increase due to the adoption of the multi-port approach, which then turns out to be a worthy choice.
The correctness of the elaboration is guaranteed by a register locking mechanism implemented in hardware. The mechanism stalls the core whenever it wants to issue an operation which requires a result of a previous instruction as an operand which has not been produced yet. The stall logic can also stall the core whenever it tries to read from Milk a result that has not been calculated yet. Due to the stall logic, the compiler is leveraged from the duty of guaranteeing correct operation in this kind of situations, leading to a simpler implementation. This logic is shown in Figure 3.
The hardware logic, used for handling the denormal operands is depicted in Figure 4. Having a hardware logic to handle denormal operands means that the user can feed to our FPU the so called denormal operands, with no need to demand their processing in a trap-based, time consuming software emulation. In almost all other devices this

Fig. 2. *Architecture of the floating-point divider. © 2008 IEEE [P16].*

Fig. 3. *Block scheme of the register locking logic. © 2008 IEEE [P16].*

feature is missing in order to save some area.
The author decided to keep it because otherwise the performance of execution is severely degraded in certain applications (especially scientific calculus). Indeed our FPU does not target a single application domain, but is meant to be an open-source item which should possibly meet the needs of anyone. Nevertheless the author made it possible for the user to exclude this logic in case he does not need it, so that some area can be saved.
An interesting point is that the impact of this logic on the overall area is less than 1% of the total area occupation using a standard-cells technology, while it ranges between 2% and 4% when mapping the design on FPGA. With such a low area occupation the author then decided to keep it (as default choice) in our design, because its cost is negligible while it adds great value to the FPU in terms of general validity and performance.
The denormal handling logic is based on a block that checks if the exponent of the input operand is null (meaning that the input operand is denormal) or not. In case the input operand is not denormal then the corresponding output is obtained by "biasing" the exponent of the input operand (by adding to it the constant 127) and the mantissa is the mantissa of the input operand (obtained by returning the hidden 1 next to the fractional part of the input operand). Otherwise the input operand is denormal: a LZC detects and counts the amount of leading zeros in the fractional part of the input operand, and based on that a shifter moves the input fractional part to the left accordingly.
The author provided the option to exclude the hardware normalization logic (by setting a dedicated VHDL generic), so that the synthesized device will require a smaller area, and all the latencies of the functional units will be automatically shortened by one clock cycle.
As a concluding note, the author reports that the user can select whether to mask (or not) some interrupt signals related to illegal instructions, or any of the five classical exceptions mentioned by the IEEE Std754-1985 standard (underflow, overflow, division by zero, inexact result and invalid operation).
An attempt has been made to optimize the VHDL code of Milk towards FPGA implementation, in order to evaluate the trade-off between possible advantages and drawbacks.

Fig. 4. *Special logic for the support of denormal operands. © 2008 IEEE [P16].*

The integer-arithmetic functions needed by the FUs of our coprocessor have been manually instantiated in the design using some dedicated library components provided directly by Altera Quartus II design software. After replacing the integer divider, the logic elements usage for Milk shrank from 18439 to 16949, and frequency from 70 MHz to 64 MHz. Replacing also the entire floating-point multiplier the previous figures become respectively 15869 and 68 MHz. However, it should be considered that part of the area reduction is due to the fact that the multiplier provided in the library does not include the hardware logic to support calculations on denormal operands. Even without these improvements it is possible to say that our implementation in its general form is already quite close to optimum, while having at the same time the big advantage of being absolutely general and portable. Because of the large capacity provided by modern FPGAs and their relatively low price, the author decided to carry on with the general version of Milk because sacrificing some additional logic elements in order to gain general validity and portability appears to be a worthy choice.

The author also developed a special version of Milk (named Cappuccino) which is meant to be tightly integrated inside the host core, in order to quantitatively explore the design space and to determine the influence of the I/O interface and stall logic. The I/O interface is straightforward, meeting the requirement for easy usage of Milk, which can indeed be easily plugged into different systems powered by different CPU cores. However, the simplicity of the interface also introduced a slight penalty to the performance of Milk by generating some overhead in the form of data transfers between Milk and the CPU. The core must first load the operand(s) to Milk, then send the desired instruction, and finally (at least in some cases) read back the final result from the register file of Milk.

To obtain the integrated version of Milk the author just had to extract the functional units and place them directly inside the datapath of the core. As a consequence the IO logic and stall logic are no longer needed, and the area of the integrated version takes approximately 10Kgates less than the external one. In terms of performance, the integrated version can save (in principle) up to three clock cycles per floating-point instruction present in the program, besides running at a higher clock frequency. Summarizing, using the integrated version some communication overhead can be avoided and some area can be saved, but the design gets more embedded in the target core. Moreover the benchmarking showed that the communication overhead introduced in the external version is not too severe. When choosing between the inte-

grated or the external version, the user should thus consider primarily the advantages in terms of area occupation (in the former case) and ease of portability to other platforms (in the latter).
The author also observed that it is possible to save roughly 2 K Gates by merging into a single, centralized block the logic which takes care of denormal and special operands, which is normally replicated inside each functional unit. On the other hand, this modification leads to some loss in modularity and flexibility: if the FUs are used as stand-alone components they are not capable of handling denormal operands or special operands, and they cannot raise exceptions autonomously. In conclusion, this kind of improvement is only valid when the user knows in advance that the trade-off is worthwhile.

To complete the framework of the design, the author developed also a tool for the debugging of code running on Qrisc and Milk. In particular, this tool is a porting of the DDD GNU debugger, modified to support Milk.

As explained before, the author explored a number of different implementations of Milk, carrying on a design space exploration.
The author provided a coprocessor version and an integrated version, to study the impact of the interface and the related communication overhead. In terms of performance, the integrated version can save (in principle) up to three clock cycles per floating-point instruction present in the program.
Summarizing, using the integrated version some communication overhead can be avoided and some area can be saved, but the design gets more embedded in the target core. Moreover the benchmarking showed that the communication overhead introduced in the external version is not too severe. When choosing between the integrated or the external version, the user should thus consider primarily the advantages in terms of area occupation (in the former case) and of ease of portability to other platforms (in the latter).
Then the author compared FUs provided with the hardware logic for the denormal numbers against corresponding lighter versions which did not feature such a hardware. As explained above, this choice saves 2K Gates, but reduces the portability.
The author studied the effects of introducing FUs which are optimized for FPGAs against generic ones. The integer-arithmetic functions needed by the FUs of our co-

processor have been manually instantiated in the design using some dedicated library components provided directly by Altera Quartus II. After replacing the integer divider, the are of Milk was 1490 logic elements smaller than the previous one, and frequency was 6 MHz lower. Replacing the entire floating-point multiplier the previous figures become respectively 2570 and 2 MHz; however it should be considered that part of the area reduction is due to the fact that the multiplier provided in the library does not include the hardware logic to support calculations on denormal operands. Even without these improvements it is possible to say that our implementation in its general form is already quite close to optimum, while having at the same time the big advantage of being absolutely general and portable. Because of the large capacity provided by modern FPGAs and their relatively low price, the author decided to carry on with the general version of Milk because sacrificing some additional logic elements in order to gain general validity and portability appears to be a worthy choice.

A number of different implementations for each of the FUs has been designed and applied, changing the number of pipeline stages, the type of integer units used inside them, their architecture, the algorithms on which they are based, and so forth. Due to these factors, it is in principle not possible to identify ”the” Milk implementation and thus ”the” synthesis results, which are indeed heavily dependent on the particular implementation chosen. Still, as a reference and for the sake of completeness, the author can mention the synthesis results for a complete, generic implementation of Milk. Milk requires (in the worst case) 105 K Gates and runs at 400 MHz on a 90nm, low-power standard cells ASIC technology, and requires 20K Logic Elements running at 67 MHz on an Altera Stratix FPGA. The latencies of functional units of Milk are reported in Table 1 and compared against the ones of other open-source FPUs. Table 2 summarizes the area occupation and maximum clock frequency achievable by the most complete version of Milk (in the worst case) and compares it with other FPUs.

One of the smallest implementations instead requires 45K Gates and runs at 600MHz on the low-power standard cells ASIC technology. The same implementation requires 10K Logic Elements running at 70 MHz on an Altera Stratix FPGA. This gives a first idea on how different results can be just depending on which functional units are actually included inside Milk. As a general rule, it emerged that the divider and the sqrt together are responsible for approximately two thirds of the total area of Milk. It is thus recommendable not to use them in area constrained designs unless they are very

frequently used operations, and the need for performance is also very high.
Also the pipelining of the functional units plays an interesting role in the operating frequency. For a non-pipelined version of Milk, the ASIC implementation runs at about 90MHz and the FPGA version runs at about 20MHz. It is thus immediately understandable how the pipelining helped in that sense. Still, even though pipelining brings good effects on performance in that it allows overlapping of several instructions and higher clock frequency, these things alone do not always imply obtaining an automatic performance gain. Moreover, very long pipelines introduce long latencies in terms of clock cycles necessary to complete each operation, leading to long and frequent stall signals sent to the host microprocessor (not to mention the related increase in terms of power consumption). Also the automatic optimization of the code performed by the compiler becomes more and more difficult as the latency increases. Finally, especially on FPGA implementation the author of this thesis noticed how the frequency increase was saturating after some point (roughly 10 pipeline stages).
Summarizing, the final choice about the pipeline length of Milk represents an optimal trade-off between performance and latency.
Also the type of integer units plays a fundamental role in the final performance and area occupation of the floating-point functional unit: indeed each floating-point functional unit contains inside the corresponding integer version. In particular, fast integer adders and multipliers are crucial since multiplications and additions are by far the most common operations in DSP algorithms, as well as in many other applications.
The choice of the author was not to force in the source code any particular architecture for the integer adder (and integer multiplier), letting the synthesis tool make the optimal choice based on the constraints set by the user. This choice guarantees easy portability and flexibility, even though it limits a bit the possibilities of pipelining. On the other hand, this is a very small price to pay when compared to the related benefits, especially considering what has been pointed out about deep pipelining. A different approach was necessary instead for what concerns the divider and the sqrt: indeed the VHDL language does not provide primitive operators for them, as it instead does for the addition and the multiplication. It was thus necessary to rely on known ASIC architectures of integer dividers and sqrt, and use them as the core for the corresponding floating-point functional units. It was found out that using a non-restoring divider is better than a restoring approach. The author suggests using a radix-2 non-restoring algorithm, which leads to smaller area and higher performance. The same applies to the square root.

Table 1. *Report of latencies (expressed in clock cycles) for the functional units of our FPU compared to other similar devices © 2008 IEEE [P16].*

Table 2. *Results of synthesis (worst case is considered) on Altera Stratix II EP2S90F1508C3 FPGA and a low-power 90nm ASIC std-cell technology of the proposed floating-point unit compared to other devices © 2008 IEEE [P16].*

There exist some schemes for dividers and sqrt functional units which rely on multiplications; such algorithms are popular in area constrained designs since they allow the reuse of the same circuit in the multiplier and in the divider. On the other hand, the author decided to adopt a replicated version because it boosts performance considerably, allowing parallel execution of different instructions at the same time. for instance, after triggering a division (which is a long operation) Milk can immediately start executing for instance a multiplication, and right after that an addition or a square root, while the division is being calculated. With a shared circuit, this would not be possible.
Similarly, a few architectures rely on serial implementations of division and square root. Once again, this guarantees a relatively small amount of area when compared to ”unfolded” implementations. Still, the author decided to support the latter ones because they allow true pipelined operations: Milk can execute any number of instructions of the same kind one after the other in close succession. Iterative implementations would instead imply ”locking” the functional unit until its previous operation has finished. Thus such a serialization degrades performance considerably.

The performance of Milk has been verified using a set of common DSP algorithms. In particular, the author reports in the following tables the empirical data related to the execution of the DSPstone benchmark and 3D graphics applications.
As it is apparent from the tables, Milk provides a significant speed-up in the execution of the code, thanks to a dramatic reduction of the number of clock cycles necessary to execute each floating-point instruction.
The computational density (discussed further in the text) is also improved, with positive impact on cache memory size, power consumption, and cost (in the case of ASIC implementation). This adds up to the usefulness of Milk.
The author benchmarked Milk with some DSP algorithms, like the DSPstone bench-

Table 3. *Comparison between the amount of clock cycles and computational densities in implementations with and without the Milk coprocessor © 2008 IEEE [P16].*

mark and a 3D graphics application written in C as well, which makes extensive usage of common algorithms like Gouraud shading, the scan line algorithm for the rendering stage, and the Z-buffer. The benchmarking results for the system equipped with Qrisc are reported in Table 3.

To evaluate the benefits coming from the usage of Milk the author measured the number of clock cycles required for each floating-point instruction, both using a software implementation and a hardware implementation (Milk coprocessor).
Table 3 shows the amount of clock cycles necessary to compute each of the algorithms in software or with the support of the Milk coprocessor. There are also shown the improvements in the code density of each algorithm using the Milk coprocessor, compared to the ones shown by a software implementation of floating-point instructions. The computational density has been calculated using the formula:

$$density = \left(\frac{f_{clk}}{cycles \cdot N.CLB} \right)$$

where f is the maximum clock frequency achieved by the circuit, cycles is the amount of clock cycles needed to perform a given algorithm, and N.CLB represents the amount of reconfigurable logic blocks (either logic slices or logic elements, depending on the FPGA family considered). The higher the density, the better the trade-off between performance and area utilized.

As a further reference, the amount of clock cycles necessary for other architectures to perform the same algorithms are listed in Tables 4 and 6.
The figures related to DSP processors are taken by the documentation of DSPstone or calculated using Code Composer Studio 3.1 (for the TMS320C6713). Since ADSP-21020 and Tartan are relatively slow, the author decided to calculate the execution time of Milk by considering its FPGA implementation, which runs at approximately 50MHz.
The author also wanted to report the results about one of the most powerful DSP processors available today: the TMS320C6713B from Texas Instruments: it is a VLIW DSP with floating-point capabilities, and is fabricated using a 0,13μm technology. It

Table 4. *Number of clock cycles needed for each algorithm by some DSP processors: TMS320C6713B, ADSP-21020, Tartan (results provided by DSPstone and calculated using Code Composer Studio 3.1 (for the TMS320C6713)) © 2008 IEEE [P16].*

Table 5. *Number of clk cycles needed for our design and TMS320C6713B to execute a 3D application. Results calculated using Code Composer Studio 3.1 © 2008 IEEE [P16].*

runs at 300MHz, and is capable of 1800 MFLOPS, 2400 MIPS, 600 MMACs. Predictably, the execution time consumed by this powerful device to execute the DSP kernels is considerably shorter than the one required by Coffee and Milk. Still, the author wanted to check what happened comparing the performance in executing a 3D application, consisting of drawing a rotating, shaded cube on a screen. In particular the author analyzed the relative performances about some critical portions of the applications, and the results are summarized in Table 5. Again, for some parts of the algorithm the performance achieved by the DSP is significantly higher than the one reported by Milk. Still, it is interesting to notice how other portions (in particular the multiplication vector-matrix and the shade-vertex, which is an implementation of the Gouraud shading algorithm) require much less time when executed by our design: this can be explained by the parallel nature of our FPU, and by the presence of a hardwired divider inside of it. The execution time of our design is assumed for an ASIC implementation running at 400MHz.
The execution time of ARM7 and ARM9 is calculated by considering the figures provided in the ARM website, which specifies that with worst case technology corner, the cores run at 300MHz and 400MHz respectively, for a speed-optimized implementation on a low-power 90nm technology. The author also considered an implementation on the worst case library corner of a low-power 90nm technology, which is characterized by a speed of 100MHz.

Table 6. *Amount of clock cycles needed for each algorithm by ARM7 and ARM9 (results determined using ARM Source-level Debugger vsn 4.60 (ARM Ltd SDT2.50)) © 2008 IEEE [P16].*

In this chapter the author presented a synthesizable VHDL description of an IP-reusable, flexible, portable floating-point unit called Milk FPU.
Milk is easily customizable and can be connected to virtually any microprocessor. It can be used as a single component, or split into its internal functional units, which can be used as stand-alone computation engines in any design. Along with its flexibility and general validity, our FPU is also able to achieve high performance, due to the low latency of its functional units and to the fact that they can operate in parallel because of the multi-port register file included in our design. As confirmation of this fact, in the execution of a series of algorithms our design reached and sometimes outperformed very powerful devices like DSP processors or optimized FPUs, while guarantees a gain equal to an order of magnitude when compared to a software implementation of the algorithms.
Milk can also autonomously handle denormalized operands, avoiding the so-called denormal bug due to a dedicated logic which can be included or removed from the instantiation. A hardware register-locking mechanism is able to stall the issue pipeline in case of data conflicts, freeing the compiler from this burden and guaranteeing computation consistency. An enhanced version of Milk has also been proposed, with increased performance and smaller area utilization, due to the fact that it is tightly integrated into the pipeline of the host processor. Milk is fully supported by a retargeted GNU GCC compiler and debugger, allowing the user to run C / C++ algorithms. Two different open-source VHDL systems, both based on a different 32-bit RISC microprocessor (respectively Qrisc and Coffee RISC) were interfaced with our floating-point unit, implemented on FPGA and then tested and benchmarked using a set of publicly available C programs and 3D graphics applications. Using Milk the amount of clock cycles necessary to execute a given algorithm is on average about ten times lower compared to a software implementation; the computational density of the code which makes use of Milk is many times higher than the software implementation, having a positive impact on the size of the program memory.

4. BUTTER RECONFIGURABLE ACCELERATOR

4.1 Different types of reconfigurable accelerators

Reconfigurable accelerators appeared quite recently on the scene of embedded systems, but opened quickly a broad set of research paths. Their main advantage consists of offering high performance by reusing the same area on silicon for different functionalities multiplexed in time. At the same time they need a small amount of configuration data (configware) when compared to FPGAs and guarantee low energy consumption.
The term reconfigurable hardware is used to describe a wide spectrum of devices. Designers interpreted the term in many different ways, producing items that are quite diverse from each other. Nevertheless, it is possible to identify a few mainstream interpretations:

- devices which are somewhat configurable at run time (for which the term ”reconfigurable but not dynamically reconfigurable” would be maybe a more precise definition).
- devices whose instruction set can be adapted depending on the target applications (as proposed by Tensilica [23]). Such devices are considered ”configurable”.
- devices whose hardware circuits can be dynamically modified and adapted at runtime to implement a set of given applications. Such devices are considered ”dynamically reconfigurable”.

The last category is the one which is closer to the actual definition of reconfigurable hardware. Still, within that category there are several other parameters which differentiate a particular device from the others:

- type of interface to the rest of the system (the reconfigurable machine can be a coprocessor, a functional unit or a standalone subsystem) [34] [35].
- bit-width of the operands they can process (leading to *fine grain* or *coarse grain* architectures).
- custom ASIC logic inside the Logic Elements (LEs).

Tensilica interpreted configurability at several abstraction levels, providing automatic tool flows able to place and connect together embedded IP blocks (available in a library) for efficiently partitioning a complex algorithm into sub modules running (possibly simultaneously) on the processing resources of the target system. This approach can be very efficient when a clear partitioning of the applications can be carried out, even though sometimes it might lead to the implementation of large and complex (thus costly) systems.
Another example is the case of the XiRISC processor, developed initially at the University of Bologna: the reconfigurable hardware is a functional unit (named PicoGA) present in the data-path of a VLIW microprocessor [51]. An advantage of this approach consists of the low latency and reduced communication overhead necessary to feed the FU with new data; on the other hand a limit must be put on the maximum length of the pipeline in order to prevent possible inefficiencies due to a high stall rate. PicoGA is a fine-grain reconfigurable machine, thus it is suitable for cryptography and telecommunication applications.
Several other devices follow instead a coprocessor-based approach; the characteristic advantages of the coprocessor-based approach are scalability and modularity, besides ease of synchronization.
The reconfigurable devices are external components somehow attached to the host microprocessor core or to the system bus. It is almost impossible to report all the different reconfigurable architectures which have been proposed so far, but the author reports here at least some of them. Well known designs are Adres [36], Molen [37], Matrix [38], RAA [39] [40], Remarc [41], Montium [42], PipeRench [43], Silicon Hive [44], Garp [45], Kress array [46] and Morphosys [47]. Most of them are already analyzed and discussed in [48]; here the author reports the key points of that survey and extends it by adding a summary of the characteristics of other reconfigurable coprocessors.
Garp [45] couples a MIPS microprocessor with a custom reconfigurable fabric. Data

and configuration are transferred between them using special instructions, used also to trigger the execution. The coprocessor can access directly the memory, which introduced problems related to memory coherency. A drawback of this architecture (which is present in almost all the coprocessor-based approaches) is the communication overhead on the system bus. This is the major bottleneck that affects the overall performance of the system. Dedicated optimizations are then needed to improve the situation, but they are not discussed here.

The reconfigurable unit in Garp is organized into an array of 24 rows of 32 computation elements. Each element contains a Look-Up Table (LUT), which is supposed to ease and speed up the elaboration of 32-bit operands.

The key innovation introduced by Garp (which has been adopted by many other reconfigurable machines) is the idea of mapping control-oriented tasks on a programmable microprocessor and heavy arithmetic computations to a reconfigurable coprocessor. Coarse grain reconfigurable coprocessors allow higher performance and denser hardware implementations, while do not support well bit-wise and control-oriented tasks.

Garp is made so that each of its rows works like a 32-bit ALU. A dedicated sequencer triggers each row so that the global execution proceeds in a pipelined way. The sequencer permits the usage of C programming language, C source code is compiled by the Garp compiler. Instead of generating assembly code, the compiler produces a Data-Flow Graph (DFG) and then maps it over the reconfigurable hardware.

Molen [37] is a reconfigurable machine implemented on a Xilinx Virtex II FPGA, attached to a PowerPC microprocessor. The tasks to be mapped on this machine are considered primitive operations which are microcoded in the architecture. Molen can process different kinds of instructions: normal ones (executed by the PowerPC) and others which are used to control the reconfigurable hardware.

Molen has a polymorphic instruction set, the data movements (between the microprocessor and the reconfigurable fabric) are explicitly declared. Computation and reconfiguration are performed using special assembly instructions.

PipeRench [43] is divided into rows; each row acts as a pipeline stage. The processing elements (PE) contain a lookup table (LUT) and a dedicated carry-chain adder circuit for fast addition performance.

The configuration of the array is propagated across the array: the configuration of each row is loaded at every cycle.

PipeRench is programmed using a special language which resembles C: C operators

are mapped directly to the resources present in the PEs.
Pact XPP [50] is composed of a set of processing elements named Processing Array Clusters (PACs). Each PAC contains an array of Processing Array Elements (PAEs) and a Configuration Manager (CM), which is meant to control the configuration of the device. Each PAE is characterized by a 16-bit datapath, a synchronization register and some functional units. PAEs are connected via a special interconnection network. Data are stored into packets and transmitted across the network.
Each PAE operates approximately like a functional node in a dataflow model: execution starts as soon as the operands are available at the input ports, and the results are put as soon as possible on the output port, which feeds another node. XPP is programmed using a language (NML) created internally, which is a structural event-based netlist description language.
XPP, as well as other reconfigurable machines, adopts configuration sequencing instead of instruction sequencing (like in programmable microprocessors): they alternate different configuration, each implementing a given functionality. Data are processed in streams instead of single words. Still, not all the applications are suitable for such a solution, and using a standard microprocessor is still more flexible and convenient from the perspective of application developers.
Morphosys [47] is made up using a 32-bit RISC microprocessor (TinyRisc) and an 8x8 Reconfigurable Cell Array of Reconfigurable Cells (RCs). Each cell has a 16-bit datapath. Inside each cell there are functional units like a multiplier, an ALU, a shifter, a small register file and an input multiplexer.
Morphosys is provided with a special configuration memory which is able to overlap configuration with computation, meaning that part of the configware memory can be rewritten while the machine is executing. Morphosys is made so that its rows are able to perform SIMD-like operations. Considering all these characteristics it is then obvious that Morphosys is conceived for applications characterized by high regularity, high data parallelism and high throughput (like video compression and image processing).
Remarc [41] is similar to Morphosys, at least from the point of view of the architecture.
Pact XPP [50], PipeRench [43] and Adres [36] have the most coarse-grain datapath.
XPP and PipeRench are stream-oriented: data are provided from the external world by special peripherals, or fed by a RISC microprocessor or a DMA controller.
In addition, Adres exploits a dedicated frame buffer to overlap data computation and

Table 7. *Short summary of coarse grain reconfigurable machines. © 2007 Springer [P9].*

transfers. Adres is composed of a VLIW microprocessor and a Reconfigurable Functional Unit (RFU), which is supported by a compilation environment named DRESC, which greatly increases the value of the architecture. Moreover Adres has a very flexible way of accessing data, which allows to support not only data streaming applications.
Still, the register file-based approach is a huge bottleneck that hampers the performance of this architecture.
A different solution to this problem is proposed in Montium [42]. Montium is a coarse-grain reconfigurable processor composed of Tile Processors (TPs). A TP contains five 16-bit ALUs controlled by a sequencer, and contains 10 1-KB RAM buffers driven by a configurable Address Generator Unit (AGU).
Table 7 summarizes the reconfigurable architectures mentioned; it is clearly visible how most of them are used as coprocessors for multimedia applications in general.

Almost all of the architectures mentioned are attached to the system bus, and are coarse grain machines, thus suitable for imaging or streaming applications, characterized by a large number of data-parallel operations. Indeed, for applications like DSP algorithms, multimedia processing and 3D graphics it has been shown how coarse-grain reconfigurable architectures allow a significant advantage in terms of ease of mapping of basic functionalities, latency, and energy consumption [52] [53] [54].
A number of intermediate solutions between opposite approaches are possible: ranging from fine-grain to coarse-grain cells and from external devices to functional units present on internal data-path of microprocessors.

4.2 Butter accelerator

Butter is a coarse grain reconfigurable coprocessor for a RISC microprocessor. It is organized as a matrix of 32-bit elements, and aims at maximizing the performance in the elaboration of multimedia, signal processing, and 3D applications.
Butter is entirely described by a parametric VHDL model, which ensures portability to any platform (either ASIC or FPGA). The mapping of VHDL on standard-cell technologies implies using more area on chip and getting lower clock frequencies

Fig. 5. *Butter inside a system.*

Fig. 6. *Butter and its internal memories ©IEEE, 2008, [P12].*

when compared to custom layout design, but it also allows obtaining higher portability, not only because the same VHDL code can be implemented on FPGAs, but also because it is not necessary to repeat all the design procedure whenever it is necessary to migrate to a more advanced microelectronics technology.

To speed up the execution of applications one possibility consists of detecting the parts which are more computationally heavy (called kernels), and map them on specialized hardware (which allows a highly parallel implementation). The author considered some applications to be implemented on Butter, manually located the kernels and mapped them on our coprocessor, since at the moment it is not available any automatic tool for that purpose.

The inclusion of Butter into a generic system is shown in Figure 5. As can be seen in the picture, Butter is a coprocessor attached to the system bus. This might put some constraints to the kind of applications which can benefit from mapping on our architecture, since the bandwidth between the microprocessor and the coprocessor is limited. For this reason the author decided to provide the possibility of using our machine also as a kind of I/O processor: once the array has been configured it is ready to process incoming data, which can be fed for example from a high-speed input peripheral like a camera. The results are produced as output through another high-speed port which (for instance) could drive directly a display, since Butter is mainly targeted to applications like image and video processing.

Some dedicated configuration bits are used to specify the elaboration that the cells of Butter must perform and from where their operands come (thus defining the topology of interconnections between cells). Configuration bits are stored in a dedicated memory inside Butter, and can be written by the core or via Direct Memory Access (DMA) transfers.

Butter is organized as a matrix of processing elements called cells. The number of rows and columns of the matrix is determined by constants included in the VHDL model of Butter (generics).

Besides the actual array, Butter is made up of configuration and data memories, plus some control logic. A picture of the global architecture is shown in Figure 6.

Each data memory is a dual-port RAM. One port is characterized by 32-bit data and is connected to the system data bus, in order to be directly accessed by the general

purpose core.
The second port is connected to the array through a set of configurable switches. The purpose of these switches is to modify the order in which the operands are sent from the memory banks to the array, since different algorithms might require reordering the data in a special way.
The local memory subsystem of Butter is driven by a direction bit defined in a local control register. This direction bit controls the data flow across the accelerator. The two local memories are indeed accessed as a single entity from the system bus. According to the value of the direction bit, one of the memories is defined as *input memory*, while the other one acts as *output memory*. Butter processes the data contained in the *input memory* and puts the results in the *output memory*. If the value of the direction bit is toggled the role of the two memories is exchanged; this way the data can be processed locally several times without requiring any data traffic on the system bus. This is useful especially if the data processing is implemented using several configuration contexts. The data can be elaborated once using the first context, then the results can be processed again using another context, and so on, just swapping the data flow direction at each step.
The size of these data memories can be set at design time. At each step the number of items to be processed by the array is configurable and it is stored in a local register that can be accessed via software. This register is used by some control logic that manages also the synchronization issues related to the data flow.

4.2.1 Dual-level reconfiguration

The architecture of Butter can be programmed at two levels: design-time and run-time. Design-time configuration affects HDL parameters of the template, such as the amount of cells instantiated, the density and the kind of the interconnections and the set of operations that each cell is capable to provide. It is maintained fixed on the FPGA for long time intervals, and may be changed only when a significant update of the applications mapped is necessary.
Run-time configuration represents an assembly-level definition of the operation performed by each cell at a given cycle, or the routing of data between different cells. It is used to implement a given computation, and it is utilized to exploit time multiplexing of the FPGA resources and ensure software-like flexibility. The set of bits describing the run-time configuration of the Butter array is termed ”configuration context”, and

is composed of 14 bits per each cell. A configuration cache is implemented on RAM blocks placed locally to the cell array, in order to provide "multi-context" capabilities: up to four different configuration contexts (only one of which is active at a time) can be programmed to allow fast context switching. Indeed, switching to a different context already present in the configuration memory takes only one clock cycle. The loading of a new configuration context from the main memory to the configuration cache takes up to 60 clock cycles, but that delay is anyway hidden most of the times, because context load can be performed while the array is running another one. If the algorithms to be mapped fit in a sub-portion of the array, then it is possible to map them simultaneously, so that multiple different configurations can be active at the same time in the array.

4.2.2 Internal architecture of Butter: cells and interconnections

Each cell inside Butter has two inputs ports to read 32-bit wide operands; a 6-bit wide input port brings inside the cell the configuration bits, used to specify the kind of operation that the cell should perform on the operands. Also reset and an enable input are used to control the internal registers of the cell.
There are two 32-bit output ports for each cell: they can be used either to bring out the 64-bit result of a 32-bit multiplication, or a generic 32-bit result coming from another functional unit, together with one of the two input operands, which is selected and fed through the cell (used to simplify the routing). Input registers inside the cells are used to sample the operands, creating this way a sort of pipeline; these registers can be also disabled to avoid useless dynamic power consumption. One special input register is used for keeping constant values inside the cell, so that they can be used during the elaboration with no need to re-route them.
Inside each cell there are three functional units (a multiplier, an adder and a barrel shifter) and a small memory (4 cells 32-bit wide) used as a lookup-table (LUT) to implement simple logical functions: this way the author tried to bridge between the coarse grain nature of Butter and some functionality which could be implemented by using fine grain reconfigurable hardware.
A special functional unit was inserted in the cells to permit the implementation of floating-point multiplications. This choice is justified by the fact that 3D graphics is nowadays very popular, and makes extensive use of floating-point arithmetic, so the author adapted the structure of the cell and of the interconnections to provide a first

Fig. 7. *Architecture of a Butter cell. © 2006 [P6].*

form of support for floating-point multiplication, because together with addition it is by far one of the most frequently used instructions.
3D graphics benefit from fast, low precision floating-point operations. For this reason the author implemented a version of the floating-point arithmetic operations which does not handle the denormal numbers (it would require additional circuits and complicate the mapping procedure). The author also decided to get the whole instruction processed internally by each cell, so that an operation requires just one clock cycle. For example, considering the architecture of a simple floating-point multiplier it is apparent how the integer multiplier and the integer adder (which are necessary to build a floating-point multiplier) come at no cost, since they are present anyway inside the cells.
The packing logic that prepares the results produced by the adder and the multiplier, rounding them to be stored in the floating-point format, is too complex to be fit inside the LUT alone, and cannot be mapped on the remaining functional units, so the author had to introduce a dedicated block inside the cells. It consists of a portion which calculates the amount of leading zeros for each of the operands, the sign of the result, and packs the internal number into the final format. The structure of each Butter cell is described in Figure 7. The first row of cells read their operands from global vertical interconnections. The results of the elaboration are put as output accessible from the underlying rows. The final result can be read externally of Butter either from its last row at the bottom of the device, or from the rightmost column. This way results can be accessed as soon as they are produced, with no need to wait that they go through all the rows.
Both the input ports of cells inside the array can be connected to the outputs of the cell straight above them, or instead to the cell which lays two rows above (interleaved connection). The interleaved interconnection is useful (for example) to propagate the 64-bit result of multiplications splitting their processing over two adjacent rows, for instance like happens in the FIR algorithm: the input operands have to be multiplied by different coefficients, and after that the results must be added together. Thanks to the interleaved connections it is possible to implement the FIR algorithm using only three rows of the array: the first row executes the multiplications, the second row the additions of the least significant bits of the products, and the third row the addition of the most significant bits.

Fig. 8. *Interconnections inside Butter.*

Both the input ports of cells inside the array can be connected to the outputs of the cell straight above them, or instead to the cell which lays two rows above (interleaved connection). The interleaved interconnection is useful (for example) to propagate the 64-bit result of multiplications splitting their processing over two adjacent rows, for instance like happens in the FIR algorithm: the input operands have to be multiplied by different coefficients, and after that the results must be added together. Thanks to the interleaved connections it is possible to implement the FIR algorithm using only three rows of the array: the first row executes the multiplications, the second row the additions of the least significant bits of the products, and the third row the addition of the most significant bits.
Another possibility provided by the interconnection network consists of enabling the input ports of a cell to receive the results of the cell on its left side, as well as the ones lying in diagonal direction (upper left and upper right). They are useful in facilitating and enhancing the mapping of some algorithms, and in reducing the amount of cells used, e.g., to elaborate some additions in parallel, using diagonal interconnections it is possible to use seven cells instead of eleven (which would be necessary using only vertical and horizontal connections).
Also two different global interconnections are provided, crossing the array in vertical and horizontal directions, connecting the output of each cell to every input of the cells lying on the row below. Global interconnections are handy for mapping algorithms like matrix-matrix multiplications and matrix-vector multiplications (largely used in 3D graphics, especially in the vertex transformation stage of the graphics pipeline).
The interconnections allow bringing the outputs of each cell also to the configuration memory of the cells in the row below, configuring them dynamically depending on the results of the previous instructions. Nearest-neighbor interconnections are anyway sufficient to implement the simplest DSP algorithms, while the global ones are more useful for matrix multiplications and 3D graphics algorithms. The global interconnections can also be configured to be regional interconnections, by specifying the number of rows and columns which define a region. This kind of interconnection is useful to implement the multiplications of a set of adaptive coefficients (like happens in filters) that must be all multiplied by a common value. A summary of the local interconnections between two Butter cells is shown in Figure 8

Fig. 9. *A screenshot from the GUI-based programming tool.*

4.2.3 Programming Butter: a GUI-based tool

It is easy to understand that writing manually the configuration bits for the cells and the interconnections of Butter would be a very impractical and time consuming task. As a solution to this inconvenience the author presents a graphical user interface (GUI) that allows the user to easily select the desired operation and interconnection choosing it from a series of scroll-down menus.
The function of the GUI is to create automatically the configuration string of each Butter cell by generating some output files (C header files), which can be used to program Butter. Fig. 9 depicts a screenshot from the GUI.

4.2.4 Butter as a mean for fast and easy FPGA implementation of applications

Over the last years FPGAs have been successfully established as a very appealing technology option for digital signal processing, in alternative to the utilization of programmable DSP or devices based on ASICs. FPGAs have developed to the point that it is possible to map an entire system-on-chip over a field-programmable device. Many commercial applications, especially for low or medium volumes, use extensively FPGAs in fields ranging from consumer multimedia to automation control, to the very recent approaches to the automotive environment. [55]. In order to meet the demands of this increasing market segment, FPGA vendors are enhancing their high-end products using embedded RAM blocks and specific computation engines for signal processing, increasing both their computational density and their flexibility, ultimately enlarging yet again the possible fields of utilization of this kind of technology.

In the context of this significant success, FPGA utilization in signal processing still presents a few drawbacks:

a) FPGAs are programmed using VHDL/VERILOG, which are bit-oriented, event-driven hardware description languages. HDLs are not suited for signal processing: on one hand, algorithm and application developers have today a very

software-oriented background. On the other hand, even though the DSP community is actually strongly investing on the formation of new application engineers with mixed HW/SW curricula, HDL inherently do not offer the design productivity needed by today's algorithmic exploration and deployment, due to their complexity and the high number of design parameters.

b) FPGAs are very costly in terms of area occupation and above all in terms of power consumption, besides having long reconfiguration time. As suggested in [35] it is not possible to utilize them as "soft-ASICs" simply porting ASIC design styles over a different technology support, mapping on HDL all necessary computation and exploiting the maximum available parallelism. The high cost of reconfigurable logic can be justified only providing very high utilization of the available resources, thus multiplexing over time different computations on the same logic.

Several potential solutions have been presented to tackle the problems described above: point (a) can be challenged by C-to-HDL automated flows. In this case, though, the results of the translations are likely to be less optimized than manually encoded HDL, thus this option is affected by the problem described in (b).
A second interesting option consists in exploiting dynamic reconfiguration [59], that is partitioning FPGAs into a fixed "static" region for control and synchronization and multiplexing different stages of the required computation, often defined as "slots" on a "dynamic" region. For instance, the Xilinx Spartan 3E features an internal port for self-reconfiguration that allows a static region to redefine at run-time the configuration of the rest of the FPGA [56]. At the moment, though, the reconfiguration update is done at very low bit-rates (roughly 66 KBytes/s) which is hardly compatible with real-time constraints posed by most application environments. Further deployments of this appealing strategy would probably require strong efforts from a technology and toolset point of view.
A possible alternative is the idea of mapping "virtual" logic architectures over the underlying FPGA fabric. These architectures can be more suitable to software-oriented programming environments and allow faster and easier self-reconfiguration. FPGA vendors provide "soft" microprocessor cores such as NIOS [57] and Microblaze [58], that are synthesizable over the FPGA fabric. In other contexts, several IPs designed for ASIC environments have been ported to FPGAs to raise the design abstraction level mapping more flexible computers over the reliable and solid FPGA support.

The MOLEN [37] architecture is a very successful example of powerful and general purpose signal processor template entirely built on a commercial FPGA. A very recent proposal is to port the concept of Coarse Grain Reconfigurable Architecture exploited in QUKU [60]: the authors define a CGRA template to be implemented on a Xilinx technology. This way, the device can take advantage of the programmability and flexibility of CGRAs, and exploit their fast and efficient dynamic reconfiguration means. On the other hand, it is also possible to adapt the structure of the CGRA's interconnect and processing elements (eventually mapped on the FPGA fabric) to match several different application fields across a longer time interval. This way it is possible to overcome the two most significant drawbacks of CGRAs implemented on silicon: the fact that CGRAs tend to be quite domain-oriented, and the fact that they are relatively difficult and expensive to implement.

In this thesis, the author proposes Butter as a CGRA template suitable for implementation on a FPGA support. With respect to the solution proposed in [60], Butter presents a significantly coarser grain (32-bits datapath), and supports Floating-Point computation. To mitigate the costs due to such a coarse grain Butter also offers the support for SIMD sub-word computation inside each cell.
The author proposes using a special configuration memory to store multiple configurations for the array. This way it is possible to switch from one to the other in one clock cycle, allowing efficient "task switching" and hiding reconfiguration overhead.
A very special feature of the proposed CGRA template consists of providing a practical mean of programming FPGA devices in an effective and easy way on high abstraction level. Butter can indeed be thought of as a programming layer for FPGAs. The user can implement a given application on FPGA remaining at a relative high level, just using a special graphical programming interface (previously described) which allows direct implementation on our CGRA with no need to write the corresponding VHDL or C code in order to get the application running on FPGA.
The process of implementing an entire application on Butter is still manual to some extent, in particular at the beginning of the implementation. Indeed the user has the responsibility to study the characteristics of the applications he wants to map, and decide if they are suitable or not for implementation on Butter. For instance, applications which operate on bit-wise data are clearly not the best ones, since they would

lead to underutilization of the hardware resources instantiated. Another point is to check that the application exposes enough data-level parallelism so that the architecture of Butter can be exploited. Once the designer founds a suitable application, it is necessary to perform a manual partitioning of the application into chunks that can fit into a configuration of Butter; if the entire application does not fit entirely into a single context of Butter, then it is necessary to use multiple configurations, and be sure that the control program running on the host microprocessor controls properly the loading of the correct configuration every time, as well as the data transfers. As mentioned, these stages are still manual; their automation and potential integration into the compilation flow are left as future work.
On the other hand, already at this stage a great advantage emerges: the user can implement applications on Butter just by entering a DFG-like representation of the algorithms using only the GUI tool; no knowledge about VHDL coding and FPGA implementation flows is required. This also opens up the doors for an easy splitting of the tasks between different engineers in design teams: application developers can create the Butter implementation of their applications just using the graphical tool. The tool automatically generates the configware for Butter and stores it into a normal C header file, which can then be delivered to the hardware engineers for ready plug-and-play usage in the hardware platform.

4.3 Testing and benchmarking

In this section the author will describe the mapping on Butter of a few of the applications considered. The author will present here the results related to the mapping on Butter of a 2D low-pass filter, of a median filter, of a noise reduction filter and of a deblocking filter used in H.264 decoding algorithm. They all represent very common and important applications in image processing domain; the author will try to show how efficiently Butter can execute such algorithms, especially when taking advantage of subword computation. The 2D FIR filter can be described by the following equation:

$$y_{m,n} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} c_{i,j} x_{m+i-(k-1)/2, n+j-(k-1)/2} \quad (1)$$

where $x_{m,n}$ is a pixel belonging to the image to be filtered, $y_{m,n}$ is the corresponding pixel in the filtered image and k is the size of the chosen window. Depending on the window-size of the filter, a certain number of image pixels must be multiplied by the corresponding filter coefficients simultaneously, and the weighted sum of all product terms must be calculated, to generate one pixel of the filtered image. The window size has to be an odd number, in order to have the original pixel centered in the window. Standard values used for the window size are 3, 5 or 7; in our case a 3x3 window has been used in order to reduce the complexity of the mapping.

Since the array is pipelined, each column calculates a new value per clock cycle. Taking into account the costs for data transfers and configuration, Butter takes 27,136 cycles to perform a low-pass filter. The same algorithm run on a 32-bit RISC microprocessor (Coffee RISC core [31]) running at the same frequency as Butter takes 24,960,000 cycles, so using Butter the speed-up in this case is up to 920. When subword capability is exploited then the number of cycles needed by Butter is halved, so the speed up in that case is 1840.
This example demonstrates how effective the usage of subword is operations in Butter: With a slight increase in area, the benefit is significant, doubling the performance and exploiting fully the 32-bit datapath (which would be otherwise only half-utilized). To further benchmark our machine the author then made the same comparison for other filters: a median filter, of a noise reduction filter and of a deblocking filter.
A similar analysis has been done also for other algorithms: a median filter, a noise-reduction filter, and the kernel of the deblocking filter for a H.264 decoder. The related profiling results are reported in table 8. Median filter is a sub-portion of the noise reduction filter; this benchmark is interesting because it exercises the reconfiguration hiding capabilities of Butter: while Butter is making calculations, the processor can load a new configuration context inside Butter (at run time).

Table 8. *Clock cycles needed to perform different filters on a 320x240 image using Butter or a software implementation (Coffee RISC microprocessor). © 2008 Springer [P15].*

4.4 Implementation and synthesis results

Table 9 reports the results of the logic synthesis of the VHDL code of an 8x4 array of Butter with subword capabilities (both on Altera Stratix II FPGA and on a 90 nm ASIC standard-cells technologies).
The author reported the results obtained requesting to the synthesis tools (Altera Quartus II and Synplicity Synplify) an area-optimized implementation and a speed-optimized one. Speed constraints were set to 120MHz on FPGA and 1GHz on ASIC.

Table 9. *Results of the synthesis of Butter on FPGA and ASIC standard-cells technology © 2008 Springer [P15].*

Synthesis results of a stand-alone instantiation of Butter (8x4 array with subword capabilities), together with figures related to other coarse-grain machines are reported in table 10. Since the other devices listed are implemented (at least partly) in fully ASIC technologies, an implementation on FPGA is not possible for them, so also for Butter the author listed only the figures related to the implementation on ASIC standard-cells.

Table 10. *Area and speed figures for Butter and other standard reconfigurable machines. © 2008 Springer [P15].*

As it can be noticed, table 10 contains data which are not homogeneous, due to different target technologies, array size, and implementation methodologies. Moreover, the architectures listed have different datapath width, and the figures related to Butter are pure synthesis results, which can differ from actual results obtained after a real physical implementation on silicon. Conversely, Butter is described by a generic, technology independent VHDL model mapped on a 90nm standard-cells technology, while the other architectures listed are obtained using fully custom design techniques (except for ADRES). For this reason the area occupation is significantly higher when compared to the one obtainable using a fully-custom ASIC technology and design.
the author anyway tried to obtain more figures for Butter, obtained applying the scaling rules to area and frequency. The aim of the scaling consists in having a set of values which are supposed to be an approximation of the results of a synthesis of

Butter on other technologies. the author calculated that on a 0.13 μm technology an 8x4 Butter array would occupy roughly 4,487,698 μm^2 running at approximately 214 MHz. On 0.35 μm technology, instead, an 8x4 Butter array would occupy roughly 35,901,584 μm^2 running at approximately 110 MHz.

The 8x8 Morphosys array from the M1 chip [61] [62] is implemented in 0.35 μm technology and occupies 55,000,000 μm^2; an 8x8 array of Butter would occupy roughly 71,803,168μm^2, which is 1.3 times the area of the Morphosys array. On the other hand, Morphosys has a 16-bit datapath, uses 16x12 bit multipliers instead of 32x32 bit ones like Butter, and makes use of fully custom technology. Even assuming the usage of 16x16 bit multipliers (besides the other functional units) the area ratio with a 32x32 bit version of a cell of Morphosys would be more than 2. Taking into account all these points, the author can conclude that Butter should present a slightly smaller area occupation than Morphosys. The clock frequency should be instead approximately the same.

Similar considerations and conclusions can be done about a 0.13 μm implementation of Butter when compared to the M2 version of Morphosys [63], even though in this case the area occupation of the M2 chip is much higher as well as the working frequency. The main reasons for this seem to be the fact that the RC inside the M2 array is more complex than in the M1 version, and they are pipelined.

About ADRES [64] the author found the data concerning a single RC cell; the author thus multiplied the related value by 32, obtaining this way an estimation of the area occupied by a 8x4 ADRES array: the value is 6,272,000, while Butter would occupy 4,487,698 μm^2. On the other hand, the author also calculated the value of a single cell of Butter in 0.13 μm technology, which would be approximately 178,030 compared to the 196,000 μm^2 of the ADRES cell. Also the working frequency of Butter would be higher.

It is instead extremely difficult to make an analysis of the comparison with Montium [42], [65], due to the fact that Montium has a very different architecture and organization: it is made up of five so-called tiles, which include ALUs and register files. It is thus very difficult to state to which array size of Butter it would correspond. Based only on the actual figures reported in table 10 the author can see that Montium has an area occupation which is half the area of Butter, and runs at a frequency which is also half the one of Butter. It must be remembered also that Montium has a 16-bit datapath.

The author would like to stress once again the fact that the figures reported about

Butter in different technologies are obtained by calculations (scaling rules) from the results of pure logic synthesis, and not from an actual chip implementation on silicon, which might lead to different results. For this reason the comparisons made are meant to be just a way of understanding if the Butter implementation delivers values which are reasonable or not, taking into account the state of the art.
As a further confirmation, a 64-point FFT algorithm was mapped on Butter, then the benchmarking results were compared to the results of the mapping on Morphosys: Butter takes 100 cycles (without exploiting the subword capabilities), while Morphosys takes 111. This result shows how Butter is effective not only when compared to a software implementation, but also compared to other similar architectures.

In this chapter the author described the design and implementation of Butter, a reconfigurable machine with subword and floating-point computation support. As an example of the effectiveness of this solution especially in the field of image processing the author described the mapping of a 2D low-pass image filter, of a median filter, of a noise reduction filter and of a deblocking filter used in H264 decoding algorithm. Their mapping and execution has been tested on a prototype of the device (8x4 matrix with a 32-bit datapath) implemented on a FPGA board. The author calculated the clock cycles needed to perform the algorithms mentioned above on a RISC microprocessor and on Butter: using butter, the author obtained speed-up factors up to two orders of magnitude. the author also compared area and frequency figures of Butter with other reconfigurable architectures: results show how Butter is likely to have an area occupation which is roughly the same or even smaller than other similar devices, while the working frequency could also be slightly higher. On the top of this, the main characteristic of Butter consists in the fact that it tries to gather the best points from different architectures, while at the same time introducing brand new solutions in the field of CGRAs. At the same time, Butter it is also totally scalable, portable and technology independent.

5. SUMMARY OF PUBLICATIONS

The main contributions of the publications included in part II are summarized here. Section 5.1 describes how the author contributed to the published work.

Publication [P1]: *A system level IP integration methodology for fast SoC design*.
This paper proposes a novel methodology for the fast development of scalable SoC architectures based on pre-made IP blocks connected via an AMBA bus.

Publication [P2]: *A Reconfigurable FPU as IP component for SoCs*.
This paper introduces the first release of Milk, an open source single-precision floating-point unit designed to be a convenient external coprocessor for RISC microprocessors. Milk is a parametric 32-bit Floating-Point Unit described using VHDL language constructs that enable hardware configurability.

Publication [P3]: *A Flexible Multiplier for Media Processing*.
2D graphics and signal processing applications in general benefit from the usage of integer Single-Instruction-Multiple-Data (SIMD) functional units, while 3D graphics applications can be significantly accelerated employing single precision floating-point functional units. This paper presents a model and implementation of a versatile multiplier able to perform either double precision, (paired) single precision floating-point multiplications or 16-bit or 8-bit SIMD integer (vector) multiplications; it was implemented on an FPGA device and compared to other floating-point multipliers and similar devices, each capable of performing only a limited subset of the proposed design.

Publication [P4]: *A FPGA Implementation of An Open-Source Floating-Point Computation System*.
This paper presents the implementation on a FPGA board of an open source, technology independent, VHDL model of a floating-point computation environment for SoCs, composed of a RISC microprocessor system closely coupled to a floating-point unit. The FPU features a full hardware handling of normalization and denormaliza-

tion hazards. A benchmark suite of DSP algorithms written in C language has been run on the proposed platform.

Publication [P5]: *A VHDL Model and Implementation of a Coarse-Grain Reconfigurable Coprocessor for a RISC Core*.
This paper presents a coarse-grain reconfigurable machine used as a coprocessor to speed up the execution of computationally demanding tasks, which would be too heavy for a generic RISC core stand alone. A VHDL model of the proposed architecture has been created for simulation and implementation. Some common algorithms for signal processing and multimedia applications have been mapped over our design, to benchmark it and compare the results against another existing architecture.

Publication [P6]: *A Coarse-Grain Reconfigurable Machine With Floating-Point Arithmetic Capabilities*.
This paper presents the implementation of a reconfigurable architecture with floating-point arithmetic capabilities. In particular, we decided to provide the machine with the capability to perform floating-point multiplications, because of their wide usage in 3D graphics applications, which are nowadays very popular. The results show that the area and clock speed of our design are reasonable and are not severely affected by the insertion of the floating-point support; at the same time our architecture provides a significant speedup in executing a series of common DSP algorithms.

Publication [P7]: *Design and Verification of a VHDL Model of a Floating-Point Unit for a RISC Microprocessor*.
In this paper is described how we verified the design of a synthesizable VHDL model of a parametric 32-bit Floating-Point Unit, using many different techniques that combined together ensure a dramatic reduction of testing time and high test coverage.

Publication [P8]: *Implementation of a Tracking Channel of a GPS receiver on a Reconfigurable Machine*.
In this paper the author proposed a dedicated version of Butter featuring special functions aiming at providing an efficient support for the implementation of a GPS receiver. The cell of Butter has been enhanced to better support the fine-grain nature of the application.

Publication [P9]: *Co-processor Approach to Accelerating Multimedia Applications*.
This book chapter summarizes the results achieved with Milk FPU and Butter.

Publication [P10]: *Programming Tools for Reconfigurable Processors*.

In this book chapter some dedicated tools used for the programming of reconfigurable architectures are overviewed.

Publication [P11]: *Implementation of a 2D Low-Pass Image Filtering algorithm on a reconfigurable device*.
This paper discusses the implementation of an image filter on Butter.

Publication [P12]: *Implementation of a floating-point matrix-vector multiplication on a reconfigurable architecture*.
This paper discusses the implementation of a matrix-vector multiplication on Butter.

Publication [P13]: *A Dedicated DMA Logic Addressing a Time Multiplexed Memory to Reduce the Effects of the System Bus Bottleneck*.
This paper introduces the introduction of DMA capabilities into Butter, showing the related added cost in terms of extra area and the related advantages in terms of performance. In particular, this paper discusses a dedicated solution aimed at reducing the delay related to the data transfer overhead over the system bus. This solution is aimed in particular at FPGA implementations.

Publication [P14]: *Reconfigurable Hardware: the Holy Grail of Matching Performance with Programming Productivity*.
This paper presents a holistic survey over different kinds of reconfigurable architectures introduced during the last years. It also discusses the related design choices, highligting advantages and disadvantages.

Publication [P15]: *A Coarse-Grain Reconfigurable Architecture for Multimedia Applications Featuring Subword Computation Capabilities*.
This paper presents the design and the implementation of a coarse-grain reconfigurable machine used as an accelerator for a programmable RISC core, to speed up the execution of computationally demanding tasks like multimedia applications. We created a VHDL model of the proposed architecture and implemented it on a FPGA board for prototyping purposes; then we mapped on our architecture some DSP and image processing algorithms as a benchmark. In particular, we provided the proposed architecture with subword computation capabilities, which turns out to be extremely effective especially when dealing with image processing algorithms, achieving significant benefits in terms of speed and efficiency in resource usage.

Publication [P16]: *Design space exploration of an open-source, IP-reusable, scalable floating-point engine for embedded applications*.

This paper summarizes all the work done in [P1], [P2], [P4], and extends it by presenting results related to benchmarking of complex applications, compared against the ones achieved by a broad set of other FPUs and DSP processors. A new, integrated version of Milk is also proposed, illustrating the related trade-off in terms of reduced area and communication latency when compared to the coprocessor version.

5.1 Author's Contribution to Published Work

Here the author's role is clarified on the work done for each publication. The thesis author was the primary author in nine publications (plus two more, not listed here for lack of time) and coordinated the work done for other publications. In addition, there are two publications listed here in which the thesis author's does not appear as the first author, but provided significant contribution. Prof. Jari Nurmi supervised the work and gave valuable comments on all of the published papers.

Publication [P1]. The purpose of this paper was to demonstrate the effectiveness of an IP-based design approach: a set of IP designs (designed and verified independently by different research groups) was provided, and integrated into a library used by a tool able to build an entire SoC based on area and performance constraints set by the user. The author of this thesis provided a scalable and parameterized IP design to M.Bocchi (the first author of the paper), who was able to use it successfully in his research work, integrating the IP in the mentioned library and generating working SoCs based on that. The author of this thesis also contributed by writing a part of the publication.

Publication [P2]. This publication was entirely planned by the author. The purpose of this paper was to demonstrate that it is possible to design a synthesizable VHDL model of an FPU fully compliant with the IEEE-754 standard characterized by reasonable figures in terms of area occupation and performance. A second target was to show how the design proposed is characterized by portability over different technology with no need to modify the VHDL code; moreover, the same design was successfully integrated into two different SoCs powered by different RISC microprocessors, demonstrating how the design can be easily reused in different projects.
The author carried out all the work described in the publication related to the FPU. F. Campi and J. Kylliäinen provided the two microprocessors to which the FPU was attached.

The author entirely wrote the publication, while the other authors gave valuable comments.

Publication [P3]. This publication was entirely planned by the author. The purpose of this paper was to demonstrate how it is possible to reuse heavily the hardware resources present in a floating-point multiplier, sharing them in order to implement different arithmetic operations in different times, at a low area overhead. The author aimed at demonstrating how it is possible to mitigate the impact of the large area occupation of a double-precision floating-point multiplier by enabling the same hardware to compute paired single-precision floating-point operations, and integer sub-word multiplications and multiply-accumulate operations.
The author carried out all the work described in the publication.
The author entirely wrote the publication, while the other authors gave valuable comments.

Publication [P4]. This publication was planned by the author together with C. Mucci and F. Campi, and is the natural evolution of *Publication* [P2].
The purpose of this paper was to demonstrate how the insertion of a FPU impacts the performance of a SoC powered by a programmable RISC core and some I/O peripherals. The author aimed at demonstrating how the usage of floating-point actually accelerated the performance of the host microprocessor, and how the related benefits in terms of code density compensated for the extra area occupation required by the FPU itself. Moreover, the floating-point acceleration is obtained in a seamless way: some common DSP kernels implemented with the C programming language were compiled into executable code running on a physical prototype of the SoC equipped with the proposed FPU (Milk). This showed how the Milk FPU runs with code generated automatically by the GCC compiler, and demonstrated that the entire SoC actually works in real life and not only in simulations.
The author together with F. Garzia performed the work described in the publication: the FPGA implementation of two SoCs based on two different microprocessors and some peripherals. Both systems included the Milk FPU. D. Rossi helped interfacing Milk with one of the microprocessors. The author carried out the benchmarking activity of the performance of the two systems.
The author entirely wrote the publication, while the other authors gave valuable comments.

In some clusters Milk floating-point coprocessor was introduced by the author.

Publication [P5]. This publication was entirely planned by the author. The aim was to create a first synthesizable and scalable VHDL model of a coarse-grain reconfigurable architecture which tried to exploit the god point of the state of the art, while introducing as a novelty the fact of being VHDL-based and open-source. The author planned also to demonstrate how the proposed machine can actually speed-up the implementation of a few DSP kernels.
The author together with F. Cinelli produced the VHDL simulation model of a coarse-grain reconfigurable architecture named Butter. The author and D. Rossi studied the implementation of some DSP algorithms on it.
The author entirely wrote the publication, while the other authors gave valuable comments.

Publication [P6]. This publication was entirely planned by the author. This publication develops further the work published in [P5], and exploits partly the knowledge built in [P3].
The purpose of this paper is to demonstrate that it is possible to add support for floating-point computation in CGRAs at the expenses of introducing only a minimal area overhead. This is the first and only example of existing CGRA featuring such functionalities, which allows to extend significantly the application fields where it is applicable.
The author carried out all the work described in the publication; the author added the support for floating-point multiplication to the reconfigurable machine Butter, and mapped on it some algorithms in order to benchmark the solution proposed. Among these, there are some kernels commonly used in 3D graphics applications, demonstrating how the proposed enhancement actually widens the application domain where Butter can be applied.
The author entirely wrote the publication, while the other authors gave valuable comments.

Publication [P7]. This publication was entirely planned by the author. The purpose of this paper was to discuss several debug and test methodologies, trying to identify a combination of them and of new ones, in order to guarantee a fast and accurate verification method. This is a crucial issue, since nowadays test and verification time is about three quarters of the entire design and development process.
The author carried out all the work described in the publication: the verification of a FPU using a combination of methodologies, to guarantee a complete test coverage in

short time.
The author entirely wrote the publication, while the other authors gave valuable comments.

Publication [P8]. This publication was planned by F. Garzia together with the author. The aim of this paper was to study the feasibility of implementing fine-grain algorithms on the proposed architecture. In particular, these algorithms belong to the tracking channel stage of a GPS receiver; this aims at widening the scope of applicability of Butter.
The author of this thesis proposed to provide the Butter reconfigurable architecture with hardware support for the implementation of a GPS receiver. The author of this thesis and R. Mastria developed the necessary modifications to the hardware of Butter. F. Garzia provided the configuration memories for Butter and carried out their integration inside an existing SoC. L. Nieminen helped providing support concerning the GPS application.
F. Garzia and the author of this thesis wrote the publication together, while the other authors gave valuable comments.

Publication [P9]. This publication was entirely planned by the author of this thesis. The aim was to summarize the main research results obtained by the author about accelerating microprocessors.
The author carried on most of the work described in the publication, which is mainly a summary of the work described in some of the previous publications, extended with more recent figures and features about the proposed accelerators, and with a theoretical chapter about the acceleration of microprocessors.
The author entirely wrote the publication, while the other authors gave valuable comments.

Publication [P10]. This publication was planned by C. Mucci. C. Mucci proposed a set of dedicated tools aimed at easing the life of application developers. The author provided a GUI-based debugging tool for the Milk FPU.
C. Mucci wrote most of the publication, while the author contributed to the part concerning the GUI-based debugger. The other authors gave valuable comments.

Publication [P11]. This publication was planned by F. Garzia together with the author of this thesis. The purpose of this paper was to demonstrate the effectiveness of Butter in performing kernels belonging to the domain of video processing appli-

cations, widening the application field of Butter. Moreover, in this paper a 16-bit version was proposed, in order to measure the related area reduction when compared to the 32-bit version. Finally, a first version of a GUI-based tool for the generation of the configware for Butter was proposed.
The author and F. Garzia proposed to develop a 16-bit version of the Butter reconfigurable architecture and to map it on FPGA for debugging and benchmarking purposes. The author, F. Garzia, and A. Ferro made the modifications to the hardware of Butter and its implementation on the FPGA board. The author together with A. Ferro studied and mapped an image filter on the Butter Architecture. A. Ferro and F. Garzia took care of the development of a GUI-based tool to ease the development and mapping of algorithms on Butter. F. Garzia wrote most of the publication, the author of this thesis contributed to the parts concerning the hardware modifications to Butter, while the other authors gave valuable comments.

Publication [P12]. This publication was planned by F. Garzia together with the author. The aim was to demonstrate a physical prototype of the version of Butter featuring floating-point arithmetic capabilities. A few kernels belonging to the field of 3D graphics were then mapped on Butter, for verification and benchmarking purposes.
F. Garzia and D. Rossi performed the mapping and the benchmarking of a matrix-vector multiplication on Butter. F. Garzia wrote most of the publication, the author of this thesis contributed to the parts concerning the hardware modifications to Butter, while the other authors gave valuable comments.

Publication [P13]. This publication was entirely planned by the author. The purpose of this publication is to show the results achieved by providing the Butter accelerator with a DMA logic, and to propose a solution particularly oriented to FPGA implementations, which are the natural target implementation for CGRA templates like Butter. The author of this thesis and C. Giliberto conceived and implemented at first an external DMA controller. C. Giliberto then continued its development, optimization and integration inside Butter, as well as taking care of the benchmarking of some applications. F. Garzia helped optimizing the compilation flow and the applications mapped. The author of this thesis entirely wrote the publication, while the other authors gave valuable comments.

Publication [P14]. This publication was planned by F. Campi together with the author of this thesis. The purpose of this publication is to gather under the same umbrella several examples of reconfigurable architectures proposed during the last years. This

paper discusses several different approaches to reconfigurability, analyzing strong and weak points of each of them, and trying to define a potential future direction for the related research. Each of the authors of this paper contributed by gathering material and writing portions of the paper, according to the respective areas of expertise.

Publication [P15]. This publication was entirely planned by the author. The purpose of this publication was to provide innovative solutions to the area overhead problems affecting CGRAs. In particular, in this paper the author of this thesis provided a version of Butter able to perform not only floating-point arithmetic and 32-bit integer arithmetic as already done before, but also subword SIMD integer operations in parallel. This way the reusing of the internal components of the cell inside Butter becomes intense, and the applications can vary between a broad range. Due to the subword SIMD operations also applications characterized by relatively fine-grain.
The author carried out all the work described in the publication: based on the work presented in [P3], [P5], and [P6], the author added the support for subword SIMD operations featuring saturating arithmetic to the reconfigurable machine Butter, and mapped on it some algorithms in order to benchmark the solution proposed. The author entirely wrote the publication, while the other authors gave valuable comments.

Publication [P16]. This publication was entirely planned by the author. The purpose of this publication is to summarize the results achieved with the design of Milk, and to study its impact on complex applications. This publication thus summarizes the results achieved in publications [P1], [P2], [P4], and explores further new aspects of the Milk architecture. For instance, a study has been performed to measure the impact of the communication overhead on the local bus between Milk FPU and the host microprocessor, comparing it with a version of Milk integrated directly in the datapath of the host microprocessor. Moreover, an FPGA-optimized version of the VHDL code of Milk has been proposed and compared to the generic one.
The author of this thesis performed all the work related to the Milk FPU, its design exploration, all its variations and its benchmarking. The author of this thesis entirely wrote the publication, while the other authors gave valuable comments.

6. CONCLUSIONS

The following section discusses the main results of this thesis work, while the next section gives the directions for future research.

6.1 Main results obtained

Considering the diverse requirements dictated by modern applications and the set of constraints related to embedded systems, the author came up with the description of a series of possible solutions to tackle the problems mentioned in the introduction of this thesis.
The author concentrated on heterogeneous systems hosting a set of different types of computation engines, i.e. programmable microprocessors, coprocessors and dedicated accelerators.
The author then provided a more detailed description about the approach taken at Tampere University of Technology: design and implementation of two different accelerators which can be used as a coprocessor for programmable RISC cores. The coprocessors are very different from each other, empowering this way different application domains: together they can thus provide a relatively wide coverage of applications, while keeping a modular approach to the implementation of an entire system. Their performance is high enough to speed up significantly the host microprocessor; their flexibility and modularity allow also an easy adaptability of their actual implementation in order to meet different requirements.
More precisely, in this thesis work the author conceived, designed, implemented and tested a Floating-Point Unit (FPU) named Milk and a coarse-grain reconfigurable accelerator named Butter. Both the devices have been designed according to the IP-reuse philosophy, and then implemented in the form of a synthesizable VHDL model, which allows simulating their behavior and to actually implement them in hardware (either on FPGA or ASIC Standard-Cells technology). These two key points were

planned since the beginning to tackle easily some common issues explained in the introduction of this thesis. As illustrated in [P1], [P2], and [P4] Milk has been successfully included in two different systems powered by different host microprocessors. Distinguishing features of the Milk coprocessor are:

- usage of concurrent and independent functional units (FUs) operating in parallel, which can be easily inserted and removed from the datapath in a transparent way. They can be used also as stand-alone computation engines connected to the system bus; this is not usually true for other existing FPUs.
- a multi-port register file (which allows concurrent writeback of results, enabling massive parallelism in the execution of applications)
- a special register-locking mechanism and a related stall logic, which automatically stalls the host microprocessor whenever it tries to read results from the FPU which are not ready yet, or when it tried to write operands to a register in the FPU which is about to be written by the FPU itself with a result. This hardware mechanism is lightweight, and at the same time allows for simpler compilers and more efficient execution
- a hardware logic which handles the (so called) denormalized operands automatically, in order to avoid a performance degradation typically referred to as denormals bug (due to interrupt service routines handling the denormalized operands). To estimate the impact of this choice, the author compared an FPGA implementation of the multiplier inside Milk against its corresponding version without this logic: the area occupation is six times smaller, and the latency drops by one clock cycle. On the other hand, things are much better when taking into consideration the effects on the entire FPU: since the denormal handling hardware logic is shared in the Milk implementation, the actual impact in that case is much more mitigated, depending on how many functional units are instantiated. Still, the multiplier stand-alone without that logic requires only 420 Logic Elements on Stratix FPGA, and runs at 98 MHz; such results are extremely close to the ones marked by dedicated, optimized library components provided by Altera, showing how anyway the denormal support logic should be implemented only when the applications use it intensively.
- high scalability and configurability: each and every functional unit in the FPU

can be freely inserted or removed by the actual implementation of the device, allowing the designer to achieve the desired trade-off between area occupation and functionalities available. Several parameters of the device can be adjusted easily and quickly the same way: datapath width, polarity of control signals, interrupt signal masking, and so forth

- hardware support for division and square-root (sqrt). This leads to a larger area when compared to another version featuring software implementation of those functions, but also guarantees a very high performance
- technology independence (any ASIC or FPGA target technology can support the implementation equally well, due to the generality and platform/technology independence of the VHDL model developed). The author obtained prototypes working on several different FPGA boards; synthesis results were obtained also for several different target ASIC standard-cell technologies using different EDA tools, demonstrating the general validity of the VHDL models. In addition, the author implemented an FPGA-optimized version of the code of Milk for comparison against the generic one; results showed how the benefits are relatively limited, providing an improvement around 10This shows how it is much more beneficial to keep the generic version of the VHDL code.
- full compliance with the IEEE-754 standard (single precision). Among the available open-source devices, Milk is the one that implements it in the most complete way.
- totally open-source
- full GCC compiler support. As planned, the author made the proposed FPU such that it is supported by the GNU C Compiler (GCC), so that the execution of applications written in C programming language is automatic.
- a GUI-based debug interface was created for Milk, based on the GNU DDD: the user can exploit a GUI-based environment which shows the contents of the registers of Milk, the source code and its corresponding assembly code, breakpoints set, and so on; this way, the development of applications is eased significantly.

As mentioned, the FUs of Milk can be used also as stand-alone computation engines connected to the system bus; this is not usually true for other existing FPUs, which

(in order to save some area) merge and share portions of their functional units. Their solution presents as a drawback the fact that the FUs cannot run different operations in parallel, besides the fact that they cannot work as stand-alone components.

The author carried on a design space exploration, trying to achieve optimal implementation of the functional units, and exploring how the area and speed figures change according with different versions and combinations of the functional units inside Milk. The automation of this procedure is left as future work. At the moment, synthesis results show how the most complete version (thus the one with the highest area occupation and lowest operating clock frequency) requires 105 K Gates and runs at 400MHz on a low-power, 90nm ASIC standard-cells technology, and requires 20 K Logic Elements (approximately 13 K ALUTs on Stratix II FPGA device) and runs at 67 MHZ on a Stratix FPGA device. On the other hand, a lighter version (without divider and SQRT) occupies 54 K Gates and runs at 600 MHz on 90nm ASIC standard-cells technology, while it uses 8.5 K ALUTs on Stratix II FPGA, and runs at 70 MHz.
The author proposed also an integrated version of the FPU, which aims at showing the area impact of the control logic, I/O logic, register-locking logic, and register file, besides showing the effects of the communication overhead over the local bus between Milk and the host microprocessor. The integrated version of Milk occupies 10 K Gates less than the coprocessor version (which means approximately 10Still, the integrated version assumes that the user can access the internal architecture of the host processor, which is not always possible. A summary about Milk, its features and characteristics is given in [P9] and [P16], while its testing and validation is explained in [P7].
Milk is the first complete, IP reusable, scalable, high performance open source FPU (fully compliant with the IEEE-754 standard) available today.

The Butter coprocessor has been implemented as well as an IP component using a synthesizable and parametric VHDL model. The first version of Butter is presented in [P5]. Butter is a CGRA template describing an array of reconfigurable computation engines interconnected by a programmable network. A survey of several different reconfigurable architectures is presented in [P14]. Besides being exploitable as a pure hardware architecture, one of the strongest and more innovative points behind Butter is its nature as intermediate layer between applications (jam) and target

FPGA hardware (bread): Butter allows the designer to map applications from C code (or a DFG-like description) to FPGA with no need to know VHDL. This has been made possible by introducing a dedicated, GUI-based tool which enables the representation of applications in a DFG-like format. Such representation has a one-to-one correspondence with the processing elements and interconnections inside Butter. A more general presentation about the issues related to the mapping of applications on reconfigurable architectures is given in [P10].
Distinguishing features of the Butter coprocessor are:

- very coarse grain. The datapath width is a VHDL generic parameter, thus it is flexible and can be chosen freely by the designer. Still, in our current implementation we chose a 32-bit wide datapath, used to support directly the format of integer numbers, but also floating-point numbers: Butter is the first and only reconfigurable machine able to do so. Such feature is a key point: it enables efficient reuse of existing hardware resources, enabling the introduction of features which were considered inapplicable before, and opening up a new and broad horizon of potential application which can benefit from CGRAs.

- support for integer SIMD subword operations: the same computation engine can process either 32-bit, 16-bit, or 8-bit integer data in parallel, as explained in [P15]. The introduction of tis feature is based on previous studies explained in [P3]. Due to the innovative architecture of the cell, Butter is able to reuse heavily and efficiently the same internal sub-block of each cell in order to implement several different operations. This feature is aimed at mitigating the area overhead typically present in CGRAs when executing algorithms featuring fine-grain data types; this feature significantly enlarges the range of the applications which can be mapped on Butter, and solves the key problem of bridging the gap between CGRAs and fine-grain reconfigurable machines.

- internal DMA logic which reduces the effects of the bus bottleneck when accessing the system memory, as illustrated in [P13]. In particular, when targeting FPGA implementations, it is possible to exploit a time-multiplexed approach.

- features two local memories which are used as buffer for data: their role can be either as input or output memory, behaving like an hourglass. This feature,

like the previous one, is also aimed at solving the well known problem of the communication overhead which plagues CGRAs in general.

- scalability and portability: several architectural parameters (i.e. the amount of computation elements, the amount of rows and columns inside the array, the operations actually implemented in each cell, the type of interconnections, etc.) can be easily and quickly modified by the designer just by setting some special VHDL generics in a configuration file.

- dedicated GUI-based tool for easy programming of the array. The tool is based on Java (to allow portability) and represents graphically the appearance of the Butter array and its interconnections. The user can specify operations and interconnections for each cell, achieving this way a DFG-like description of the algorithm to be mapped on Butter. The tool then generates automatically the bit-stream (configware) that will be used in practice to configure the hardware array in order to implement the desired application. An advantage of our approach and programming model is that once the bit stream has been created for a given function (for instance, a FIR filter), it is generated as a C header file from our GUI-based tool. This way that header file can be used anytime as a library component belonging to a set of accelerated functions that can be used inside any other C application file. It is thus possible to build an entire library of C functions which are directly executable Butter; those functions can thus be freely inserted by software designers inside any of their applications.

- Provides an easy and intuitive interface to enable application mapping on FPGA

- parametric template, which allows optimal implementation size and feature mix

Synthesis results indicate that the area occupation and the operating frequency of Butter are roughly the same as the ones scored by other similar design. In addition to this, the amount of clock cycles taken by Butter to perform a given algorithm is orders of magnitude smaller than the one required by a corresponding software implementation on a RISC microprocessor, demonstrating the effectiveness of the approach.
We made a comparison between synthesis results obtained for a version of our architecture not able to perform subword nor floating-point operations, a version capable of subword computation only, and a version which can perform all of them. From an

analysis of the figures we can see how *the introduction of floating-point multiplication capability comes at almost no cost*, as shown in [P6]; this is a very interesting point, mainly due to the observation that the integer multiplier and adder required inside the floating-point multiplier are already present inside the cell of our architecture, then the additional area is consumed by dedicated rounding and packing logic. The floating-point addition is more invasive, since we had to insert inside the cell some additional, dedicated components like leading-zero counters, comparators, and multiplexers, which are quite resource consuming on FPGA technologies. The same applies to the introduction of subword capabilities: the routing and multiplexing necessary to allow the implementation of saturating subword operations increased the amount of ALUTs required to map a cell of our architecture. Still, the subword cell is in principle capable of executing in parallel four operations, which would then require four separate cells of our machine using the traditional cell, justifying this way the area overhead introduced. Using the normal version of the cell, our machine (together with his host system) requires 35828 ALUTs, and the maximum working frequency is 45 MHz. Using cells with subword capabilities (but no floating-point) inside our architecture, then 37347 ALUTs are used (and 216 DSP blocks), and the maximum working frequency is equal to 45 MHz. We can conclude that on FPGA the ALUT usage ratio is about 1.04 and the speed ratio is 1.5 when passing from the simplest version of the cell to the one with subword capabilities. Still, it should be remembered that exploiting the subword capabilities it is possible to map on a single cell an amount of different operations which would instead require up to 4 elementary cells. Taking into account the average case (a single subword-enabled cell can be used instead of two elementary cells) the result is that the actual amount of cells needed to map a given kernel is half the amount that would be needed using the regular cell, leading to the conclusion that in fact the ALUT usage ratio is in practice better when using subword-enabled cells. Due to the introduction floating-point support it is possible to exploit our machine for a number of important applications that so far could not be mapped on state of the art reconfigurable architectures, like 3D graphics algorithms.

Several applications were mapped on a system hosting both Milk and Butter as coprocessors to a RISC core (Coffee RISC or Qrisc). For example, screensaver-like 3D applications, channel tracking of a GPS receiver [P8], Mp3 decoder, H.264 decoder,

and a set of common DSP kernels (FFT, FIR, IIR, etc...). In [P11] is illustrated the mapping on Butter of some image filtering algorithms, while in [P12] is analyzed the implementation of floating-point matrix-vector multiplications, used extensively in 3D graphics applications. The execution time of algorithms mapped on Butter is in general shortened by one to three orders of magnitude when compared to a corresponding software implementation running on a programmable RISC microprocessor. An overview of Butter is presented in [P9] and [P15].

Butter is the coarsest grain available today in reconfigurable machines (together with ADRES), allowing a thorough study of the potential of this choice and the related trade-offs. The author pushed the research way further, coming to propose the first CGRA ever which is able to elaborate floating-point numbers. This choice was never proposed before, mainly due to the relatively large area occupation implied by floating-point hardware. This point induced the belief within the research community that such a feature cannot possibly be applied to the field of CGRAs. The author demonstrated instead that it is possible to have such a feature in a CGRA without increasing significantly its area occupation. This has been made possible by a proper reuse of the existing hardware resources. What's more, this choice opened up the horizon for an unprecedented usage of CGRAs in several application domains.

By designing, verifying, doing the design space exploration and FPGA prototyping of these two open-source, IP-reusable and portable coprocessors, the author contributed to the state-of-the-art in accelerator design.

6.2 Future Development

The main results obtained in this thesis can be briefly summarized as follows.

- The author proposed two coprocessors to be used as accelerators for a programmable microprocessor. As explained in the publications, the proposed accelerators can boost the performance of the host microprocessor by orders of magnitude in a series of algorithms belonging to a broad range of applications, from automotive to DSP, from 3D graphics to audio and video compression, from GPS to image processing. Powering so many and so different application domains using just two accelerators represents a significant advantage in terms

of area occupation and simplicity of the system architecture (not to mention the flexibility) when compared to traditional approaches based on the usage of multiple ASIC accelerators.

- The proposed accelerators have been designed in such a way that their usage in other systems is very straight-forward, thanks to their interface which wraps them hiding the internal details and exposing to the external world only a few, key ports. They are thus excellent stand-alone IP components, ready to be plugged into virtually any digital system: the author demonstrated this feature by successfully including the proposed accelerators into different systems powered by different microprocessors.

- The author used VHDL language to implement both the coprocessors. Introducing in the VHDL code several generics, the author ensured the delivery of flexible designs, whose key parameters could be easily adjusted at taste by the final user. This turned out to be very convenient when performing comparisons between different versions of the proposed CGRA (based on different width of its datapath). The VHDL model produced has been mapped over a number of different ASIC standard cells technologies using different synthesis tools. The same model could also be mapped on different FPGA technologies using different tools; the FPGA suppport has been exploited also for real implementation and prototyping on different FPGA boards, demonstrating how the VHDL model proposed is characterized by general validity and can be actually used in real life implementations.

- The author proposed an FPU which is fully compliant with the IEEE-754 standard, and which supports the GNU GCC compiler. These features ensure general validity to the proposed design. In particular, the GCC support ensures the possibility of running complex applications (written using the C programming language) with no need for the user to intervene with manual activity. The author also introduced a porting of the GNU DDD debugger, which is very useful during the development of the C code of the applications meant to run on the proposed FPU.

- Similarly, a dedicated tool based on a graphical interface is proposed in order to ease the task of mapping kernels of algorithms on the proposed CGRA. This tool involves some manual activity by the user, which is supposed to draw a

DFG-like scheme of the algorithms to be mapped. Still, after this stage the tool is able to generate automatically the configware which is used to program the CGRA, relieving the user from taking care of this tedious and time-consuming task.

- The author proposed an innovative CGRA which is provided with special features aimed at enhancing its performance while reusing existing resources. In particular, the author proposed the first CGRA architecture in the world able to support directly floating-point operations. Such an innovation boosts significantly the performance achievable by the proposed coprocessor, also widening the potential application fields. The author demonstrated how the introduction of such capabilities comes at a relatively small extra cost in terms of area occupation due to an attentive reuse of existing hardware resources.

- The proposed CGRA has also been provided with the ability of supporting subword SIMD operations. Also this feature does not introduce a significant extra cost, since it is based on existing resources. The benefits in terms of usability are instead noticeable: such a solution enables bridging the gap between the very coarse grain of the proposed hardware and the relatively fine grain of several algorithms. This way, the same hardware can support well relatively fine grain algorithms while being (together with ADRES proposed by IMEC) the CGRA with the coarsest grain available today.

- The proposed accelerators are completely open source, enabling the research community with valuable tools and designs supporting future research.

As a future work, a new tool is envisioned which allows the user to evaluate the area occupation and speed figures for different instantiations of Milk and Butter. For instance, a library of different versions of the functional units of Milk could be provided, letting the user select them depending on the requirements and the constraints of the particular applications he/she intends to run. Such a tool would exploit the features of Milk and facilitate achieving an efficient implementation. The same applies to Butter: such a tool would ease the task of identifying the best mix of features for heterogeneous implementation of its cells. This way the user can have a particular instance of Butter which guarantees the required functionalities at the minimum possible cost in terms of area occupation. Cooperation with another research group

at Tampere University of Technology in order to implement such a tool is currently under discussion.

An interesting enhancement from a usability point of view would be providing the compiler for Coffee with the ability of automatically identify suitable computation kernels and map them on Butter.

From a hardware point of view, Milk could be extended to support double precision floating-point data (64-bit), thus finalizing the work done already for the multiplier: Milk could then have all its FUs supporting either one double floating-point operation, two paired (concurrent) single precision floating-point operations, or a number of paired SIMD subword integer operations. Moreover, the integer part of the floating-point FUs could be merged with the corresponding FUs of Coffee (i.e. the integer multiplier), thus significantly decreasing the amount of hardware used exclusively for integer processing.
An asynchronous version of Milk could be implemented: this would be an interesting experiment, in order to verify how much area and energy can be saved when compared to the current implementation.

The architecture of the Butter could be enhanced so that more application domains could benefit from its usage; in particular, applications belonging to the field of Software Defined Radio could represent an interesting and innovative application field, which would benefit largely by a high-performance and flexible machine such as Butter.

Butter could be equipped with a hardware mechanism that allows for self-triggering of the execution when data are available. This dataflow-like model of computation would avoid the need for the microprocessor to trigger explicitly the execution, saving this way some communication overhead.

BIBLIOGRAPHY

[1] C.-H. Jeong, W.-C. Park, T.-D. Han, and S.-D. Kim, "Cost/Performance Trade-off in Floating-Point Unit Design for 3D Geometry Processor", in *Proceedings of the IEEE Asia Pacific Conference on ASIC (AP-ASIC 1999)*, Seoul, Korea, Aug 23-25 1999, pp. 104–107.

[2] Theo A.C.M. Claasen, "System on a Chip: Changing IC Design Today and in the Future", in *IEEE Micro*, Vol.23, May/June 2003, pp. 20–26.

[3] D. Sima, T. Fountain, and P. Kacsuk, *Advanced computer architectures: a design space approach*. Addison Wesley, 1997.

[4] J. Silc, B. Robic and T. Ungerer *Processor architecture: from dataflow to superscalar and beyond*. Springer, 1999.

[5] S. Leibson, Lower SoC operating frequencies to cut power dissipation, *Portable Design*, Feb.2004.

[6] G. Frantz and R. Simar, Comparing Fixed and Floating-Point DSPs, Texas Instruments,
URL:www.ti.com/etechsept04fltgptwhpaper (visited on January 2007).

[7] M. Kuulusa, DSP Processor Core-Based Wireless System Design, Tampere University of Technology, Doctoral Thesis, publication 296, 2000.

[8] J. Hennessy and D. Pattersson *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2007.

[9] W. M. Johnson, *Superscalar Processor Design*. Prentice Hall, 1991.

[10] C. Panis, Scalable DSP Core Architecture Addressing Compiler Requirements, Tampere University of Technology, Doctoral Thesis, publication 483, 2004.

[11] S.J. Eggers, J.S. Emer, h.M. Leby, J.L. Lo, R.L. Stamm and D.M. Tullsen, “Simultaneous multithreading: a platform for next-generation processors” *IEEE Micro*, Vol.17, Sept/Oct 1997, pp. 12–19.

[12] A. Douillet and G.R. Gao, “Software-Pipelining on Multi-Core Architectures” *IEEE International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania,15-19 Sept 2007, pp. 39–48.

[13] G. Martin, ”Overview of the MPSoC Design Challenge”, in *Proceedings of the ACM Design Automation Conference (DAC 2006)*, San Francisco, California, USA, July 24-28 2006, pp. 274–279.

[14] URL:www.altera.com (visited on January 2007).

[15] URL:www.xilinx.com (visited on January 2007).

[16] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., 1985.

[17] I. Kuroda, *Digital Signal Processing for Multimedia Systems*, K.K. Parhi and T. Nishitani Eds., Marcel Dekker Inc, 1999.

[18] URL:www.gaisler.com (visited on January 2007).

[19] URL:www.opencores.org (visited on June 2007).

[20] Marcus, G., Hinojosa, P., Avila, A., and Nolazco-FIores, J.: ”A Fully Synthesizable Single-Precision, Floating-Point Adder/Substractor and Multiplier in VHDL for General and Educational Use”, IEEE, Proc. International Caracas Conference on Devices, Circuits and Systems, 2004, Volume 1, 3-5 Nov. 2004 Page(s): 319 - 323

[21] Jeong, C.-H., Park, W.-C., Kim, S.-W., and Han, T.-D.: ”The Design and Implementation of CalmlRISC32 Floating-Point Unit”, Proc. Second IEEE Asia Pacific Conference on ASICs, 2000 (AP-ASIC 2000), 28-30 Aug. 2000 Page(s): 327 - 330

[22] URL:www.arm.com (visited on June 2007).

[23] C. Rowen, *Engineering the Complex SoC*. Prentice/Hall International, 2004.

[24] A. Ferrari and A. Sangiovanni-Vincentelli, "System design: traditional concepts and new paradigms," in *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, Austin, USA, October 1999.

[25] M. Hunt and J. A. Rowson, "Blocking in a system on a chip," *IEEE Spectrum*, November 1996, pp. 35–41.

[26] A. M. Rincon, C. Cherichetti, J. A. Monzel, D. R. Stauffer, and M. T. Trick, "Core design and system-on-a-chip integration," *IEEE Design and Test of Computers*, vol. 14, no. 4, pp. 26–35, October-December 1997.

[27] M. Keating and P. Bricaud, *Reuse Mehodology Manual for Sysem-On-A-Chip Designs*, 3rd ed. Norwell, Massachusetts, USA: Kluwer Academic Publishers, 2002.

[28] J. Becker, L. Kabulepa, F.M. Renner and M. Glesner, "Simulation and Rapid Prototyping of Flexible Systems-on-a-Chip for Future Mobile Communication Applications", in *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, Paris, France, 21-23 June 2000, pp. 160–165.

[29] Dfpau floating-point Pipelined Divider (visited on April 2008), URL:http://www.altera.com

[30] QxFP floating-point functional units from Quixilica (visited on April 2008), URL:http://www.xilinx.com

[31] J. Kylliäinen, J. Nurmi, and M. Kuulusa, "COFFEE - a Core for Free", in *Proceedings of the International Symposium on System-on-Chip (SoC2003)*, Tampere, Finland, 19-21 November 2003, pp. 17–22.

[32] ARCES laboratories, *The XiRISC processor research project* (visited on May 2004). University of Bologna, URL:http://xirisc.deis.unibo.it

[33] Department of Computer Systems, *The COFFEE™RISC Core research project* (visited on April 2008). Tampere University of Technology, URL:http://coffee.tut.fi

[34] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable Computing: Architectures and Design Methods", *IEE Proc. Computers and Digital Techniques*, vol. 152, no. 1, March 2005, pp.193–207

[35] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective", in *Proceedings of the International Conference in Automation and Test in Europe (DATE 2001)*, Munich, Germany, 13-16 March 2001, pp. 642–649.

[36] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix", in *International Conference on Field Programmable Logic and Applications (FPL 2003)*, Lisboa, Portugal, 1-3 September, 2003, pp. 61–70.

[37] S. Vassiliadis, J. Silc, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov and E. Moscu Panainte, "The MOLEN Polymorphic Processor", *IEEE Transactions on Computers*, Vol. 53, No. 11, November 2004, pp.1363–1375.

[38] E. Mirsky, A. DeHon, "MATRIX: A reconfigurable Computing Architecture with Configurable Instruction and Deployable Resources", in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 96)*, Napa Valley, CA, USA, 17-19 April, 1996, pp.157–166.

[39] T. Ristimäki and J. Nurmi, "Reprogrammable Algorithm Accelerator IP Block", in *Proceedings of the IFIP International Conference on VLSI-SOC (VLSI-SOC 2003)*, Darmstadt, Germany, 1-3 December 2003, pp. 228–232.

[40] T. Ristimäki, C.Brunelli and J. Nurmi, "Back-End Tool Flow for Coarse Grain Reconfigurable IP Block RAA", in *Proceedings of the IP based SoC Design Conference and Exhibition (IP/SOC 2006)*, Grenoble, France, 6-7 December 2006, pp. 201–206.

[41] T. Miyamori and K. Olukotun, "A quantitative analysis of reconfigurable coprocessors for multimedia applications" in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, USA, 15-17 April 1998, pp. 2–11.

[42] P.M. Heysters, G.K. Rauwerda and L.T. Smit: *A Flexible, Low Power, High Performance DSP IP Core for Programmable Systems-on-Chip* (visited on April 2008),
URL:http://www.us.design-reuse.com/articles/article12159.html

[43] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe and R. Reed Taylor: PipeRench: A Reconfigurable Architecture and Compiler, *IEEE Computer*, Vol. 33, No. 4, April 2000, pages 70-77.

[44] www.siliconhive.com (visited on April 2008)

[45] J.R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor", in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, USA, April 1997, pp.12–21.

[46] R. Hartenstein, M. Hertz, T. Hoffmann and U. Nageldinger: Using the Kress Array for Reconfigurable Computing, *Configurable Computing: Technology and Applications, Proceedings of SPIE* Vol. 3526, 1998, pp.150–161.

[47] H. Singh, M.H. Lee, G. Lu, F.G. Kurdahi, N. Bagherzadeh, T. Lang, R. Heaton and E.M.C. Filho, "MorphoSys: An Integrated Re-Configurable Architecture", in *Proceedings of the NATO International Symposium on System Synthesis*, Monterey, CA, USA, April, 1998.

[48] F. Campi and C. Mucci, "Run-Time Reconfigurable Processors", J.Nurmi (Ed.), *Processor Design-System-On-Chip Computing for ASICs and FPGAs*, Springer, 2007, pp.177–208.

[49] A. Major, I. Nousias, S. Khawam, M. Milward, Y. Yi and T. Arslan, "H.264/AVC In-Loop De-Blocking Filter Targeting a Dynamically Reconfigurable Instruction Cell Architecture," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, Edinburgh, UK, 5-8 August 2007, pp.134–138.

[50] J. Becker, A. Thomas, M. Vorbach and V. Baumgarte, "An Industrial/Academic Configurable System-on-Chip Project (CSoC): Coarse-grain XPP/Leon-based Architecture Integration", in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, Munich, Germany, 3-7 March 2003, pp.1120–1121.

[51] A. Lodi, C. Mucci, M. Bocchi, A. Cappelli, M. De Dominicis and L. Ciccarelli, A Multi-Context Pipelined Array for Embedded System, in *International Conference on Field Programmable Logic and Applications (FPL 2006)*, Madrid, Spain, 28-30 August 2006, pp. 581–588.

[52] M. Galanis, G. Theodoridis, S. Tragoudas, D. Soudris and C. Goutis, Mapping DSP Applications to a High-Performance Reconfigurable Coarse-Grain Data-Path, in *International Conference on Field Programmable Logic and Applications (FPL 2004)*, Leuven, Belgium, 30 August -1 September 2004, pp. 868–873.

[53] G. Dimitroulakos, M. Galanis and C. Goutis, Performance Improvements Using Coarse-Grain Reconfigurable Logic in Embedded SoCs, in *International Conference on Field Programmable Logic and Applications (FPL 2005)*, Tampere, Finland, 24-26 August 2005, pp. 630–635.

[54] B. Mei, F. Veredas and B. Masschelein, Mapping an h.264 decoder onto the ADRES reconfigurable architecture, in *IEEE International Conference on Field Programmable Logic and Applications (FPL 2005)*, Istanbul, Turkey, 15-18 June 2006, pp. 288–291.

[55] K. Paulsson, M.Hubner, and J. Becker, Strategies to On- Line Failure Recovery in Self- Adaptive Systems based on Dynamic and Partial Reconfiguration, in *IEEE NASA/ESA Conference on Adaptive Hardware and Systems*, Tampere, Finland, 24-26 August 2005, pp. 622–625.

[56] S. Bayar, and A. Yurdakul, Dynamic Partial Self-Reconfiguration on Spartan-III FPGAs via a Parallel Configuration Access Port (PCAP), in *Proceedings of 2nd HiPEAC Workshop on Reconfigurable Computing*, Goteborg, Sweden, January 27, 2008, pp.1–10.

[57] Nios microprocessor (visited on April 2008), URL:http://www.altera.com/products/ip/processors/nios/nio-index.html

[58] Microblaze microprocessor (visited on April 2008), URL:http://www.xilinx.com/products/

[59] P. Sedcole, B. Blodget, T. Becker, J. Anderson and P. Lysaght "Modular Dynamic Reconfiguration in Virtex FPGAs" *IEE Proceedings on Computers and Digital Techniques*, Vol. 153, No. 3, May 2006, pp.157–164.

[60] S. Shukla, N.W. Bergman and J. Becker, "QUKU: A Two-Level Reconfigurable Architecture", in *IEEE International Symposium on VLSI 2006 (ISVLSI06)*, Karlsruhe, Germany, 2-3 March 2006.

[61] H. Singh, M.H. Lee, G. Lu, F.G. Kurdahi, N. Bagherzadeh, "MorphoSys: An Reconfigurable Architecture for Multimedia Applications", Proc. of the XI Brazilian Symposium on Integrated Circuit Design, Rio De Janeiro, Brasil, September 1998, p134.

[62] G. Lu, H. Singh, L. Ming-Hau, N. Bagherzadeh, F.J. Kurdahi, E.M.C. Filho and V.Castro-Alves, "The MorphoSys Dynamically Reconfigurable System-on-Chip", Proc. of the First NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, July 1998, pp.152-160.

[63] A.H. Kamalizad, C. Pan and N. Bagherzadeh, "Fast Parallel FFT on a Reconfigurable Computation Platform", Proc. of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD03), Sao Paulo, Brazil, November 2003, p.254

[64] F.J. Veredas, M. Scheppler, W. Moffat and M. Bingfeng, "Custom Implementation of the Coarse-Grained Reconfigurable ADRES Architecture for Multimedia Purposes", Proc. of the International Conference on Field Programmable Logic and Applications (FPL2005), Tampere, Finland, August 2005, pp.106-111

[65] P.M. Heysters and G.J.M. Smith, "Mapping of DSP algorithms on the Montium Architecture", Proc. of the Parallel and Distributed Processing Symposium (IPDPS2003), Nice, France, April 2003, p.6

Part II

PUBLICATIONS

PUBLICATION 1

Reprinted, with permission, from *Proceedings of the International Symposium on System-on-Chip*, (SoC 2003), pages 127–130, Digital Object Identifier 10.1109/ISSOC.2003.1267734 Tampere, Finland, 19–21 November 2003, M. Bocchi, C. Brunelli, C. DeBartolomeis, L. Magagni and F. Campi, "A system level IP integration methodology for fast SoC design".

PUBLICATION 2

Reprinted, with permission, from *Proceedings of the International Symposium on System-on-Chip*, (SoC 2004), pages 103–106, Digital Object Identifier 10.1109/ISSOC.2004.1411160 Tampere, Finland, 16–18 November 2004, C. Brunelli, F. Campi, J. Kylliäinen and J. Nurmi, "A Reconfigurable FPU as IP component for SoCs".

PUBLICATION 3

Reprinted, with permission, from *Proceedings of the IEEE International Workshop on Signal Processing Systems*, (SiPS2005), pages 70–74, Athens, Greece, 2–4 November 2005, Digital Object Identifier 10.1109/SIPS.2005.1579841 C. Brunelli, P. Salmela, J. Takala and J. Nurmi, "A Flexible Multiplier for Media Processing".

PUBLICATION 4

Reprinted, with permission, from *Proceedings of the International Symposium on System-on-Chip*, (SoC 2005), pages 29–32, Digital Object Identifier 10.1109/ISSOC.2005.1595636 Tampere, Finland, 15–17 November 2005, C. Brunelli, F. Garzia, C. Mucci, F. Campi, D. Rossi and J. Nurmi, "A FPGA Implementation of An Open-Source Floating-Point Computation System".

PUBLICATION 5

Reprinted, with permission, from *Proceedings of the International Conference on Ph.D. Research in Microelectronics and Electronics*, (PRIME2006), pages 229–232, Otranto, Italy, 12–15 June 2005, C. Brunelli, F. Cinelli, D. Rossi and J. Nurmi, "A VHDL Model and Implementation of a Coarse-Grain Reconfigurable Coprocessor for a RISC Core".

PUBLICATION 6

Reprinted, with permission, from *Proceedings of the International workshop on Reconfigurable Communication-centric Systems-on-Chip*, (ReCoSoC2006), pages 1–7, Montpellier, France, 03-05 July 2006, C. Brunelli, F. Garzia and J. Nurmi, "A Coarse-Grain Reconfigurable Machine With Floating-Point Arithmetic Capabilities".

PUBLICATION 7

Reprinted, with permission, from *Proceedings of the International Symposium on System-on-Chip*, (SoC2006), pp. 87–90,
Digital Object Identifier 10.1109/ISSOC.2006.321974 Tampere, Finland, 14-16 November 2006, C. Brunelli and J. Nurmi, "Design and Verification of a VHDL Model of a Floating-Point Unit for a RISC Microprocessor".

PUBLICATION 8

Reprinted, with permission, from *Proceedings of the International Conference on Computer as a Tool (Eurocon2007)*, pp. 875–881, Digital Object Identifier 10.1109/EURCON.2007.4400609 Warsaw, Poland, 9-12 Sept. 2007, F. Garzia, C. Brunelli, L. Nieminen, R. Mastria and J. Nurmi, "Implementation of a Tracking Channel of a GPS receiver on a Reconfigurable Machine".

PUBLICATION 9

PUBLICATION 10

PUBLICATION 11

PUBLICATION 12

Reprinted, with permission, from *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, Miami, Florida, USA, April 14-15 2008, Digital Object Identifier 10.1109/IPDPS.2008.4536538 F. Garzia, C. Brunelli, D. Rossi and J. Nurmi, "Implementation of a floating-point matrix-vector multiplication on a reconfigurable architecture".

PUBLICATION 13

Reprinted, with permission, from *Proceedings of the International Conference on Field Programmable Logic and Applications*, (FPL 2008), pages 487–490, Digital Object Identifier 10.1109/FPL.2008.4629990 Heidelberg, Germany, 8–10 September 2008, C. Brunelli, F. Garzia, C. Giliberto and J. Nurmi, "A Dedicated DMA Logic Addressing a Time Multiplexed Memory to Reduce the Effects of the System Bus Bottleneck".

PUBLICATION 14

Reprinted, with permission, from *Proceedings of the International Conference on Field Programmable Logic and Applications*, (FPL 2008), pages 409–414, Digital Object Identifier 10.1109/FPL.2008.4629972 Heidelberg, Germany, 8–10 September 2008, C. Brunelli, F. Garzia, J. Nurmi, F. Campi, and D. Picard, "Reconfigurable Hardware: the Holy Grail of Matching Performance with Programming Productivity".

PUBLICATION 15

PUBLICATION 16